VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

HLUBOKÉ NEURONOVÉ SÍTĚ: IMPLEMENTACE PRO VESTAVĚNÉ SYSTÉMY

DEEP NEURAL NETWORKS: EMBEDDED SYSTEM IMPLEMENTATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. ALEŠ MATĚJ**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. VOJTĚCH MRÁZEK**
SUPERVISOR

BRNO 2017

Zadání diplomové práce/15852/2017/xmatej48

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů　　　　　　　　　　　　　　　　　Akademický rok 2017/2018

Zadání diplomové práce

Řešitel: **Matěj Aleš, Bc.**
Obor: Bioinformatika a biocomputing
Téma: **Hluboké neuronové sítě: implementace pro vestavěné systémy**
Deep Neural Networks: Embedded System Implementation
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s problematikou hlubokých konvolučních neuronových sítí a jejich efektivní implementací ve vestavěných systémech založených na jádru procesoru ARM Cortex řady M.
2. Vyberte si vhodnou úlohu pro demonstraci využití konvolučních neuronových sítí (např. rozpoznávání číslic).
3. Navrhněte aplikaci pro vestavěný systém realizující konvoluční neuronovou síť ve zvolené úloze. Při návrhu minimalizujte energetické a paměťové nároky.
4. Implementujte navrženou aplikaci. V největší míře využijte specializovaných instrukcí pro zpracování dat.
5. Zhodnoťte dosažené výsledky a diskutujte možnosti pokračování projektu.

Literatura:

1. Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

1. Splnění bodů 1 a 2 zadání.

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

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

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

Vedoucí: **Mrázek Vojtěch, Ing.**, UPSY FIT VUT
Datum zadání: 1. listopadu 2017
Datum odevzdání: 23. května 2018

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

prof. Ing. Lukáš Sekanina, Ph.D.
vedoucí ústavu

Abstrakt
Cílem této práce je první navrhnout a implementovat aplikaci pro vestavěné systémy realizující konvoluční neuronovou síť jenž klasifikuje čísla MNIST, ve druhé části pak optimalizovat paměťové a energetické nároky této sítě. Práce v teoretické části popisuje základy neuronových sítí a výpočetní platformy Cortex-M pro vestavěné systémy. Následuje popis implementace, síť je první vytvořena a naučena pomocí knihovny Theano v Pythonu na PC a poté je převedena do C pro vývojovou desku STM32F429 Discovery, kde je následné optimalizována. Optimalizace je zaměřena na konvoluci, skalární součin a formát uložení vah a biasů sítě.

Abstract
The goal of this thesis is to firstly design and implement an application for embedded systems which will classify MNIST numbers and secondly optimize energy and memory requirements of this network. The basics of neural networks, Cortex-M processor cores and embedded devices are described in the theoretical part. Followed by implementation details. Networks learning is done with Python and Theano library on a PC. The network is then converted to C for a board STM32F429 Discovery. Last part consist of network optimization, which focuses on convolution, dot product and number representation of weights and biases of the network.

Klíčová slova
hluboké konvoluční neuronové sítě, vestavěné systémy, ARM, Cortex-M, DSP, optimalizace, STM32F429 Discovery, MNIST

Keywords
deep convolutional neural networks, embedded systems, ARM, Cortex-M, DSP, optimalization, STM32F429 Discovery, MNIST

Citace
MATĚJ, Aleš. *Hluboké neuronové sítě: implementace pro vestavěné systémy.* Brno, 2017. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vojtěch Mrázek

Hluboké neuronové sítě: implementace pro vestavěné systémy

Prohlášení

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

. . . .
Aleš Matěj
20. května 2018

Poděkování

Chci poděkovat svému vedoucímu Ing. Vojtěchu Mrázkovi za vedení práce, rady k technické realizaci i k textu práce.

Obsah

1 Úvod **3**

2 Neuronové sítě **4**

2.1 Síť neuronů . . . 4
2.1.1 Neuron . . . 4
2.1.2 Síť . . . 5
2.2 Vícevrstvá neuronová síť . . . 6
2.3 Konvoluční neuronová sít . . . 6
2.3.1 LeNet-5 . . . 7
2.4 Použití . . . 7
2.5 Učení . . . 8
2.5.1 S učitelem . . . 8
2.5.2 Bez učitele . . . 8
2.5.3 Gradial descent . . . 8
2.5.4 Backpropagation . . . 9
2.5.5 Přetrénování . . . 9

3 Výpočetní platformy pro vestavěné systémy **10**

3.1 ARM procesory . . . 10
3.2 Cortex-M . . . 10
3.2.1 Cortex-M0 a Cortex-M0+ . . . 12
3.2.2 Cortex-M1 . . . 12
3.2.3 Cortex-M4 a Cortex-M3 . . . 12
3.2.4 Cortex-M7 . . . 12
3.3 STM32F429 Discovery . . . 13
3.3.1 ST-LINK/V2 . . . 14
3.3.2 General purpose Input/Output (GPIO) . . . 14
3.3.3 Obecný časovač . . . 14
3.4 Paměť . . . 14
3.5 NVIC . . . 15
3.6 Pohyblivá řádová čárka . . . 15
3.7 Zpracování digitálních signálů . . . 16
3.7.1 Optimalizace v DPS . . . 16
3.8 Programování vestavěných zařízení . . . 16
3.8.1 Datové typy . . . 17
3.8.2 SDK . . . 17
3.8.3 STM32CubeF4 . . . 17
3.8.4 HAL (*Hardware Abstraction Layer*) . . . 18

3.8.5 LL (*Low-Layer*) 18
3.8.6 CMSIS 18
3.9 Neuronové sítě 19

4 Návrh a implementace **21**
4.1 MNIST databáze 21
4.2 Vymezení problému 21
4.3 Učení v PC 22
4.3.1 Knihovny a frameworky pro sítě 22
4.3.2 Architektura sítě 25
4.3.3 Učení sítě 26
4.3.4 Export vah a biasů z vrstev naučené sítě 26
4.4 Klasifikace v MCU 27
4.4.1 Programování 27
4.4.2 Uložení vah a biasů sítě 28
4.4.3 Uložení dat MNIST 28
4.4.4 Aktivační funkce 28
4.4.5 Vrstvy 29

5 Optimalizace **32**
5.1 Profilace slabých míst implementace 32
5.2 Skalární součin 32
5.3 Konvoluce 33
5.4 Reprezentace čísel 34

6 Vyhodnocení **36**
6.1 Způsob měření 36
6.1.1 Měření vnitřním čítačem 36
6.1.2 Měření osciloskopem 37
6.2 Porovnání implementací 37
6.2.1 Náhrada DSP funkcí 37
6.2.2 Porovnání dílčích funkcí 37
6.2.3 Výsledné porovnání sítí 38

7 Závěr **40**

Literatura **42**

A Obsah přiloženého paměťového média **45**

B Měření postupné optimalizace **46**

Kapitola 1

Úvod

Neuronové sítě jsou stále populárnější a pronikají do čím dál více aplikací, mobilní vestavěné systémy nevyjímaje. V současnosti poskytují nejlepší řešení k mnoha problémům v disciplínách jako rozpoznání obrazu, zpracovaní přirozeného jazyka a rozpoznání řeči.

Tyto problémy, pro nás tak jednoduché a přirozené, jako čtení tohoto textu nás klamou. Jednoduché totiž rozhodně nejsou. Evoluce nás vybavila miliardami neuronů elegantně propojených a zabalených do formy mozku, které se z části specializují přesně na tyto úkoly. Mozek je něco jako superpočítač, vyladěný během milionů let na porozumění vizuálního světa, téměř bez našeho vědomí vykoná všechnu práci za nás. Pokud ale chceme takové rozpoznání a klasifikaci vzorů algoritmizovat, velmi rychle zjistíme o jak obtížný úkol se jedná. Jako nejúspěšnější v jejich řešení se zatím ukázaly hluboké, rekurentní a, zejména pro zpracování obrazu, konvoluční neuronové sítě.

Většinu těchto aplikací chceme také integrovat do našeho každodenního života, a to znamená malá přenosná zařízení se spoustou senzorů a čidel. Tato zařízení se tak připojují ke stále většímu fenoménu internetu věcí (*IoT* – *Internet of Things*) a i pro ně je tedy důležitá výdrž baterie, nízká cena a samozřejmě korektní funkčnost.

Pokrok v technologii baterii je pomalý v porovnání s vývojem informatiky a nestíhá tak držet krok. Jsme proto nuceni optimalizovat výdrž jinými způsoby. Jedna z energeticky nejnáročnějších částí celého systému *IoT* je přenos obrovského množství dat od senzorů k výpočetním jednotkám. Bylo by tedy možné dosáhnout značné úspory energie, pokud by se data zpracovala přímo v zařízení se senzorem, čímž by se redukovalo množství dat pro přenos. Tato práce se zabývá efektivní implementací hlubokých konvolučních neuronových sítí ve vestavěných systémech založených na jádru procesoru ARM Cortex-M. Je rozebrána teorie neuronových sítí, způsob fungování procesoru Cortex-M a efektivní implementace softwaru na zařízeních obsahující Cortex-M procesor.

Konkrétně je práce rozdělena následovně. Kapitola 2 se zabývá samotnými neuronovými sítěmi, od základů jejich prvků až po způsoby učení. V další kapitole číslo 3 jsou rozebrány především Cortex-M procesory a konkrétní použitá vývojová deska. Kapitola 4 se věnuje návrhu a implementačním detailům práce, definuje se základní rozdělení na část prováděnou v PC a část patřící do MCU. Je také popsána použitá hluboká konvoluční neuronová síť a její jednotlivé vrstvy. V kapitole 5 se zabýváme optimalizací sítě, postupně jsou předvedeny přístupy, které byly použity k urychlení výpočtu nebo k redukci využití paměti ve vestavěném zařízení. Výsledky, způsoby měření a porovnání naměřených hodnot se zabývá kapitola 6.

Kapitola 2

Neuronové sítě

Přístup neuronových sítí k řešení problémů je odlišný od klasické algoritmizace. Na rozdíl od klasických programů, tedy sekvenci operací, se umělé neuronové sítě učí samy z množiny trénovacích dat, typicky čím větší tím lepší. Ta například obsahuje dvojce představují vstup a požadovaný výstup. Může to tak být bitmapa a číslo které je na ní zobrazeno nebo signál a slovo které reprezentuje. Trénovací algoritmus neuronové sítě během fáze učení automaticky odvodí pravidla pro zadaný problém z trénovacích dat.

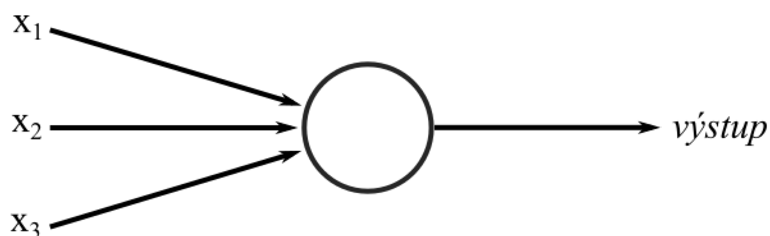
Každou neuronovou síť můžeme abstrahovat na funkci. Na vstupu má určité množství parametrů, kterých je typicky obrovské množství. Parametry představují váhy a aktivační prahy neuronů i samotný vstup sítě. Tedy například pixely obrázku nebo úseky signálu. Výstup funkce je pak to, co od sítě chceme. Například hodnota, která určí, jaké číslo bylo na obrázku nebo co za slovo je zakódováno v signálu. Při použití sítě měníme pouze tu část parametrů, která reprezentuje vstup sítě (obrázek, signál), ale při učení sítě manipulujeme všechny parametry této teoretické funkce.

2.1 Síť neuronů

Různé druhy neuronů a různé způsoby jejich propojení (topologie), určují druh neuronové sítě a tvoří tak jejich klasifikaci.

2.1.1 Neuron

Původním a nejjednodušším druhem neuronu je Perceptron. Perceptron, ilustrován na obrázku 2.1, může mít libovolný počet vstupů a pouze jeden výstup. Výstup je ale možné rozmnožit do velkého množství následujících perceptronů. Jednotlivé vstupy jsou vynásobeny dopředu specifikovanými váhami a sečteny, pokud výsledná hodna přesáhne aktivační práh, je neuron aktivní.



Obrázek 2.1: Obecný model neuronu

$$výstup = \begin{cases} 0 & pokud \;\; w \cdot x + b \leq 0 \\ 1 & pokud \;\; w \cdot x + b > 0 \end{cases} \tag{2.1}$$

Z důvodů přehlednější a jednodušší notace se aktivační práh zapisuje jako bias, podle rovnice: $bias\ (b) = -threshold$. Bias nám říká, jak jednoduché je, aby se neuron aktivoval. Sumu násobků vah a hodnot vstupů perceptronu, můžeme zapsat také jako skalární součin vektorů vah w a vstupů x tedy: $\sum_j w_j x_j = w \cdot x$.

Chování perceptronu lze tedy popsat podle rovnice 2.1. Tento model neuronu však není používaný, protože je nevhodný pro učení sítě [21]. Při učení sítě potřebujeme malou změnou váhy nebo biasu neuronu docílit malé změny výstupu. Perceptron však může i při malé změně nastavení změnit chovaní pro konkrétní hodnoty mezi nulou a jedničkou, čímž výrazně ovlivní celkové chování sítě. Tato vlastnost by učení značně komplikovala, proto používáme model umělého neuronu, který má vstupy a výstupy ve spojitém intervalu mezi nulou a jedničkou. Sigmoid neuron je jeden z neuronů mající tuto vlastnost.

Sigmoid neuron, stejně jako perceptron, má vstup ve formě vektoru x, jehož položky ale nabývají hodnot $x_i \in < 0, 1 >$. Dále má také jako perceptron pro každý vstup váhu w_i a celkový bias. Výstup sigmoidu, při jeho aktivaci, je však řízen sigmoidou, aktivační funkcí 2.2. Existují další druhy neuronů s různými aktivačními funkcemi [4], jako například hyperbolický tangens nebo *ReLu*. Hyperbolický tangens (*tanh*) odpovídá funkci sigmoid vynásobené konstantou, tato vlastnost znamená, že *tanh* má větší gradient, což je důležité při učení sítě. Naopak *ReLu* funkce je význačná svou jednoduchostí a rychlostí, odpovídá výrazu $ReLu(x) = max(0, x)$. Funkce *ReLu* může výrazně urychlit proces učení.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

2.1.2 Síť

Typická síť má jako nejlevější (první) vstupní vrstvu, nejpravější (poslední) výstupní vrstvu a mezi nimi skryté vrstvy. Určit počet skrytých vrstev a počet neuronů v nich není jednoduchý problém. Často se používají různé heuristiky pro odhady správných počtů. Například jak nastavit poměr počtu skrytých vrstev vůči době trénování sítě.

Sítě můžeme podle architektury rozdělit na:

Dopředné

Výstup vrstvy je použit jako vstup do následující vrstvy. To znamená, že v síti nejsou žádné smyčky, informace jde vždy pouze dopředu. Tato kategorie je v současnosti nejrozšířenější. Učení dopředných sítí je jednodušší než učení rekurentních.

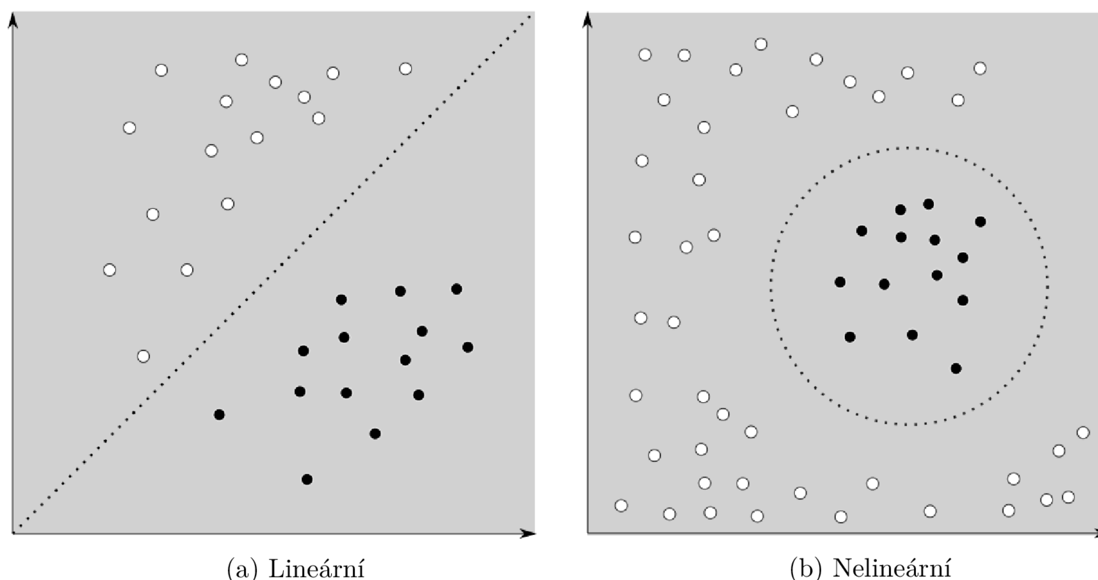
Rekurentní

Tyto sítě jsou principiálně o krok blíže fungování mozku a umožňují řešit některé problémy, které jsou klasickými dopřednými neřešitelné. Příkladem může být zpracování videa, kdy dopředná síť může brát v potaz v jeden moment jen jeden snímek, zatímco rekurentní síť může pracovat i se svým předchozím výstupem (předchozím snímkem), tím lze zajistit větší kontinuitu videa [10].

Obecně vstup sítě závisí na výstupu sítě v minulém čase. Je možné tak docílit kaskády postupně aktivujících se neuronů.

2.2 Vícevrstvá neuronová síť

Více vrstevní neuronová síť má alespoň tři vrstvy neuronů. Navíc dokáže klasifikovat také data, jež nejsou lineárně oddělitelná, ilustrováno na obrázku 2.2. První vrstva dělá jednoduchá rozhodnutí na základě vstupních hodnot. Každá další (skrytá) vrstva dělá rozhodnutí na základě výsledků předchozí vrstvy. Tedy čím hlubší vrstva tím komplexnější a abstraktnější jsou rozhodnutí. Síť s mnoha vrstvami může dělat sofistikované rozhodování.



(a) Lineární (b) Nelineární

Obrázek 2.2: Oddělitelnost dat

Hluboké sítě

Hluboké neuronové sítě nejčastěji označují sítě s alespoň dvěma skrytými vrstvami. Tyto sítě nejdou prakticky učit tradičním algoritmem backpropagation, který je představen v pozdější kapitole, proces je příliš pomalý.

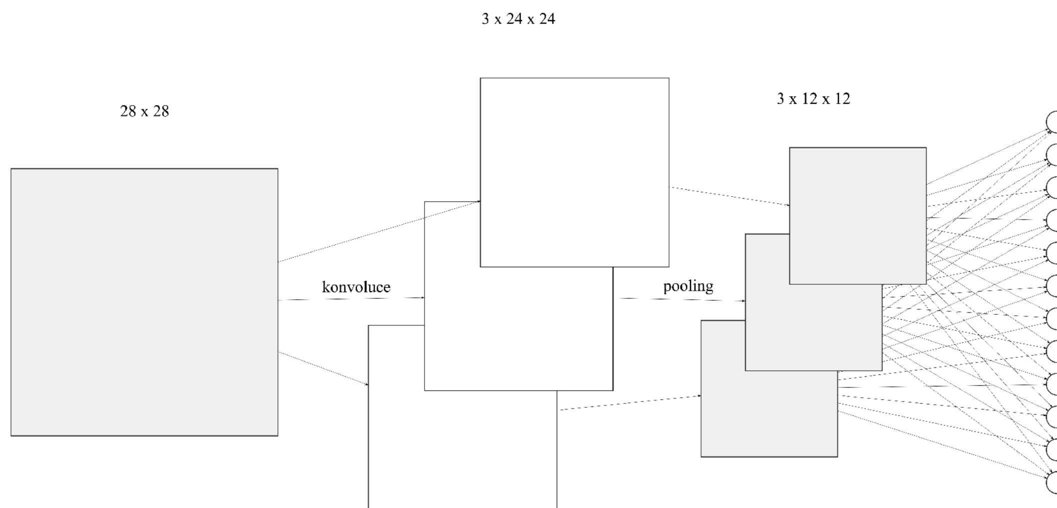
2.3 Konvoluční neuronová sít

Pro klasifikaci obrazu není vhodné používat plně propojené sítě. Plně propojená síť nebere v potaz rozložení obrazu, tedy dva pixely, které leží vedle sebe jsou zpracovány stejně jako dva pixely ležící v opačných rozích. Síť tak musí tuto prostorou informaci odvodit z trénovacích dat. Z tohoto důvodu používáme konvoluční sítě, jejich architektura je navržena pro práci s obrazy. Vstupní vrstva je rozložena jako matice, tedy už se na ni nedíváme jako na vektor. Matice odpovídá rozložením vstupnímu obrázku, neurony mají hodnotu pixelů na dané pozici, tato vlastnost je důležitá pro napojenou konvoluční vrstvu. Tyto konvoluční vrstvy, blíže popsány dále, je možné propojit s plně propojenými nebo libovolnými jinými vrstvami, zapadají tak do systému neuronových sítí.

V konvoluční vrstvě sdílí neurony váhy a biasy. Síť už není plně propojená s předchozí vrstvou, každý neuron přijímá vstup jen z několika pixelů (vstupních neuronů), které jsou umístěny vedle sebe, například 5x5 okno. Předchozí vrstva je tak postupně procházena posuvným oknem, kde každé okno patří jednomu neuronu v následující vrstvě. Dvojice

okno a připojený neuron sdílí nastavení vah a biasů pro celou vrstvu. Jde tak říci, že tím definují jádro nebo filtr, celá vrstva tak detekuje jednu vlastnost.

Pooling vrstvy jsou většinou použity okamžitě po konvolučních vrstvách a v podstatě zjednodušují výstup konvolučních vrstev. Typický je například *max-pooling*, kdy například z 2x2 regionu vybereme pouze největší hodnotu. Můžeme tak zredukovat výstup z konvoluční vrstvy 24x24 na 12x12. Stejně jak konvoluce je aplikovaná s různými jádry i *pooling* je aplikováno na každý výstup samostatně v rámci jedné vrstvy. Celkově *pooling* zjednoduší a zrychlí výpočet. Minimálně pro výstup konvoluční síť také obsahuje vrstvu plně propojenou s výstupem *poolingu*. Obrázek 2.3 ukazuje základní prvky konvoluční sítě.



Obrázek 2.3: Příklad jednoduché konvoluční sítě

2.3.1 LeNet-5

Jedna z prvních konvolučních sítí byla představena v roce 1994 jejímž autor byl Yann LeCun. Jedná se o speciální druh více vrstvé neuronové sítě. Jako většina ostatních sítí je trénována pomocí *backpropagation* algoritmu, je ale odlišná od tehdejších sítí především svou architekturou. Tato síť, označovaná jako LeNet-5, protože obsahuje 5 vrstev (konvoluční, *pooling*, konvoluční, *pooling* a plně propojené neurony), je navržena pro rozpoznání ručně psaných a tištěných znaků [18]. Podstatným krokem vpřed, který významně zlepšil klasifikační schopnost tohoto modelu, bylo právě použití konvoluce, ta umožnila identifikovat podobné obrazové prvky, jež mohou být rozprostřeny přes celý obraz. Redukce propojení i sdílení vah a biasů také velmi ulehčila trénování. Poslední vrstvy sítě byly klasické architektury a plného propojení pro závěrečnou klasifikaci.

2.4 Použití

Samotné dopředné vyhodnocení sítě je oproti učení jednoduchá záležitost. Podle rovnice 2.3, lze vypočítat postupně všechny vrstvy neuronové sítě, včetně poslední, která tak dá výsledek. Index l reprezentuje pořadí vrstvy v síti. w^l je matice vah neuronů pojící l-tou vrstvu neuronů. b^l je vektor bias hodnot patřící l-té vrstvě neuronů. Poslední součástí je vektor a^l, v němž jsou aktivační hodnoty pro l-tou vrstvu. Aktivační hodnota pro daný neuron, je výsledek zvolené aktivační funkce, která měla jako vstup sumu násobků vah a

aktivačních hodnot z předchozí vrstvy plus konkrétní bias. V obecné rovnici 2.3 je použita aktivační funkce σ, ta však může být zvolena libovolně. Použití funkce na výsledný vektor je rovno použití funkce na každý člen vektoru samostatně.

$$a^l = \sigma(w^l a^{l-1} + b^l) \tag{2.3}$$

2.5 Učení

Učení neuronových sítí je proces postupného nalezení co nejlepších vah a *biasů* sítě, který je výpočetně náročný. Typicky potřebujeme obrovské množství dat nebo nějaký způsob, jak je generovat, na kterých budeme síť učit. Pro některé problémy tak existují dostupné sady s deseti tisíci položkami. Těch je ale velmi málo a získání dat představuje velkou výzvu.

Každé učení potřebuje cenovou funkci *cost function*. *Cost function* nám říká, jak dobrý je aktuální stav. Typicky tak funkce počítá nějakou sumu ohodnocení všech trénovacích vstupů. Tedy spustí algoritmus sítě pro každý trénovací vstup s aktuálními váhami a biasy a výsledek porovnává se správným ohodnocení, to musí být součástí trénovacích dat, pokud provádíme učení s učitelem. Učení je jen minimalizace této funkce.

Počítat cenovou funkci tímto způsobem je vhodné zejména díky jejímu hladkému průběhu. V principu by to totiž stačilo maximalizovat počet správně zařazených trénovacích dat, ale opět, stejně jako u perceptronu, potřebujeme, aby malá změna ve váhách nebo biasech byla promítnuta jako malá změna ve výsledku, to nám umožní efektivnější učení.

Minimalizace funkce je obecný problém, řešitelný pomocí spoustu přístupů. Hledáme konkrétně takový, který si poradí s komplikovanými funkce v mnoha, někdy miliardy, dimenzích, protože jednotlivé dimenze představují váhy a biasy. Důležitými faktory jsou také rychlost a efektivita výpočtu, protože tuto minimalizaci budeme opakovat a počítat mnohokrát. Toho lze dosáhnout například pomocí algoritmu *Gradial Descent*. Nezávisle na algoritmu můžeme učení rozdělit do dvou základních kategorií.

2.5.1 S učitelem

Základem učení s učitelem jsou trénovací data. Jedná se o dvojce vstup a požadovaný výstup. Cílem učení je pak najít funkci, respektive parametry sítě, jenž tvoří funkci, tak aby, pro každou dvojičku, první člen dvojce byl namapován na druhý člen. Toho se snažíme dosáhnout minimalizací cenové funkce. Takto se trénuje síť pro například rozpoznání vzorů (klasifikace), aproximace funkcí (regrese) a aplikace s kontinuálními daty [23].

2.5.2 Bez učitele

Síť dostane pouze data a nějakou funkci nad danými daty. Takto funkce se typicky minimalizuje, ale je možné hledat i maximum. Tvar funkce je odvislý od druhu aplikace, tedy funkce pro kompresi bude odlišná od funkce pro shlukování. Mezi další aplikace učení bez učitele patří filtrace, odhad statistické distribuce a obecně další odhady.

2.5.3 Gradial descent

Algoritmus funguje na základě opakovaného výpočtu gradientu *cost function* a provedení malého kroku v opačném směru. Velikost kroku je určena koeficientem učení, což je malé nezáporné celé číslo. Velikost koeficientu učení je odvislá od několika podmínek. Koeficient musí být dostatečně velký, aby algoritmus reálně někdy skončil. Koeficient představuje

velikost kroku ve směru k minimu funkce. Zároveň však musí být koeficient dostatečně malý, aby nedošlo k překročení minima. Jeho hodnotu tak můžeme v průběhu výpočtu upravovat. Provádět velké kroky ze začátku a s blížícím se řešení krok zmenšovat, tedy zvyšovat přesnost.

Gradial descent je možné si představit jako puštění míčku do údolí, míček se zastaví na prvním minimu i lokálním. Jedním z problémů toho přístupu je výpočetní náročnost. Musíme spočítat gradient pro každý trénovací vstup. Toto lze řešit úpravou algoritmu na stochastický gradientní sestup . Tedy skutečný gradient aproximujeme pomocí výpočtu několika náhodně zvolených trénovacích vstupů. Tato množina se nazývá *mini-bach*. Učení tak můžeme rozdělit do *Epoch*, kdy každá epocha představuje postupné trénování se všemi možnými *mini-batch* trénovací sady.

2.5.4 Backpropagation

Dnes nejpoužívanější algoritmus pro učení se nazývá *backpropagation*, algoritmus první projde síť klasickým způsobem dopředu, poté spočítá chybu a tu propaguje postupně, skrze jednotlivé vrstvy, zpátky na začátek sítě [19].

Tento algoritmus je však nevhodný pro učení hlubokých sítí ve své výchozí podobě. Problém je pomalé učení hlubokých sítí. Konkrétně čím více má síť vrstev, tím pomaleji se vrstvy vzdálené od konce, odkud se propaguje chyba, učí. Tento problém se obecně nazývá *unstable gradient problem*. Podstata je v tom, že gradient počátečních vrstev je násobek následujících vrstev. Pro backpropagation je tak síť s více vrstvami automaticky nestabilní. Toto nastane i pokud na začátku zvolíme počáteční váhy velké, protože chceme kompenzovat malé změny.

2.5.5 Přetrénování

Přetrénování nastane, pokud výsledný model (nastavení sítě) odpovídá přesně nebo je velmi podobný trénovacím datům, čímž přestává odpovídat testovacím datům [2]. Takový model špatně zobecňuje. Nejjednodušší metoda, jak předejít přetrénování je sledovat výstup testů sítě na testovacích datech a zastavit učení v momentě, kdy se síť přestane zlepšovat. Podobné strategie založené na tomto principu se nazývají *early stopping*. Další velmi efektivní způsob, jak zabránit přetrénování, je jednoduše zvětšit množinu trénovacích dat. Při velké množině trénovacích dat je i pro velké sítě velmi těžké dojít do bodu přetrénování. Naneštěstí získat trénovací data může být velmi drahé nebo náročné, takže toto není vždy možné.

Dále existuje metoda nazývaná *regularizace*. Ta zahrnuje několik přístupů jako udržovaní vah sítě na malých hodnotách, dočasné odstranění některých neuronů během učení nebo umělé rozšíření trénovacích dat. Data lze většinou rozšířit několika způsoby, odvislých od typu dat. Například pro obrázek je možné použít jednoduchou rotaci nebo translaci.

Kapitola 3

Výpočetní platformy pro vestavěné systémy

Vestavěné systémy jsou specializované systémy na vykonávání několika konkrétních funkcí [3]. Díky této specializaci je možné výrazně optimalizovat celkovou velikost, cenu, příkon, spolehlivost i výkon. Jsou běžně vyráběny ve velkých nákladech pro další úsporu nákladů. Vestavěný systém je nejčastěji řízen mikrokontrolerem, který ovládá další přítomný hardware.

Mezi typické příklady vestavěných systémů patří routery, přepínače, digitální hodinky a například MP3 přehrávače. Složitostí tak postihují spektrum od jednoprocesorových čipů až po pokročilé vysoce výkonné jednotky s několika procesory.

3.1 ARM procesory

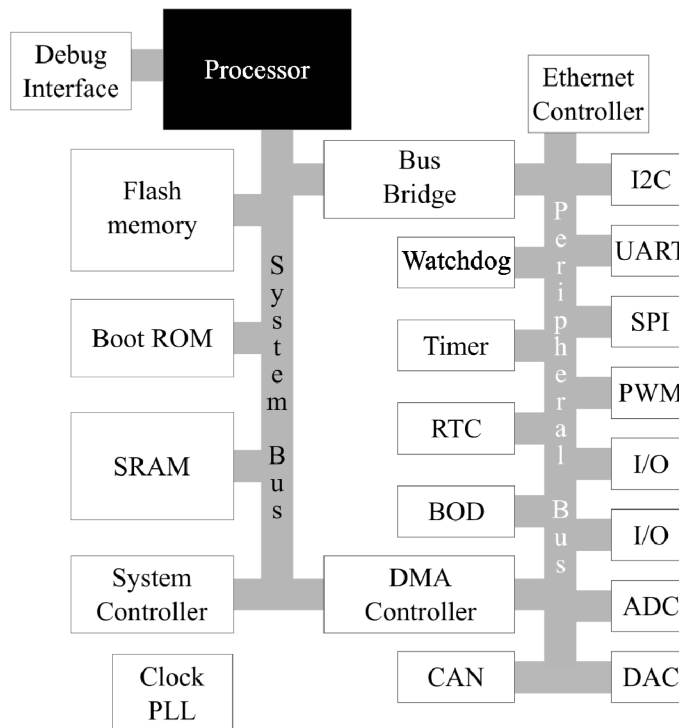
Je důležité rozlišit mezi jádry procesorů a mikrokontrolery. Firma Arm navrhuje jádra procesorů a případně další komponenty, které si pak licencují jednotlivý výrobci a doplňují je o další periferie, tím vnikají produkty jako jsou například mikrokontrolery. Schéma mikrokontroleru je ilustrováno na obrázku 3.1. Jak lze vidět mikrokontroler obsahuje spoustu různých bloků jako sběrnice, paměti, časovače a další periferní zařízení, procesor tvoří jen malou část celkového návrhu. I když mnoho výrobců mikrokontrolerů používá jádro ARM jako CPU, další zmíněné bloky se mohou velmi různit mezi jednotlivými produkty. Arm distribuuje jádro ve formě RTL[1] schéma.

Dnes jsou velmi rozšířená jádra Cortex, firma Arm má tři hlavní větve Cortex-A, Cortex-R a Cortex-M. Každá je navržená s odlišným cílem. Profil A je zaměřený na vysoce výkonné aplikace. Profil R pro pokročilé vestavěné systémy, kde je potřeba zpracování v reálném čase. Poslední Cortex-M cílí na menší aplikace jako mikrokontrolery, kde jsou hlavní kritéria cena, nízký příkon, malá latence přerušení a jednoduchost použití. Tato práce je zaměřená právě na řadu M, která je pro zpracování dat například v uzlech IoT nejvhodnější.

3.2 Cortex-M

Rodina jader procesorů Cortex-M firmy Arm je populární a vyskytuje se v celé řadě mikrokontrolerů od různých výrobců. Obecně jsou považovány za RISC, tedy *Reduced Instruction*

[1] *Register Transfer Logic* je abstrakce používaná v jazycích popisující HW jako je Verilog a VHDL pro tvorbu vysoko úrovňové reprezentace obvodu.



Obrázek 3.1: Schéma mikrokontroleru [30]

Set, procesory s menší instrukční sadou, ale s postupem vývoje technologie procesorů je tento rozdíl vůči CISC (*Complex Instruction Set* – procesory se spoustou druhů instrukcí), stále méně patrný [30]. Všechny verze používají 32 bitovou architekturu, to znamená že všechny registry, rozhraní sběrnice i datové cesty mají šířku 32 bitů. Procesory Cortex-M existují v různých verzích, specializované na konkrétní aplikace. Přehled je ukázán na obrázku 3.2. Správná volba konkrétního zařízení se zvoleným jádrem není jednoduchá a závisí na celé řade požadavků jako cenová dostupnost, výkon, dostupnost obecně, bezpečnost firmware[2], přítomnost různých periferních zařízení a mnoho dalšího.

Mezi výhody Cortex-M patří:

- nízký příkon,
- vysoká výkonnost,
- energetická úspornost,
- konfigurovatelná přerušení,
- jednoduché použití, díky například lineárně mapované paměti a absenci omezení architektury,
- škálovatelnost,
- dostupnost a rozšíření,
- přenositelnost a znovupoužití software.

[2]Firmware je program vestavěný do hardware zařízení a je jeho nezbytnou součástí [13].

3.2.1 Cortex-M0 a Cortex-M0+

Tyto jádra procesorů jsou založeny na ARMv6-M architektuře a mají malou instrukční sadu. Celkově mají relativně malé množstvím hradel, okolo 12000. Jsou tak vhodné pro levné a úsporné mikrokontrolery. Jsou specializované na nízký příkon a energetickou úsporu. To ale znamená, že pro komplexní systémy na zpracování dat nejsou vhodné a je lepší zvolit vyšší verze rodiny Cortex-M.

3.2.2 Cortex-M1

I procesor Cortex-M1 je založen na ARMv6-M, nicméně je navržen výhradně pro FPGA aplikace. Díky vysoké úrovni optimalizace dovoluje v pokročilých FPGA provádět operace na vysokých kmitočtech hodin.

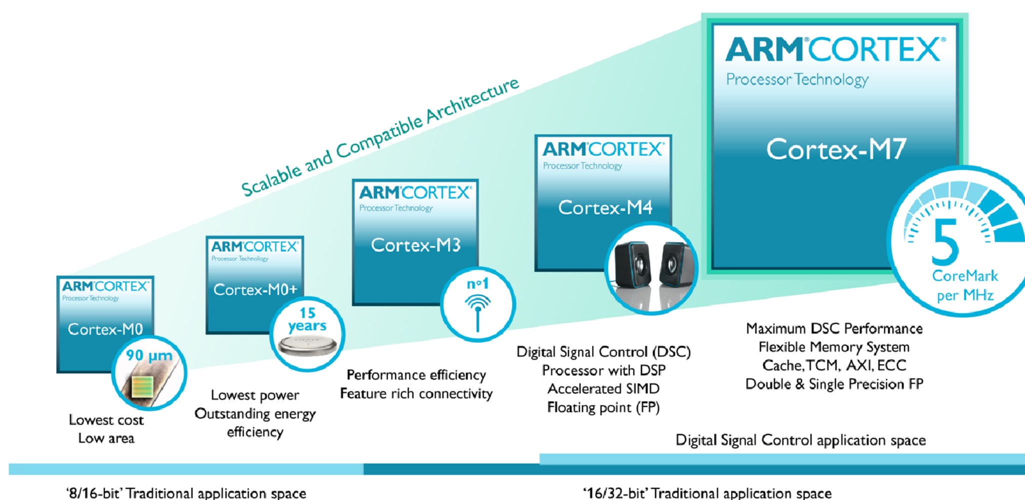
3.2.3 Cortex-M4 a Cortex-M3

Konkrétní procesory Cortex-M3 a Cortex-M4 jsou široce používané v dnešních vestavěných zařízeních s mikrokontrolery, pro které jsou speciálně navrženy. Vzájemně si jsou velmi podobné a většinou podporují shodné instrukce. Oba používají tří stavové zpracování instrukcí, tedy načtení, dekódování a vykonání. Řada Cortex-M4 je ale rychlejší zejména ve zpracování digitálních signálů. Protože podporuje operace s pohyblivou řádovou čárkou je možné některé instrukce vykonat v méně hodinových cyklech.

Stejně jako všechny procesory řady Cortex-M používají 32 bitové adresování, je tak možné pracovat s maximálně 4 GB paměti pro libovolný mikrokontroler. Tento prostor sdílí všechno na čipu: kód programu, data, periferie i komponenty podporující debuggování.

3.2.4 Cortex-M7

Cortex-M7 je nejvýkonnější člen rodiny. Poskytuje všechny možnosti Cortex-M3 a Cortex-M4 s vylepšenou podporou operací s plovoucí řádovou čárkou. Tento procesor cílí na aplikace s rozpoznáním řeči, zpracování obrazu a automobilový průmysl.

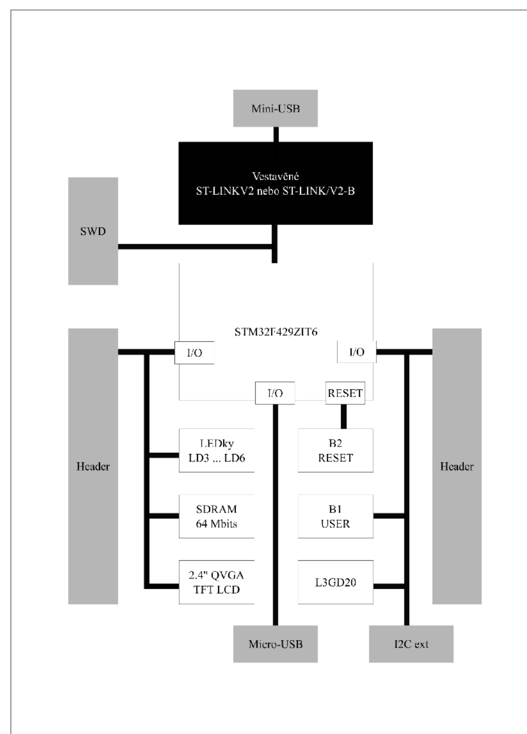


Obrázek 3.2: Rodina procesorů Cortex-M [1]

3.3 STM32F429 Discovery

Konkrétní vývojová deska, na kterém bude práce implementována je STM32F429 Discovery, ta používá ARM Cortex-M4, 32 bitové RISC jádro operující na maximální frekvenci 180 MHz. Jádro má FPU jednotku, jenž podporuje všechny ARM 32 bitové instrukce pro zpracování dat i jejich datové typy, dále také skupinu DSP (zpracování digitálních signálů) instrukcí a jednotku MPU pro ochranu paměti. Čip je součástí vývojové desky, která je svou nízkou cenou a jednoduchým použitím zaměřená na rychlý vývoj s STM32F429 mikrokontrolery. Deska je postavená kolem STM32F428ZIT6 mikrokontroleru, obrázek 3.3 ilustruje jeho propojení s periferními zařízeními jako ST-LINK, tlačítka, LEDky, USB, gyroskop a další. [26] Mezi klíčové vlastnosti zařízení patří:

- Mikrokontroler s SRAM (256 KiB), Flash (2048 KiB),
- SDRAM s 8 MB,
- vestavěný ST-LINK/V2,
- USB funkce (port pro debuggování, virtuální COM port, zdroj napájení),
- 2.4"QVGA TFT LCD,
- 6x LED,
- pohybový senzor (3 osy),
- a dvě tlačítka.



Obrázek 3.3: Ilustrace propojení mezi mikrokontrolerem a jeho periferiemi

3.3.1 ST-LINK/V2

Významnou součástí je programovací rozhraní ST-LINK/V2, jedná se vestavěný debugger a programátor pro STM8 a STM32 rodiny mikrokontrolerů, je tedy pomocí ní možné nahrávat kód a data do zařízení. Pro komunikaci se samotným mikrokontrolerem a ST-LINK/V2 jsou použita rozhraní SWIM (jeden vodič) a JTAG/SWD (*Serial Wire debugging*). Pro umožnění programování zařízení je také nutné aktivovat dva *jumpery* v sekci CN4 na desce zařízení. ST-LINK/V2 lze napájet pomocí USB připojení 5 V a skrze USB lze také komunikovat. ST-LINK/V2 disponuje LED zobrazují status komunikace s PC, LED bliká, pokud probíhá komunikace.

3.3.2 General purpose Input/Output (GPIO)

GPIO jsou piny na desce integrovaného obvodu jejichž chování je definováno za běhu uživatelem, ten může dokonce určit, zda se jedná o vstupní nebo výstupní pic. Každé GPIO označené písmeny jako A, B, C atd. má deset 32 bitových konfiguračních registrů. Registry je možné nastavit například rychlost, výstupní stav, výstupní data, zamykací mechanismus a další. Podle specifických hardware charakteristik konkrétního I/O zařízení je možné každý bitový port individuálně nastavit do několika módů: například analogový, pull-up/down vstup (určují logickou hodnotu na vodiči, pokud není definována žádným zařízením), otevřený kolektor (výstupní pin se chová jako přepínač mezi připojením k zemi nebo odpojením) s možností pull-up nebo pull-down a další. Všechny tyto módy určují, jak se bude pin chovat [27].

3.3.3 Obecný časovač

Každý z rodiny mikrokontrolerů STM32F4 obsahuje *general purpose timer* v dokumentaci označené jako TIM2 až TIM5. Skládají ze z 16 bitového nebo 32 bitového čítače, který řídí programovatelná dělička (*prescaler*). Dělička je 16 bitová a dovoluje dělit frekvenci časovače libovolným faktorem mezi 1 až 65536. Časovač lze použít například pro měření délky impulsu signálu nebo generování výstupních signálů (PWM[3]), jak je popsáno v dokumentaci [27]. Vše lze nastavit přímo pomocí registrů nebo lze využít HAL (*Hardware Abstraction Layer*), který abstrahuje ovládání časovače do funkcí.

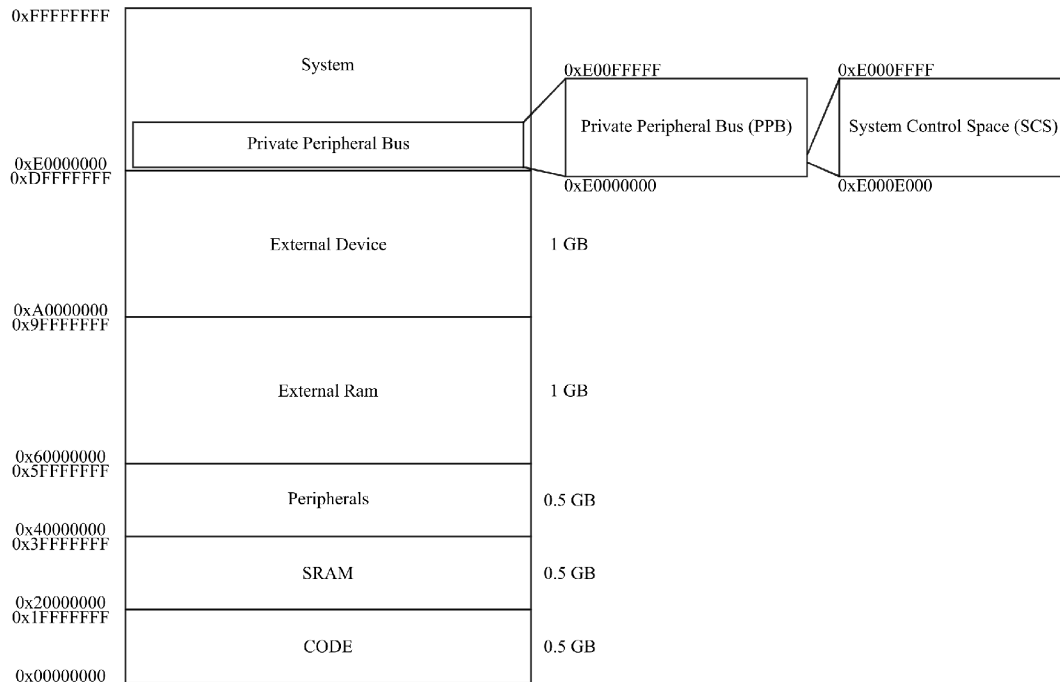
3.4 Paměť

Mikrokontrolery často obsahují alespoň dva typy pamětí. Flash paměť pro kód programu a statickou RAM (SRAM) pro data. Někdy je také přítomna *Electrical Erasable Read Only Memory* (EEPROM). Pro programátora jsou jejich rozhraní typicky nerozlišitelná a stačí tak vědět velikost a adresu. Může existovat více rozhraní sběrnice k různým částem paměti.

Procesory Cortex-M mají rozdělený adresovací prostor do několika regionů viz 3.4. Nicméně celý systém je flexibilní, takže lze například kód i data číst z CODE nebo SRAM bloků. Tato paměť není součástí procesoru, ale každé zařízení s mikrokontrolerem má vlastní velikosti pamětí. Zařízení tak používají různé části, typicky menší, daných regionů a některé mohou být i nevyužité. Konzistence mapy pamětí umožňuje snazší převedení kódu mezi Cortex-M zařízeními a znovu použité softwaru. Jako většina procesorů i Cortex-M

[3] *Pulse Width Modulation* – Pulzně šířková modulace slouží k přenosu analogového signálů pomocí signálu s pouze dvěma hodnotami.

používají zásobník, například pro uložení kontextu při volání funkcí. Zásobník se nachází v Systémové části paměti.



Obrázek 3.4: Mapa paměti rodiny procesorů Cortex-M

3.5 NVIC

NVIC (*Nested Vector Interrupt Controller*) je ovladač pro obsloužení přerušení. V ARM terminologii je přerušení speciální druh výjimky. Výjimky jsou události způsobující změnu ve vykonávání programu. Pokud nějaká nastane procesor pozastaví vykonávání programu a vykoná část programu na její obsloužení. Poté je obnoven průběh původního programu. Číslování od 1 do 15 je pro systémové výjimky, od 16 a výše, typicky do 100, se jedná o přerušení. NVIC se nalézá v *System Control Space* části paměti, společně s MPU (*Memory Protection Unit*). Mezi jeho hlavní funkce patří: nastavení výjimek i přerušení, určení priorit mezi nimi a jejich maskování. Pro určení specifické rutiny na obsloužení přerušení je použita vektorová tabulka. Jedná se o pole v paměti, které spojuje druh výjimky a adresu obslužného kódu.

3.6 Pohyblivá řádová čárka

Čísla s pohyblivou řádovou čárkou umožňují procesoru pracovat s mnohem větším rozsahem v porovnání s celými čísly. Cortex-M4 procesor má možnost podpory jednotky pro *single-precision* (tedy šířka báze 32 bitů) čísla pohyblivé řádové čárky. Tím je možné akcelerovat jejich operace. Vyšší přesnost musí být řešena knihovnami a je tak mnohem pomalejší. Tato jednotka FPU (*Floting Point Unit*) je použitelná jen v případě, že výpočet bude pouze jedna nebo několik instrukcí, při složitých funkcích jako sinf() nebo cosf() není možné

FPU použít. FPU je vnímáno jako koprocesor[4]. Speciální registr CPACR umožňuje svým nastavení aktivovat/deaktivovat FPU. Při použití je nutné dávat pozor na použité funkce, které často pracují s *double-precision* (šířka báze 64 bitů), které podporuje pouze Cortex-M7 a značně zpomaluje výpočet u nižších verzí. FPU poskytuje 32 *single precision* registrů pro operace s pohyblivou řádovou čárkou pro Cortex-M4.

3.7 Zpracování digitálních signálů

Procesory Cortex-M4 a Cortex-M7 obsahují rozšíření pro zpracování digitálních signálů – DSP (*Digital Signal Processing*). Tyto instrukce urychlují práci s audiem, videem, naměřenými hodnotami a dalšími druhy dat. Patří sem například SMULL pro násobení čísel se znaménkem, SMLAL pro znaménkové vynásobení a přičtení , SMMLS pro znaménkové vynásobení a odečtení a další...

Je možné využít také SIMD – *Single Instruction Multiple Data* instrukce, pracující se zabalenými 8 nebo 16 bitovými celými čísly v 32 bitovém registru. Jeden 32 bitový registr pojme 2x16-bit nebo 4x8-bit hodnot. V kódu C je první nutné nahrát data do proměnné typu *int32_t* a poté použít odpovídající SIMD instrukci.

3.7.1 Optimalizace v DPS

DSP lze urychlit několik způsoby, základním je nastavit optimalizační mód v překladači. Výhodné je ale i například seskupit načítání a ukládání dat, protože po sobě následují práce s pamětí jsou rychlejší. Během jakékoli optimalizace je dobré kontrolovat kód assembleru, zda vše probíhá správně a jsou použity korektní instrukce.

Další možnost optimalizace je například rozbalování smyček. Cortex-M4 má pro každou iteraci smyčky režii 3 cykly. Rozbalení smyčky N násobně tak redukuje režii na $3/N$ pro iteraci. Rozbalení lze provést ručně, opakováním konkrétních příkazu, nebo je možné aby to za nás provedl kompilátor. Tuto funkci podporuje například Keil MDK kompilátor, který lze navést použitím speciálního příkazu *pragma*. Příkaz *pragma unroll(N)* by měl vést k N krát rozbalené smyčce. Je vhodné ujistit se, že vygenerovaná kód je efektivní. Klíčové jsou registry a jejich použití, pokud rozbalováním přesáhneme jejich kapacitu začnou se mezivýsledky ukládat na zásobník, čímž značně snížíme celkovou efektivitu.

Urychlení lze také dosáhnout používáním *inline* funkcí, omezením přesnosti nebo použitím 32 bitových instrukcí.

3.8 Programování vestavěných zařízení

Protože se všechny vestavěné zařízení liší je očividné, že se bude většinou lišit i jejich specifikace a zapojení. Při programování těchto zařízení je tak nezbytné mít k dispozici dokumentaci a další zdroje od výrobce. Typicky je možné najít manuál pro čip mikrokontroleru, který obsahuje programovací model periferních zařízení, mapu paměti a další informace potřebné pro vývoj softwaru. Dále je také nutný *datasheet* daného mikrokontroleru, jenž mimo jiné obsahuje informace o způsobu zapojení pinů, operační podmínky (například teplotu) a proudové charakteristiky. Je taky možné najít další specifické knihovny nebo firmware, podle výrobce.

[4]Koprocesor je specializovaný procesor určený pro rozšíření funkcí primárního procesoru

3.8.1 Datové typy

Běžné překladače podporují všechny standardní C typy (*int, signed, char, unsigned long long*) a stejně tak i typy integerů s pevnou délkou, jak je definuje standard C99[5]. Tyto datové typy mají za cíl zvýšit přenositelnost programů, protože základní typy integer mají svou velikost závislou na platformě a může se tak lišit, toto je obzvláště důležité pro vestavěné zařízení, jedná se o typy *int8_t, int16_t, uint8_t* a větší. Stejného principu využívá i CMSIS, který dále definuje *float32_t* a *float64_t* a nový typ Q (*q7_t, q15_t, q31_t, q63_t*) [8].

Formát Q jsou notací čísla s pevnou řádovou čárkou, takže jsou uložena jako normální znaménkový integer a stejně tak se na nich provádí i operace běžným hardware a ALU. Formát ale má specifikovaný počet desetinných bitů a případně celočíselných bitů, typ *q7_t* tak odpovídá jednomu celočíselnému bitu a 7 desetinným bitům, číslo tedy vyžaduje 8 bitů a má rozsah podle vzorce: $[-(2^{m-1}), 2^{m-1} - 2^{-n}]$ tedy: $[-1, 0.9921875]$. Q formát je často použit v hardware, který nemá jednotku pro čísla s plovoucí řádovou čárkou. Čísla jsou uložena ve dvojkovém doplňku. Q čísla jsou tedy poměr dvou integerů, čitatel je uložené číslo a jmenovatel se dopočítá jako 2^n. Při provádění matematických operací je nutné hlídat saturaci. Velkou výhodou je že jmenovatel je mocnina dvou, takže násobení může být implementováno jako aritmetický posun doleva a dělení jako posun doprava, na mnoha procesorech je tato operace rychlejší jak klasické násobení nebo dělení.

3.8.2 SDK

SDK označuje *software development kit*, což je sada různých nástrojů pro vývoj aplikací pro specifickou platformu a zařízení na ní založené. Výrobci mikrokontrolerů většinou poskytují hlavičkové soubory a zdrojové kódy s definicemi registrů pro periferní zařízení a funkcemi pro jejich nastavení a použití. Často poskytují i demo aplikace, které mohou také pomoci při vývoji.

Pro programovaní mikrokontrolerů jsou dále potřeba softwarové nástroje pro překlad, linkování a nahrávání do paměti. Existuje více jak 10 různých nástrojových řad pro Cortex-M procesory. Jedná se například o Keil MDK, ARM DS-5, IAR Systems, Atollic TrueStudio, GNU Compiler Collection, Texas Instruments Code Composer Studio a další.

3.8.3 STM32CubeF4

STMCube je obecně snaha firmy STMicroelectronics ulehčit práci vývojářům pomocí redukování potřebného úsilí, času a ceny na vývoj softwaru. STM32CubeF4 je implementací STMCube, která pokrývá portfolio STM32F4. STM32CubeF4 obsahuje STM32CubeMX, což je program umožňující pomocí grafických průvodců generovat inicializační C kód. Dále také obsahuje STM32CubeF4 MCU balíček, jenž se skládá z HAL (*hardware abstraction layer* – vrstva pro abstrakci hardware) a více LL (*Low-Layer* - nízko úrovňové) API, HAL i LL jsou dostupné pod BSD licencí pro otevřený software. Součástí balíčku je také skupina *middleware* komponent jako RTOS[6], USB, FAT souborový systém, grafiku a TCP/IP. Všechny software utility pro vestavěná zařízení jsou také použita v příkladech a demech běžících na mikrokontrolerech od STMicroelectronics.

[5] C99 je neformální jméno pro ISO/IEC 9899:1999 verzi jazyka C. Tuto verzi překonala C11 v roce 2011 [16].

[6] *Real-Time Operating System* – Operační systém pro aplikace běžící v reálném čase, které zpracovávají data, typicky okamžitě bez zdržení způsobené vyrovnávacími paměti [24].

3.8.4 HAL (*Hardware Abstraction Layer*)

Vrstva ovladačů HAL poskytuje několik obecných a jednoduchých rozhraní se kterými můžou vyšší vrstvy interagovat. Je postavená s ohledem na obecnou architekturu, takže umožňuje vrstvám postavených na ní, jako je middleware, implementovat jejich funkce, aniž by museli podrobně znát jak se MCU používá. Tím je zajištěna lepší znovu použitelnost a jednoduchá přenositelnost knihoven na další zařízení. Například periferní zařízení pro komunikaci obsahují API pro inicializaci a nastavení periferního zařízení, správu přenosu dat dle dotazování, zpracování přerušení nebo DMA a řízení chyby v komunikaci. HAL se skládá z generických a rozšiřujících API. Generické API poskytují běžné a obecné funkce pro celou STM32 sérii. Rozšiřující API se zaměřují na konkrétní funkce pro danou rodinu nebo součástku.

Ovladače vrstvy HAL také implementují detekci chyb za běhu skrze kontrolu hodnot všech vstupů do funkcí. Toto dynamické hlídání přispívá k robustnosti firmware. Detekce za běhu je také vhodná pro uživatelský vývoj aplikací a hledání chyb [28].

3.8.5 LL (*Low-Layer*)

Ovladače LL nabízí hardware služby na základě dostupných schopností periferních zařízení STM32. Tyto služby přesně odpovídají schopnosti daného hardware a poskytují atomické operace, které musí být vykonány podle programovacího modelu, jenž je popsán v referenčním manuálu zařízení. Všechny operace jsou provedeny pomocí změny hodnot v registrech periferních zařízení, nepotřebují tak žádné další paměti, čítače nebo jiné zdroje. Na rozdíl od HAL jsou LL API dostupné jen pro periferie, pro které je optimalizovaný přístup klíčovou funkcí. LL jsou tak doplňkem pro HAL, protože poskytují nízko úrovňové API v rámci registrů, lepší optimalizaci, ale méně přenositelnosti. Vyžadují pokročilou znalost MCU, periferních zařízení a jejich specifikací.

3.8.6 CMSIS

CMSIS je důležitý standard pro softwarová rozhraní mikrokontrolerů, vyvinutý společností Arm. Standard je podporován spoustou produktů napříč rozsáhlým Cortex-M ekosystémem. Umožňuje tak kratší vývoj skrze znovu použití kódu a současně redukuje šanci výskytu chyb. Jádro standartu tvoří CMSIS-Core, historicky původní část obsahuje množinu rozhraní pro aplikace nebo middleware[7] jenž potřebují přístup k blokům procesoru, bez ohledu na konkrétní použitý mikrokontroler. CMSIS-Core tak standardizuje:

- Definice pro periferie procesoru. To znamená registry pro *NVIC*, hodiny procesoru (*SysTick*), jednotku pro ochranu paměti (*MPU*), různé registry pro ovládání systému a část rozhraní pro debuggování.
- Funkce pro přístup k vlastnostem procesoru jako řízení přerušení pomocí *NVIC* a funkce pro přístup ke speciálním registrům. Stále je možné k registrům přistupovat přímo, ale použití API[8] může pomoci přenositelnosti software.
- Standardní funkce pro použití speciálních instrukcí. Tyto instrukce nelze generovat pomocí ISO C, proto CMSIS poskytuje sadu funkcí pro jejích použití. Jedná se například o instrukce pro čekání na přerušení nebo přepnutí zařízení do režimu spánku.

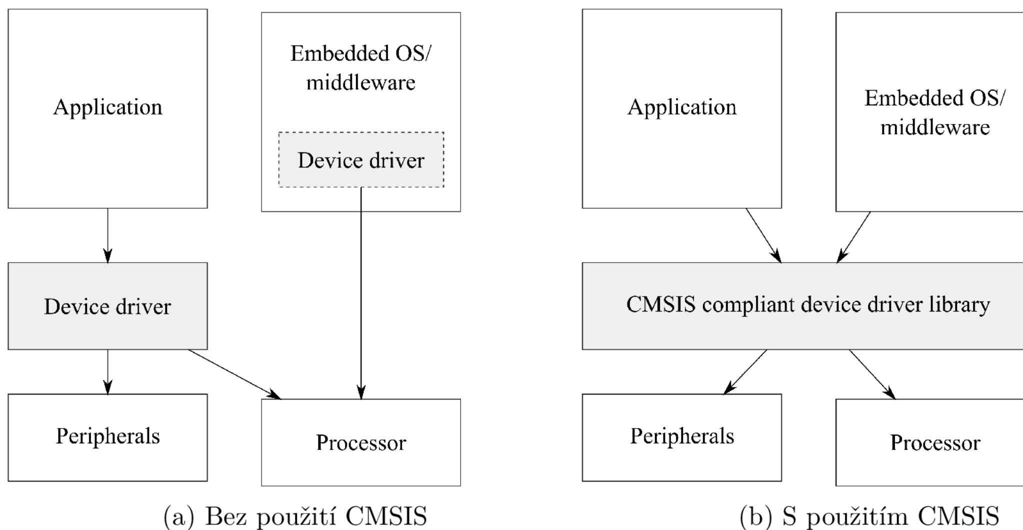
[7] Middleware je software který poskytuje další služby pro aplikace mimo ty dostupné od operačního systému

[8] Zkratka *Application Programming Interface* označuje rozhraní pro programování aplikací.

- Standardní jména funkcí pro zpracování výjimek.
- Definice funkce pro inicializaci systému. Funkce nastavují periodu hodin a registry pro řízení spotřeby před spuštěním aplikace. Pro CMSIS se jedná o *SystemInit()* funkci, její implementace je samozřejmě specifická pro každé zařízení, ale její použití, umístění a jméno jsou vždy totožné.
- Jméno softwarové proměnné pro hodnotu rychlosti hodin. *SystemCoreClock* nebo *SystemFreq*, ve starších verzích CMSIS, jsou přístupné pro aplikaci.

Mezi další části CMSIS patří CMSIS-DSP, knihovna pro zpracování signálů, podporující mimo jiné: matematické funkce, filtry, funkce pro práci s maticemi, transformace a statistické a interpolační funkce. Tyto operace můžou být velmi užitečné pro efektivní implementaci konvolučních sítí. Knihovna má zvlášť rozdělené funkce pro 8 bitové hodnoty, 16 bitové hodnoty, 32 bitové hodnoty a 32 bitové hodnoty s plovoucí řádovou čárkou. Knihovna je také optimalizovaná pro využití DSP rozšíření, pokud na přítomné.

Standard CMSIS ještě obsahuje CMSIS-SVD, CMSIS-RTOS a CMSIS-DAP pro další různé funkce. CMSIS předchází potřebě ovladačů uvnitř operačního systému nebo middleware a jejich použití, aplikace může používat periferní zařízení přímo, obrázek 3.5.



(a) Bez použití CMSIS (b) S použitím CMSIS

Obrázek 3.5: Rozhraní procesoru a periferií

3.9 Neuronové sítě

Neuronové sítě můžeme zefektivnit pomocí hardwarové akcelerace, existují architektury jako *Origami* [9], které se zaměřují na maximální znovu použití načtených dat, lokalitu a co nejefektivnější výpočet, nebo *YodaNN* pracující s binárními váhami. Pro *YodaNN* jednotlivé váhy nabývají hodnot 0 nebo 1, což značně urychlí a zjednoduší celý výpočet, protože je možné oprostit se od složitých a drahých násobiček [7].

Tento přístup staví na obecnějším *BinaryConnect*, kde je ukázáno, že neuronová síť dokáže obecně pracovat s binárními váhami. Pokud při dopředném výpočtu (použití sítě) a při zpětné části výpočtu (během učení) v *backpropagation* použijeme binární váhy, s tím že samotná úprava hodnot podle chyby se počítá z původních přesných vah, dosáhne síť

porovnatelných výsledků se sítí s plnou přesností. Toto je možné, protože omezení na binární váhy je v podstatě regularizace, tedy zabraňuje přetrénování [11].

Kapitola 4

Návrh a implementace

Tato kapitola se zabývá hlavní částí zdrojových kódů práce. Práce je rozdělena na dvě hlavní části, učení v PC a klasifikace v MCU. Každé části se věnuje samostatná sekce, ve které jsou rozebrány její implementační detaily a problémy. Podle kapitoly zabývající se Cortex-M procesory je očividné, že první vhodný procesor pro efektivní implementaci hluboké konvoluční sítě bude verze procesoru Cortex-M4. Cortex-M4 mají k dispozici FPU a podporují DSP. Zvolené zařízení pro část dopředného vyhodnocení, klasifikace v MCU, je STM32F429 Discovery, to má procesor STM32F428ZIT6 řady Cortex-M4.

4.1 MNIST databáze

Pro práci bylo nutné zvolit vhodnou úlohu pro demonstraci využití konvolučních neuronových sítí. Zvolil jsem klasickou úlohu rozpoznávání číslic, kterou začíná většina příkladů pro tyto sítě. Tento problém je populární díky známému datasetu MNIST[1], který se k němu neodmyslitelně váže. Rozhodl jsem se držet této konvence a také celou práci implementovat kolem klasifikace ručně psaných číslic. Databáze obsahuje 60000 trénovacích příkladů a 10000 testovacích příkladů, toto rozdělení je oficiální MNIST popis, není ale nutné jej dodržet a je možné si příklady přerozdělit podle potřeby.

Každý příklad je černobílý (pouze různé stupně šedí) obrázek s rozměry 28x28 pixelů, kde samotná číslice má normalizovanou velikost a je v poli s pevným počtem pixelů vycentrovaná, k obrázku patří také označení, jaké číslo představuje. Databáze MNIST je vhodná pro studium strojového učení, klasifikace a rozpoznání obrazu dat z reálného světa, protože minimalizuje potřebu předzpracovaní a formátování. Tento problém má potenciál i pro další použití čipu s kamerou. Ukázku číslic z databáze lze vidět v tabulce 4.1.

4.2 Vymezení problému

Celou práci lze konceptuálně rozdělit do dvou hlavních částí, první je vytvoření a naučení sítě na x86 architektuře, do druhé pak patří dopředné vyhodnocování sítě a její optimalizace ve vestavěném zařízení s ARM procesorem. Jako menší mezi krok pak lze vnímat přenesení sítě mezi platformami. Rozdělení je ilustrováno na obrázku 4.1.

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

Tabulka 4.1: Ukázka MNIST číslic

Přehled první části:

V první části bude ze začátku nutné zvolit síť a určit, jak ji implementovat, v jakém jazyce a zda, případně jaký, použít framework. Dále provést učení sítě, což je nejkomplikovanější a výpočetně nejnáročnější část práce s neuronovými sítěmi a posléze vyexportovat hodnoty vah a biasů, ke kterým učení dospělo. Právě díky výpočetní náročnosti je jednoznačně lepší řešení síť trénovat na výkonnějším počítači například s dnes nejrozšířenější architekturou x86. x86 je označení pro instrukční sady se kterými pracují procesory navazující na 16 bitový procesor Intel 8086 [5]. Z historických důvodů je toto označení dnes používáno jak pro 32 bitové počítače tak i, méně častěji, pro 64 bitové počítače, které představují všudy přítomné klasické stolní počítače, servery, notebooky i superpočítače. Jsou také nejsnáze dostupné ve výkonnějších konfiguracích, které mnohonásobně předčí vestavěné systémy a jsou tak vhodnější pro trénování sítě.

Přehled druhé části:

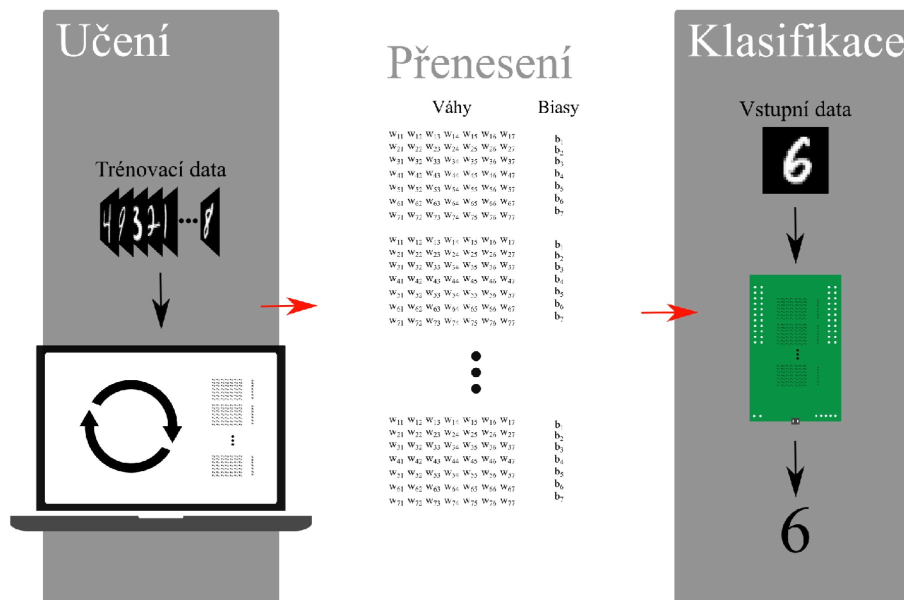
Druhá část spočívá v naprogramování dopředného vyhodnocení sítě ve vestavěném zařízení, nahrání a uložení všech potřebných dat jako jsou vstupní obrázky číslic MNIST i váhy a biasy sítě do paměti mikrokontroleru, otestování, zda mikrokontroler správně klasifikuje čísla, změření rychlosti vykonání dopředného průběhu sítě, optimalizace implementace sítě, a nakonec změření dosaženého zrychlení nebo energetické úspory.

4.3 Učení v PC

Základ práce tedy tvoří konvoluční neuronová síť, která úspěšně klasifikuje čísla databáze MNIST. Tato sekce představuje první část práce.

4.3.1 Knihovny a frameworky pro sítě

Pro tvorbu sítí existuje nespočet knihoven, frameworků a obecně návodů. Bylo by také možné síť implementovat úplně od začátku, ale byť by tato možnost přinesla největší porozumění fungování programu, jednalo by se o spoustu práce, kde by zejména část učení značně zpomalila další postup. Hledáme proto knihovnu, která umožní abstrahovat především učení sítě, počítání gradientů v algoritmu backpropagation[19], ale zároveň bude srozumitelná a transparentní tak, aby parametry sítě byly snadno přístupné a celá dopředná



Obrázek 4.1: Abstrakce práce do částí

část výpočtu šla replikovat ve vestavěném zařízení. Následuje výčet možných knihoven a frameworků:

Knihovna Theano

Theano není knihovna specializovaná na neuronové sítě, ale na definování, optimalizaci a vyhodnocení matematických výrazů, efektivně pracujícími s více rozměrnými poli [29] v Pythonu. Theano využívá populární knihovnu NumPy (přidávající podporu pro práci s vícerozměrnými poli a maticemi i velkou sbírku funkcí pracujícími s nimi), ale jejím hlavním kladem je efektivní derivace symbolů, tedy derivace pro funkce s jednou nebo více vstupy. Tato vlastnost je velmi žádaná při učení sítí.

TensorFlow

TensorFlow je *open source* knihovna pro výkonné numerické výpočty za pomoci grafů datových toků [6]. Uzly grafu reprezentují matematické operace a hrany představují vícerozměrné pole dat (tensory) pohybující se mezi uzly. Takto flexibilní architektura umožňuje nasadit kód na stolních počítačích na CPU i GPU, serverech nebo mobilních zařízení bez přepisování zdrojových souborů. Původně vyvinutá týmem *Google Brain* z *Google AI* organizace. Knihovna má dobrou podporu pro strojové učení a její flexibilní numerické jádro je používáno v mnoha vědních doménách.

CNTK

Microsoft Cognitive Toolkit je *open source* toolkit pro komerční distribuované strojové učení. Specializuje se tedy na neuronové sítě, které popisuje posloupností výpočetních kroků pomocí orientovaného grafu. V tomto grafu listy reprezentují vstupy nebo parametry sítě, zatímco ostatní uzly představují maticové operace nad jejich vstupy. CNTK umožňuje jednoduše vytvářet a kombinovat modely pro dopředné, konvoluční a rekurentní sítě. Pro učení

je využit stochastický gradientní sestup (SGD), popsaný v kapitole 2.5.3, a algoritmus *backpropagation*, popsán: 2.5.4, s automatickou diferenciací a paralelizací na více jader GPU a serverů [20].

Keras

Keras je API nadstavba nad TensorFlow, Theano nebo CNTK pro neuronové sítě napsaná v jazyce Python. Jejím hlavním cílem je umožnit rychlé experimentování, a proto typování se sítěmi.

Tiny-dnn

Tiny-dnn je C++ 14 implementace knihovny pro strojového učení přizpůsobená pro limitované výpočetní zdroje, jako jsou vestavěné zařízení a IoT zařízení. Knihovna tedy pro svůj běh nepotřebuje žádné jiné programy ani další knihovny, navíc ji není nutné nijak instalovat, stačí přeložit zdrojové kódy a poté includovat hlavičkové soubory. Knihovna navíc podporuje použití před trénovaných modelů pomocí *caffe-convertoru* [22].

Caffe

Convolutional Architecture for Fast Feature Embedding je framework pro strojové učení specializovaný na hluboké sítě. Napsaný v C++ s rozhraním v jazyce Python. Caffe je zaměřené na klasifikaci a segmentaci obrazu. Framework také vyžaduje řadu dalšího software, jako Boost, BLAS, hdf5 nebo glog pro své fungovaní, tedy jeho instalace je komplikovaná [17]. Získává tak ale na robustnosti a rychlosti [15]. Caffe tedy není vhodné na experimentování a testování, nýbrž se hodí pro produkční prostředí.

Torch

Další framework pro strojové učení je Torch, podporuje řadu algoritmů pro učení a upřednostňuje GPU funkcionalitu. Pracuje se skriptovacím jazykem LuaJIT, který slouží jako rozhraní pro C/CUDA[2].

Lasagne

Stejně jako Keras i Lasagne je nadstavba nad Theanem. Jedná se o knihovnu pro tvoření a trénovaní neuronových sítí v jazyce Python. Podporuje dopředné sítě, různé optimalizační metody a volně definovatelnou cenovou funkci [12].

Mezi další frameworky, knihovny a toolkity patří například Apache MXNet, DSSTNE, Eclipse Deeplearning4j a řada dalších. Většina jich je však specializovaná na efektivní rozšíření na více GPU a více počítačů, nejsou tak dobrá řešení pro vestavěné zařízení nebo pro experimentování a učení sítě.

Požadavky práce nejlépe splňují Theano a TensorFlow, obě knihovny jsou dostatečně nízko úrovňové, aby šlo odvodit, jak provádí dopřednou část výpočtu a současně jsou schopné vyřešit učení sítě, bez větších komplikací. Protože obě knihovny jsou pro obecné matematické výpočty, samostatně neposkytují žádnou pomoc s neuronovými sítěmi. Nicméně kniha *Neural Networks and Deep Learning* [21] obsahuje kromě přehledného vysvětlení

[2]CUDA je platforma pro paralelní programovaní a API model vytvořený společností Nvidia.

konvolučních a neuronových sítí obecně, také implementaci sítí s knihovnou Theano. Tyto zdrojové kódy pro klasické i konvoluční sítě jsou velmi detailně popsány a vysvětleny, což usnadní re-implementaci ve vestavěném zařízení.

4.3.2 Architektura sítě

Implementace neuronové sítě s knihovnou Theano z *Neural Networks and Deep Learning* je pouze necelých 500 řádků v Pythonu, které tvoří obecný framework pro neuronovou (konvoluční) síť. Hlavní třída *Network* se kromě inicializace skládá téměř výhradně z SGD metody, která pomocí dvou vnořených *for* cyklů přes definovaný počet epoch a počet *mini-batch* v epoše trénuje síť. Síť dále potřebuje jednotlivé vrstvy, ty jsou dodány při inicializaci a každá vrstva je reprezentována vlastním objektem. Tento model je výhodný svou modularitou a rozšiřitelností. Existuje tak *FullyConnectedLayer* představující plně propojenou vrstvu neuronů, *ConvPoolLayer* pro konvoluci a pooling dat společně a *SoftmaxLayer* jenž aplikuje na vstupy funkci soft-max, čímž získáme rozložení pravděpodobnosti mezi neurony.

Každá definovaná vrstva má určený počet vstupních a výstupních neuronů, aktivační funkci a případně míru "odstřelených"(jedná se o *dropout*, technika regularizace) neuronů. Tyto parametry se typicky zadávají dynamicky při tvorbě objektu vrstvy. Při tvorbě objektu vrstvy se také vytvoří a inicializují Theano sdílené proměnné pro váhy a biasy. Každá vrstva pak má metodu jménem *set_inpt*, která je zařazena do matematického vyhodnocení sítě, protože popisuje, jakým způsobem pro danou vrstvu přepočítat vstup na výstup. Tedy v případě plně propojené vrstvy proběhne jen vektorový součin, pro konvoluční vrstvu je zde konvoluce a pooling a tak dále. Stejným způsobem by šlo framework rozšířit o další vrstvy, definované dle potřeby.

Dále je nutné určit hyper-parametry konkrétní sítě, to znamená například po kolik epoch se bude síť učit, najít vhodnou architekturu sítě nebo inicializační hodnoty vah. Najít tyto parametry může být problém sám o sobě, naštěstí tato práce se nezaměřuje na maximální úspěšnost klasifikace MNIST čísel. Nielsen například udává nalezenou přesnost 99.43 % pro následují nastavení: celkem 5 vrstev, první dvě konvoluční, poté dvě plně propojené a nakonec soft-max vrstva.

1. konvoluční vrstva:

Vstup je klasický jeden MNIST obrázek 28x28 pixelů, konvoluce je s jádrem 5x5 a krokem 1, je proveden max pooling s maticí 2x2, a nakonec je použita ReLU aktivační funkce. Výstup je 20 *feature map* o rozměru 12x12, tedy vrstva má 20 konvolučních jader.

2. konvoluční vrstva:

Má na vstupu 20 12x12 *feature map* z předchozí vrstvy, konvoluce je s jádry 5x5 a krokem 1, je proveden max pooling s maticí 2x2, a nakonec je použita ReLU aktivační funkce. Výstup je 40 *feature map* o rozměru 4x4, každý jeden ze 40 výstupů je propojen s každým s 20 vstupů, tedy vrstva má 800 konvolučních jader (20 pro každý výstup).

3. plně propojená vrstva:

Největší vrstvou, co se počtu vah týče je 3 vrstva. Jedná se o vrstvu plně propojující každý vstupní pixel $40 * 4 * 4$ s každým výstupním neuronem. Vrstva má 100 výstupních neuronů a aktivační funkci ReLU.

4. plně propojená vrstva:

Nejjednodušší vrstvou je 4 vrstva. Vrstva plně propojuje 100 vstupních a 100 výstupních neuronů. Aktivační funkcí je opět ReLU.

5. soft-max vrstva:

Přijímá 100 vstupních neuronů, stejným způsobem jako plně propojená vrstva je přepočítá na 10 výstupů a na ty aplikuje soft-max funkci, čímž získáme výsledné rozložení pravděpodobností. Tato pravděpodobnost nám říká, jaké číslo, s jakou šancí je podle sítě na obrázku.

Kniha *Neural Networks and Deep Learning* dále doporučuje velikost *mini-batch* 10 a 60 trénovacích epoch. SGD dále pracuje s parametrem udávajícím míru učení η, ten je nastaven na 0.1.

4.3.3 Učení sítě

Jak již bylo zmíněno celé učení sítě probíhá v podstatě ve dvou *for* cyklech. Jednoduše iterujeme přes epochy a trénujeme síť na datech z patřičných mini-batch. Theano za nás udělá většinu práce skrze definované symbolické výrazy pro cenovou funkci, výpočet odpovídající derivace pro gradient i aktualizaci parametrů sítě. Toto je hlavní výhoda a funkce knihovny Theano, umožní nám abstrahovat složité výpočty pro učení sítě.

4.3.4 Export vah a biasů z vrstev naučené sítě

Objekt třídy *Network* má uložené všechny vrstvy v proměnné *layers* přes které se lze dostat až k jednotlivým váhám a biasům. Vytvořil jsem proto skript v Pythonu, který se spustí po dokončení učení a exportuje hodnoty do souboru. Výhodou skriptovacího jazyka Python je jednoduché formátování textu, protože ale pracujeme s obrovským množstvím hodnot s plovoucí řádovou čárkou je nutné nastavit Numpy, aby nijak neomezoval a nesumarizovalo výpis, toto zajistí metoda *set_printoptions* volaná přímo nad importovaným *numpy* objektem. Poté už lze přímočaře pomocí *replace* metody zformátovat data přímo do podoby C hlavičkového souboru, aby jej bylo možné, bez dalších úprav, použít přímo ve zdrojových souborech mikrokontroleru.

Během exportu ale kromě formátování a přidání syntaktických náležitostí jazyka C, je nutné pro každou vrstvu matice vah transponovat. Theano má totiž pro efektivnější výpočet hodnoty uloženy jako 1D pole a během transformace metodou *reshape* v konvoluční vrstvě, dojde k jejich transponování. Tedy síť je naučená klasifikovat pro transponované váhy. Skript provádí ještě zarovnání nulami, které je vysvětleno v kapitole 5.3.

Možné další rozšíření dynamického charakteru konvertování sítí z Pythonu do vestavěného zařízení by šlo docílit vytvořením struktury (pole) v hlavičkovém souboru s informacemi o architektuře sítě, podle nějž by pak byla dynamicky sestavena síť právě v mikrokontroleru.

4.4 Klasifikace v MCU

Celá druhá část je zaměřená na mikrokontroler, konkrétně je použito zařízení STM32F4 Discovery. Protože vestavěné zařízení mají obecné k dispozici méně zdrojů než klasické stolní x86 počítače, provádí se zde pouze dopředné vyhodnocení sítě. V této sekci je blíže popsáno

fungování konvoluční sítě právě ve vestavěném zařízení, bez zaměření na optimalizaci rychlosti výpočtu. Pro tuto základní implementaci nebyl použit žádný framework, cílem bylo vytvořit co nejjednodušší implementaci dopředného vyhodnocení konvoluční neuronové sítě. Dále s touto implementací změřit dobu vykonání a použít ji jako výchozí referenční bod pro optimalizovanou verzi, popsanou v další kapitole.

4.4.1 Programování

Pro překlad zdrojových kódů pro STM32F429 je použit překladač arm-none-eabi-gcc, jenž patří do *GNU Compiler Collection*. GCC je populární sada nástrojů pro generování spustitelných souborů na různých architekturách. Podle volné konvence lze jméno překladače interpretovat jako architekturu, zde ARM, prodejce (není žádný) a dále *eabi* značí *Embedded Application Binary Inteface* tedy že se bude generovat assembler kód pro vestavěné zařízení [25]. Vzniklý *elf* soubor je možné pomocí arm-none-eabi-objcopy přeložit do binární nebo hexadecimální formy. Programy v *elf* formátu jsou užitečné protože je možné je použít při debuggování programem arm-none-eabi-gdb. Binární program je ale zase možné nahrát přímo do flash paměti zařízení, které jej začne vykonávat. Pro naprogramování zařízení je použita *open source* verze Stlink Tools pojmenovaná STLINK[3]. STLINK je software zaměřený na programování a hledání chyb v programech pro desky STM32 Discovery. Ty obsahují vestavěný čip ST-LINK/V2 jenž překládá USB příkazy odeslané z hostujícího PC do JTAG/SWD příkazů, blíže popsáno v kapitole 3.3.1. STLINK závisí na *libusb-1.0* a *cmake*, pakliže jsou závislosti splněny je možné zdrojové soubory přeložit.

Software obsahuje:

- komunikační knihovnu *libstlink.a*,
- GDB server *st-util*,
- nástroj pro manipulaci flash paměti *st-flash*,
- programátor a nástroj pro získání informací o čipu *st-info*.

Pakliže chceme interagovat se zařízením je nutné spustit GDB server (*st-util*) poté je možné použít arm-none-eabi-gdb k práci se zařízením. Z GDB se lze připojit na server pomocí: *target extended localhost:4242*, GDB pak má mapu paměti čipu a je schopné nahrát projekt do flash nebo SRAM paměti, podle toho jak byl projekt sestaven. Stačí specifikovat *.elf* soubor a příkazem *load* data nahrát do zařízení.

Pro jednodušší nahrání nebo čtení binárních dat z různých částí paměti slouží *st-flash*. *St-flash* přijímá jako parametry, zda chceme číst/zapisovat (*read*/*write*), cílový/zdrojový soubor a adresu paměti. Je také možné použít *.hex* soubor pomocí parametru *--format ihex*.

4.4.2 Uložení vah a biasů sítě

Každá vrstva má dvě 1D pole, jedno pro váhy a jedno pro biasy. Pole s váhami je typicky mnohokrát větší, jednoduše proto že každý neuron má jen jeden bias, ale může být propojen i se stovkami dalších neuronů. Ani váhy ani biasy se během dopředného vyhodnocení sítě nemůžou měnit, je možné je tedy uložit jako konstanty do hlavičkového souboru. Důležitou otázkou také je do které paměti data uložit, jedná se totiž o značné množství hodnot, pro

[3]https://github.com/texane/stlink/

síť specifikovanou v 4.3.2 mají soubory velikost 1-2 MB, podle toho zda používáme data pro optimalizovaný běh. Je zjevné, že do SRAM paměti se data nevejdou, zbývají tedy paměti FLASH a externí SDRAM. SDRAM je pomalejší proti paměti FLASH, bylo by tedy výhodné, aby se celý kód, včetně programu, vlezl do FLASH paměti. Protože SRAM je paměť nejrychlejší kopíruje linker skript při startu všechna data tam, výjimkou jsou data konstantní, tedy u kterých je použito klíčové slovo *const*. Ty jsou ponechána linker skriptem při nahrávání dat do vestavěného zařízení ve větší paměti FLASH.

4.4.3 Uložení dat MNIST

Jeden obrázek čísla z databáze MNIST je reprezentován 784 (28x28) hodnotami s pohyblivou řádovou čárkou, představující úroveň šedi na daném bodě. Nejedná se tedy o velké množství dat, ale pokud bychom v rámci testování chtěli vyzkoušet síť na stovkách obrázků, může i zde paměťová náročnost narůst.

Zajímavou možností je také dynamické nahrávání dat na specifické místo v paměti. *St-flash* umožňuje zapsat do FLASH paměti na specifické místo, je tak možné nahrát nové číslo do paměti, aniž by se musely znova nahrát všechny binární soubory zdrojových souborů, a především dat vah a biasů. V programu je pak napevno napsaná adresa, ze které se načítá klasifikované číslo, to má vždy stejnou velikost, takže stačí mikrokontroleru říct, že od této adresy začíná pole *float32_t* s 784 položkami. Je ale potřeba dbát na reprezentaci čísel, pro zkonvertování čísel z textové reprezentace, které rozumí lidé, do binární reprezentace, je použit program BinMake [4].

4.4.4 Aktivační funkce

Implementace aktivačních funkcí v jejich nejjednodušší podobě.

Sigmoid

Funkce Sigmoid má předpis podle rovnice 2.2 a její průběh je zobrazen na obrázku 4.2. Výpočet funkce se kromě jednoho dělení skládá především z výpočtu exponenciální funkce o základu e. Nejjednodušší způsob výpočtu exponenciální funkce je pomocí Taylorovy řady [14], výpočet je ale náročný, protože pro větší hodnoty vstupu pomalu konverguje. Není tak jasné kdy nekonečný rozvoj ukončit, pro menší hodnoty by stačilo sečíst méně členů, ale pro větší by mohly vzniknou velké nepřesnosti. Samozřejmě čím více členů řady sečteme tím přesnější, ale pomalejší výpočet bude. Tato funkce má velký potenciál pro optimalizaci.

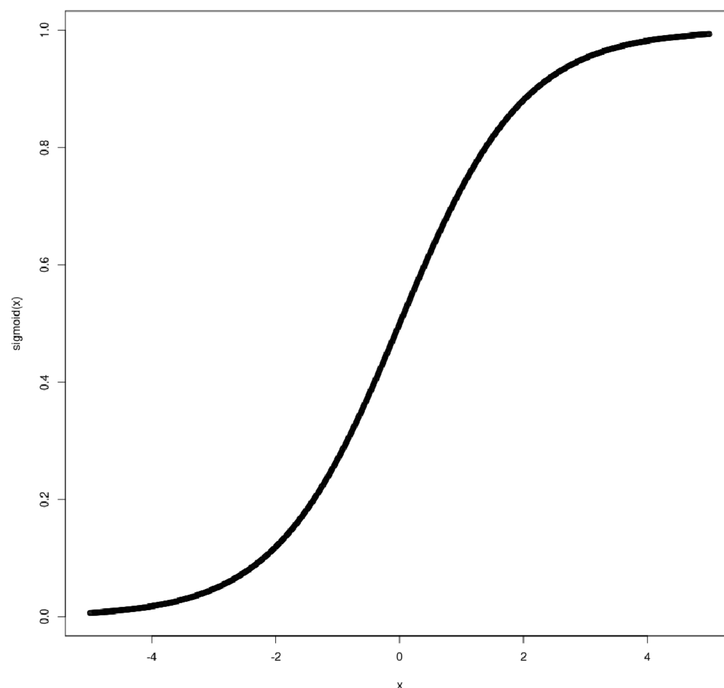
ReLU

Nejjednodušší aktivační funkcí je ReLU, jedná se doslova o jednu podmínku. Pokud je vstupní číslo menší jak nula, funkce vrátí nulu, jinak je vráceno stejné vstupní číslo. Tato funkce nepůjde optimalizovat.

SoftMax

Vrstva s aktivační funkcí SoftMax produkuje na svém výstupu distribuci pravděpodobností. Tedy výstupní hodnoty musí vždy v součtu dát 1, pokud stoupne hodnota jednoho výstupu, musí klesnout hodnoty ostatních výstupů o celkově stejné množství. Funkci lze

[4] https://github.com/dadadel/binmake



Obrázek 4.2: Průběh Sigmoid funkce

popsat rovnicí 4.1. Její výpočet pro celou vrstvu, pro každý neuron, lze provést efektivně dohromady tak, že sumu spočítáme pouze jednou a poté jen dělíme jednotlivé e^{x_j} členy. Při výpočtu je nutné počítat s velkými hodnotami ve spodní části zlomku, suma umocněných Eulerových čísel, může snadno překročit rozsah 32 bitového typu s plovoucí čárkou. V kontextu neuronových sítí $\sigma(x_j)$ představuje výstupy sítě x_j jsou vstupy sítě.

$$\sigma(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \text{ pro } j = 1, .., K. \tag{4.1}$$

4.4.5 Vrstvy

Celá síť se skládá z vrstev. Následující sekce se věnuje popisu jednotlivých vrstev.

Plně propojená vrstva

Nejjednodušší vrstvou je plně propojená vrstva. Skládá se pouze z jednoho *for* cyklu, přes všechny neurony dané vrstvy, kdy se vynásobí postupně jednotlivé váhy s odpovídajícími aktivačními hodnotami předchozí vrstvy. Tato operace odpovídá skalárnímu součinu, protože výsledné produkty vah a aktivačních hodnot je také potřeba sečíst. Vrstva tedy pro každý svůj neuron provede skalární součin vah a vstupů z předchozí vrstvy, přičte hodnotu svého biasu a na takto získanou hodnotu aplikuje specifikovanou aktivační funkci. Skalární součin je samozřejmě také implementován jako *for*, bez použití žádných knihovních funkcí. Je nutné dát pozor na způsob uložení jak hodnot vah, tak výstupů předchozí vrstvy, v základní implementaci se násobí hodnoty řádků matice vah se sloupcem z matice aktivačních hodnot, oboje jsou ale uložené jako obyčejné pole s jednou dimenzí. Způsob uložení

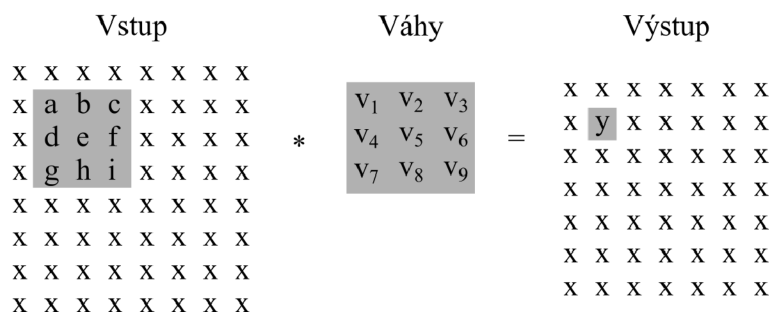
a přistup k hodnotám může značně ovlivnit výkonnost, v optimalizační fázi by bylo možné vyzkoušet jiné postupy a způsoby uložení. Konkrétní implementaci může také ovlivní jaká vrstva přechází plně propojené vrstvě. Pokud se jedná o konvoluční vrstvu je nutné sečíst hodnoty ze všech *feature map* vynásobené odpovídajícími váhami.

SoftMax vrstva

SoftMax vrstva je téměř identická s plně propojenou vrstvou, jen aplikování aktivační funkce SoftMax je nutné provést na všechny neurony zároveň, tedy musí se první spočítat všechny skalární součiny, přičíst biasy a až poté se provede společný výpočet SoftMax.

Konvoluční vrstva

Konvoluční vrstva je v implementaci spojená s pooling vrstvou. První se provede konvoluce poté pooling a nakonec se aplikuje určená aktivační funkce. Nejkomplexnější operací je konvoluce, její hlavním přínosem je respektování obrazové informace, konvoluce je 2D, tedy v rámci obrázku se pracuje s pixely (neurony) pod sebou, nejen vedle sebe. Klasická vrstva seřadí pixely za sebe do řady (pole), čímž přijde o informaci obrázku representovanou seřazením pixelů ve sloupcích, blíže popsáno v kapitole 2.3. Protože ale konvoluce pracuje s čtvercovým výřezem obrázku, tuto informaci zachová, jak je ilustrováno na obrázku 4.3.



Obrázek 4.3: Ilustrace 2D konvoluce

Konvoluce má kromě velikosti okna (čtverce) se kterým pracuje ještě další parametry. Je nutné kopírovat jejich nastavení z Python Theano implementace, která byla použita při učení sítě. Velikost okna je nastavená na 5x5 pixelů, a protože se začíná od okraje vstupního obrázku a neprobíhá žádné doplnění nulami, je výstupní obrázek menší o 2 pixely z každé strany. Posunutí okna neboli krok je nastaven na 1, tedy po výpočtu jedné konvoluce se okno nad vstupní maticí posune o 1. Váhy jsou vždy totožné pro každou pozici okna. Dále také pokud je jako vstup konvoluční vrstvy více obrázků, například různé barevné kanály pro barevný obrázek nebo různé *feature mapy*, je nutné výstupy konvolucí sčítat pro danou pozici pixelu. Takto probíhá konvoluce při učení a je nutné, aby tak probíhala i během dopředné části výpočtu v mikrokontroleru.

Po konvoluci následuje pooling. Pooling funguje ve stejném duchu jako konvoluce, tedy posuvné okénko pevné velikosti, matice vah a operace. Prováděná operace ale není konvoluce, nicméně nějaké jiné mapování na jednu výstupní hodnotu. Implementace v práci používá okénko 2x2 s krokem 1, čímž se velikost výstupních *feature map* zmenší na polovinu. Použitou operací je hledání maxima a váhy nejsou použity, takže jsou nastaveny na samé jedničky.

Posledním akcí konvoluční vrstvy je přičtení hodnoty biasu k dopočítaným výsledkům a použití aktivační funkce.

Kapitola 5

Optimalizace

Práce je zaměřena na implementaci pro vestavěné systémy, cílem tedy je síť implementovat s ohledem na architekturu. Použitý mikrokontroler STM32F428ZIT6 má k dispozici CMSIS DSP, popsáno v 3.7, následující kapitola se zabývá využitím DPS, skrze dostupnou knihovnu, k akceleraci neuronové sítě. Kapitola také obsahuje sekci zabývající se nahrazením typů s plovoucí řádovou čárkou za typ s pevnou řádovou čárkou a menším rozsahem.

5.1 Profilace slabých míst implementace

Aby bylo možné implementaci optimalizovat, je výhodné identifikovat slabá místa, tedy části kódu, které se provádějí nejdéle, nejčastěji nebo obecně zabírají spoustu výpočetního výkonu. Mezi nejčastěji aplikované operace při dopředném výpočtu sítě patří skalární součin, konvoluce a aktivační funkce. Při inicializaci se také nuluje velký prostor paměti, protože většina operací své výsledky musí akumulovat přes smyčky. Všechny operace také pracují s přesnými čísly pohyblivé řádové čárky (*float_32t*), jejich převod na typ s pevnou řádovou čárkou, jako je například *integer*, by mohl způsobit urychlení za cenu ztráty jen malé přesnosti, jak je blíže popsáno v kapitole 3.9.

Dále parametry sítě ve 32 bitovém formátu zabírají velký prostor, což je problém obzvláště pokud pracujeme s tak omezenými zdroji jaké má k dispozici mikrokontroler. Formou optimalizace by tak určitě bylo i reprezentovat parametry sítě pomocí čísel zakódovaných na méně bitech, například pomocí Q formátu, který je popsaný v kapitole 3.8.1.

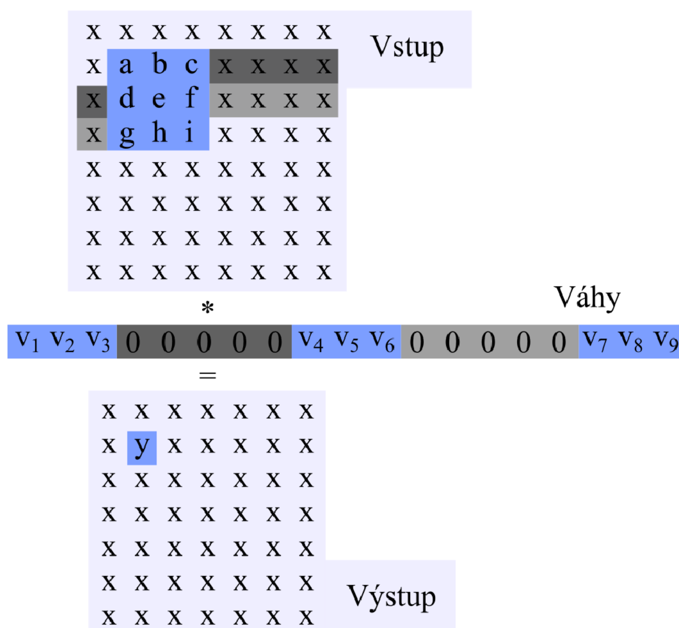
5.2 Skalární součin

Nejčastější výpočet, který probíhá ve všech vrstvách napříč celým dopředným vyhodnocením sítě je skalární součin. Protože neoptimalizovaná verze sítě obsahuje tuto funkci často, vybízí se její přímočarou implementaci, pomocí akumulátoru a jednoho cyklu, nahradit za funkci *arm_dot_prod_f32* poskytovanou knihovnou *CMSIS DSP Software Library*. Při bližším prozkoumání zdrojových souborů této funkce lze vidět, že provádí rozbalování smyček a počítá čtyři výstupy zároveň pokud je rozpoznáno zařízení rodiny procesorů Cortex-M4 nebo Cortex-M3. Toto slibuje značné urychlení výpočtu.

5.3 Konvoluce

Knihovna DSP je určená pro zpracování signálů, konvoluce je tedy jedna z jejích hlavních funkcí. Knihovna poskytuje několik verzí konvoluce. Rozdělení na nejvyšší úrovni je na konvoluci a částečnou konvoluci. Konvoluce, v plné verzi, počítá všechny hodnoty konvoluce signálů od počátku, kdy se překrývají jen jednou hodnotou, přes všechna plná překrytí, až po konec, kdy se signály opět kryjí jen jednou hodnotou. Délka výstupu funkce *arm_conv_f32* je tak vždy rovna součtu délek obou vstupů mínus jedna. Částečná konvoluce umožňuje zadat parametrem od které hodnoty a kolik výsledků se má počítat, není tak nutné počítat hodnoty, kde se signály plně nepřekrývají. Další dělené je na základě typů mezi *float_32t* a *Q* typ. Konvoluce pro Q typ se dále dělí na rychlou konvoluci, která nehlídá přetečení a má pouze jeden ochranný bit, konvoluci se *scratch* buffery a klasickou.

Protože se jedná o knihovnu pro zpracování signálů a ne obrazu, neexistuje varianta konvoluční funkce, do které by šla poslat 2D matice (obraz) nebo její výsek. Aby bylo možné použít existující funkci pro částečnou konvoluci s parametry ve formátu s plovoucí řádovou čárkou (*arm_conv_partial_f32*) je nutné přizpůsobit vstup funkce. Jednou z možností, jak toho dosáhnout by bylo pracovat s obrázkem po řádcích a provést jejich konvoluci vždy s jedním řádkem matice vah. Toto by se muselo vypočítat pro každý řádek matice vah, dílčím výstupem by tedy bylo n matic, kde n je počet řádků matice vah. V těchto datech by stačilo sečíst opět n odpovídajících čísel v jednom sloupci, ale každé z jiné dílčí matice, čímž by se získal výsledek konvoluce jednoho okna. Pro další okna by n matic nebylo nutné počítat znova, stačilo by sečíst další odpovídající hodnoty ze sloupců. Tento přístup vyžaduje velké množství režie, mezi výpočtů a celkově konvoluce probíhá pouze s krátkými úseky hodnot, což je značně nevýhodné.

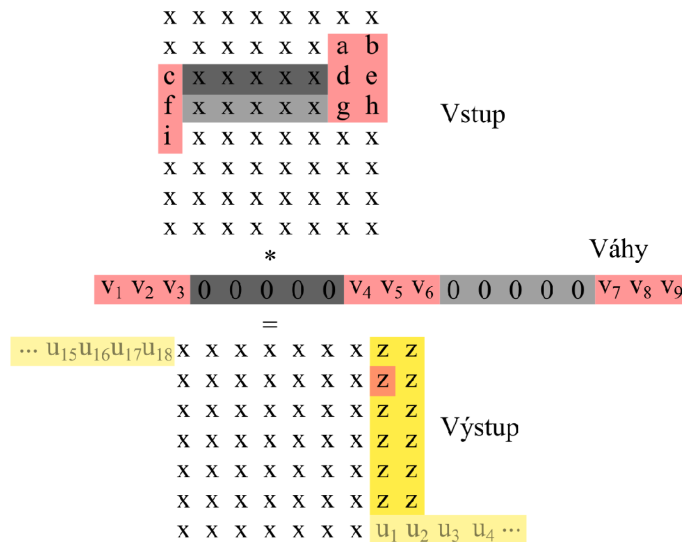


Obrázek 5.1: Vyplnění nulami a zarovnání pro optimalizovanou konvoluci

Lepším způsobem, jak přizpůsobit vstup pro 1D konvoluci je použít vyplnění nulami, zobrazeno na obrázku 5.1. Na obrázku můžeme vidět, že počet hodnot vah narostl z 9 na 19, protože přibily nuly. Díky přidaným nulám lze vypočítat konvoluci celého okna na

jednou, dochází sice k několika zbytečným násobením nulou v rámci konvoluce, ale není nutná žádná opakovaná režie navíc. Dokonce stačí funkci *arm_conv_partial_f32* pro celý vstup zavolat pouze jednou, jak se bude postupně "signál" vah, rozšířený o nuly, posouvat přes celý vstup vypočítá všechny výstupní hodnoty. Rozšíření o nuly je tedy nutné, aby se zachovalo posuvné okénko jak je zobrazeno na obrázku 4.3. Za výhodu lze považovat i možnost rozšíření o nuly provést staticky. Skript popsaný v kapitole 4.3.4 může už během export váhy o nuly rozšířit, získáme tak rychlejší výpočet na úkor paměti.

Výsledek bude ale obsahovat navíc několik špatných hodnot. Na začátku a na konci výstupu bude $h - 1$ neplatných čísel, kde h je celkový počet hodnot v "signálu" vah, ty vzniknou neúplným překryvem signálů na začátku a na konci konvoluce. Naštěstí funkce pro částečnou konvoluci počítá přesně s tímto případem a stačí jí parametrem poslat kolik čísel na začátku přeskočit a kolik jich poté spočítat, tím se tento problém vyřeší. Další výsledky navíc vzniknou při přechodu vah přes okraj (délku) řádku matice vstupu, toto je zobrazeno na obrázku 5.2 jako čísla z. Čísla u_i pak reprezentují neúplné překrytí na začátku a na konci konvoluce.



Obrázek 5.2: Ukázka výpočtu konvoluce s přebytečnými hodnotami

Poslední extrémní možností, jak zkonvertovat 1D konvoluci na 2D by bylo rozšířit pole s uloženými váhami o spoustu dalších nul tak, aby stačilo pro celou vrstvu zavolat funkci *arm_conv_partial_f32* pouze jednou. Aby si ale jednotlivá okénka neovlivňovaly výsledky muselo by vyplnění nulami být dlouhé minimálně $n - h$, kde h je stále celkový počet čísel vah a n je počet čísel vstupu. Tato konvoluce už počítá velké množství zbytečných hodnot, jenž většinou vyjdou nula, a nevyplatí se v porovnání s režii vzniklou v minulém případě, tedy když počítáme každý neuron zvlášť.

5.4 Reprezentace čísel

Obecně jde tvrdit, že výpočty pracující s čísly s pohyblivou řádovou čárkou by měly být pomalejší než výpočty s jednodušším typem s pevnou řádovou čárkou jako je například *integer*, je to dáno komplexností jednotlivých reprezentací a komplexitou jejich dílčích výpočtů. V dnešní době však pro typy s pohyblivou řádovou čárkou existuje specializovaný hardware,

jako je FPU, takže nelze jednoznačně tvrdit že *int32_t* bude rychlejší jak *float32_t*. Ve vestavěných zařízeních je situace ještě komplikovanější, protože přítomnost FPU na malých a levných čipech není samozřejmostí. STM32F428ZIT6 FPU obsahuje, i přesto je ale možné, že výpočet pracující s čísly s pevnou řádovou čárkou bude o něco rychlejší. Rozhodl jsem se proto vyzkoušet celou síť převést z typu *float32_t* na typ s pevnou řádovou čárkou. Jako cílový typ jsem zvolil Q7.9, typ je blíže popsaný v kapitole 3.8.1, jedná se tedy o typ reprezentující hodnoty na 16 bitech. Vyzkoušel jsem i 8 bitový datový typ, ale ten neposkytuje dostatečnou rozlišovací schopnost.

Zvolil jsem typ Q7.9 protože poskytuje rozsah [-64, 63.99805], hodnoty uvnitř neuronové sítě se sice většinou pohybují kolem 0, ale pro úspěšnou klasifikaci je potřeba použít i desítkový řád. Q7.9 tak poskytuje kompromis, jenž dokáže reprezentovat desetinná čísla s dostatečnou přesností a zároveň okamžitě nepřeteče i pro čísla větší. Nevýhodou však je že knihovna DSP tento datový typ nepodporuje a není proto možné použít její funkce. V síti je tedy nutné nahradit všechny datové typy na Q7.9, je také potřeba konvertovat hodnoty vah a biasů. Toto je naštěstí možné provést okamžitě při jejich extrakci ze zdrojové sítě, jedná se o rozšíření skriptu popsaného v kapitole 4.3.4. Průběh výpočtů dále značné zpomalí všudy přítomné kontroly saturace, protože datový typ je značně omezený rozsahem je nutné při každém výpočtu kontrolovat přetečení. V poslední řadě je také nutné před výpočtem konvertovat hodnoty vstupního čísla.

Kapitola 6

Vyhodnocení

V této kapitole se zabýváme použitými způsoby pro měření, seznámíme se s dosaženými výsledky a proběhne porovnání jednotlivých implementací v kontextu různých stupňů optimalizace. Porovnávají se tedy jednotlivé verze celých sítí i profilované jednotlivé funkce, zkoumá se nejen rychlost ale i prostorová náročnost některých verzí.

6.1 Způsob měření

Aby bylo možné dosažené výsledky nějak hodnotit, potřebujeme být schopni porovnávat jednotlivé implementace. Zajímá nás především spotřeba, což je příkon krát čas. Vzhledem k tomu, že příkon je více méně stejný, stačí se zaměřit na časovou náročnost, ta nám tím pádem přímo určuje náročnost energetickou. Měříme tedy dlouho trvá jedno dopředné vyhodnocení sítě. V této sekci je rozebráno, jakými způsoby se dá doba vykonání měřitjak.

6.1.1 Měření vnitřním čítačem

Mikrokontrolery z rodiny STM32F4 obsahují na svých čipech periferii *general purpose timer*, kapitola 3.3.3, ta mimo jiné umožňuje dopočítat se k nějaké hodnotě a poté vygenerovat přerušení, jenž spustí rutinu pro obsluhu přerušení, popsáno v kapitole 3.5. Tento mechanismus, na rozdíl od aktivního čekání, umožňuje vykonávat program a přitom měřit čas. Měřit čas v tomto kontextu neznamená měřit reálný čas, ale spíše počet vygenerovaných přerušení. Přerušení je sice možné pomocí správného nastavení časovače generovat například po 1 ms, ale výsledná hodnota bude vždy odvislá od přesnosti frekvence systémových hodin. Tato nepřesnost nám ale pro tento způsob měření nevadí, protože chceme především zjistit zrychlení jedné implementace vůči druhé. Zajímá nás tak poměr rychlostí, na konkrétních hodnotách tolik nezáleží, stačí ponechat nastavení časovače stejné během měření. Obecně ale lze u časovače nastavit dělička, tedy hodnota, kterou se vydělí vstupní systémová frekvence (pro STM32F4 Discovery maximálně 180MHz), a také perioda, maximální velikost čítače, po jejíž uplynutí se například vygeneruje přerušení. V této práci se tedy čítač podle dopočítané frekvence inkrementuje a jakmile dosáhne maximální hodnoty vygeneruje přerušení. Obsluha přerušení inkrementuje proměnnou, která tak počítá celkový potřebný čas pro vykonání.

6.1.2 Měření osciloskopem

Druhou možností, jak měřit dobu vykonání a získat přitom i reálný čas běhu je použít osciloskop. Tato metoda je jednodušší ale vyžaduje přístup právě k osciloskopu. Pomocí GPIO stačí invertovat hodnotu pinu, ke kterému je připojen osciloskop, vždy po dokončení dopředného výpočtu a provádět výpočet sítě stále dokola. Osciloskop nám ukáže signál, který by měl být podobný obdélníkovému signálu se střídou 50 % a jehož jedna perioda bude odpovídat dvěma iteracím výpočtu. Takto naměřené hodnoty udávají reálný čas vykonání pro každou naměřenou implementaci nebo zvolenou iterovanou funkci. Konkrétně byl k měření použit osciloskop **DSO1004A** firmy **Agilent Technologies**.

6.2 Porovnání implementací

6.2.1 Náhrada DSP funkcí

Jako první jsou v tabulce B.2 v příloze B zobrazeny rychlosti měřené vnitřním čítačem, jak byly postupně optimalizovány části implementace. Všechny běhy jsou pro sadu 36 čísel, s nulovou optimalizací (-o0), které se vykonají sériově za sebou pěti vrstvou sítí specifikovanou v kapitole 4.3.2. Hodnoty jsou naměřené pomocí vnitřního čítače a měli by být podobné skutečným hodnotám v milisekundách. V tabulce označení l_i označuje i-tou vrstvu a popisek vždy udává co konkrétně ve vrstvě bylo optimalizováno. Optimalizace v tomto kontextu znamená nahrazení původní mnou vytvořené funkce voláním funkce z knihovny DPS. Z dat v tabulce je jasně patrné, že největší efekt na rychlost mělo nahrazení konvolucí. Konkrétně nejlepších výsledků dosáhla konvoluce, která zapisovala data přímo do výstupního bufferu a nealokovala zbytečně prostor pro mezi výsledky.

Z dat v tabulce B.2 je očividné, že náhrady dalších funkcí jako například skalární součin, přičítaní biasů nebo hledání maxima v pooling funkci nepřináší v poměru s konvolucí významný přínos. Poslední položka v tabulce udává naměřené hodnoty pro síť, která používá pouze jedno volání konvoluce pro celou vrstvu. Jedná se o jeden z přístupů popsaných v kapitole 5.3, který využívá přidání nul jak do matice vah, tak mezi jednotlivé matice vah, aby bylo možné konvoluční funkci zavolat jen jednou. Z výsledků měření je jasné, že režie volání více funkcí je menší než zbytečné výpočty vzniklé přidáním nul.

Nicméně tabulka B.1 ukazuje, že pro hodnoty měřené s aktivní optimalizací překladače neodpovídají výsledky trendům pozorovaným při měření bez optimalizace. Data jsou také získána z vnitřního čítače a všechna ostatní nastavení kromě optimalizace jsou identická předchozímu měření. Zde je naopak jasně vidět, že síť s nahrazenou konvolucí, používající verzi z DSP knihovní je pomalejší. Náhrada ostatních funkcí stále zlepší rychlost výpočtu, ale jen v omezené míře.

6.2.2 Porovnání dílčích funkcí

V rámci zkoumání jednotlivých verzí jsem vytvořil jednoduchý modul pro porovnání rychlosti různých funkcí. Jedná se o postupná volání funkcí se stejně velkým vstupem, přičemž se měří doba vykonání pomocí invertování GPIO pinu a osciloskopu, způsob měření je popsán v kapitole 6.1.2. Podstatné výsledky ukazují tabulky 6.1 a 6.2.

V tabulce 6.1 se porovnávají implementace pro skalární součin, první dva záznamy jsou mnou implementované funkce a druhé dva jsou funkce z knihovny DSP, měření probíhalo se dvěma vektory po 20 000 vzorcích. Z výsledků mezi mnou implementovanými funkcemi pro typ *float32_t* a typ *q9_t* jde vidět, že funkce je dokonce rychlejší pro větší typ *float32_t*.

Funkce	**Čas vykonání (-o0/-o3)[ms]**
dot_product_f32	4.90 / 1.12
dot_product_q9	7.00 / 1.42
arm_dot_prod_f32	2.74 / 0.76
arm_dot_prod_q15	2.85 / 0.0048

Tabulka 6.1: Skalární součin, měřeno se dvěma vektory po 20 000 vzorcích

Toto je částečně dáno přidanými kontrolami saturace. Naproti tomu knihovní funkce s maximálním stupněm optimalizace -o3 je mnohonásobně rychlejší pro typ q15. Knihovna DSP využívá pro 16 bitový typ q15, volaní ___*SIMD32* funkce a skládá tak více dat do výpočtu realizovaného jednou instrukcí.

Celkově ale rozdíl mezi mnou implementovaným skalárním součin a knihovní verzí pro typ *float32_t* není až tak velký. I bližší průzkum assembler instrukcí neodhalil žádné zásadní rozdíly. V principu se obě implementace chovají stejné a knihovna nevyužívá žádných speciálních instrukcí, smyčky sice rozbaluje, ale k žádné souběžnému vykonání nedochází.

Výsledky samostatných konvolučních funkcí jsou v tabulce 6.2. První dva záznamy jsou pro funkce implementovány samostatně, poslední záznam odpovídá funkci, která používá volání konvoluce z knihovny DSP. Jedná se o verzi popsanou v druhé části kapitoly 5.3, tedy konvoluce probíhá pro každý neuron v jednotlivých vrstvách a vymezení nulami je pouze v rámci jedné matice vah. Hodnoty byly měřeny metodou s osciloskopem a jako vstup sloužily data o velikost 20x20 a 5x5. Podle výsledků je zjevné, že i pro toto nastavení se nevyplatí použít knihovní funkci, protože se musí přidat nuly a počítají se zbytečné hodnoty. Mezi typem *q9_t* a *float32_t* není zásadní rozdíl co se rychlosti týče.

Funkce	**Čas vykonání (-o0/-o3)[ms]**
convolution_additive_f32	38.00 / 3.98
convolution_additive_q9	38.00 / 4.48
convolution_additive_f32_optimized	44.00 / 11.80

Tabulka 6.2: Konvoluční 2D funkce, měřeno na matici 20x20 s filtrem 5x5

6.2.3 Výsledné porovnání sítí

Celkové výsledky pro jednotlivé finální sítě jsou v tabulce 6.3, jedná se měření skutečného času vykonání, pomocí osciloskopu, jednoho dopředného vyhodnocení sítě. Data potvrzují závěry z kapitoly 6.2.1, bez optimalizace překladače je jednoznačně lepší použít funkce DSP knihovny, ale s optimalizací (libovolnou, rozdíly mezi nimi jsou zanedbatelné) jsou funkce, které nepoužívají konvoluci z DSP knihovny rychlejší. Síť konvertovaná na 16 bitový typ Q7.9 je sice pomalejší než implementace z *float32_t*, ale na druhou stranu zabírá v paměti jen pouze skoro polovinu místa.

Zajímavým rozporem v konzistencí jsou naměřené hodnoty pro samotnou konvoluci z tabulky 6.2, které jasně říkají, že je konvoluce využívající funkci DSP knihovny pomalejší i pro nulovou (-o0) optimalizaci překladače. Jedno možné vysvětlení je velikost vstupních dat. Je možné, že pokud by se rychlost samotné konvoluce měřila na množství dat odpovídající

těm použitým v síti, tak by se výsledek pro nulovou optimalizaci překlopil tak, aby byla konvoluce využívající knihovnu rychlejší.

Síť	**Čas vykonání (-o0/-o1/-o2/-o3)[ms]**
původní síť (*float32_t*)	1130 / 127 / 108 / 108
nahrazeny všechny funkce (*float32_t*)	655 / 182 / 166 / 166
nahrazeny všechny funkce, kromě konvoluce (*float32_t*)	1120 / 125 / 108 / 108
původní síť (*q9_t*)	1350 / 152 / 152 / 153

Tabulka 6.3: Sítě, popsány v kapitole 4.3.2, měřeno jedno dopředné vyhodnocení

Kapitola 7

Závěr

Cílem této práce bylo navrhnout efektivní aplikaci pro vestavěné systémy realizující konvoluční neuronovou síť schopnou klasifikovat čísla z databáze MNIST. Tuto aplikaci poté implementovat se zaměřením na její rychlost a paměťové nároky v zařízení STM32F429 Discovery. V teoretickém úvodu bylo představeno fungování neuronových sítí se zaměřením na konvoluční neuronové sítě. Dále byly obecně představeny procesory ARM Cortex řady M a konkrétně cílové zařízení STM32F429 Discovery s Cortex-M4 procesorem. V rámci návrhu efektivní aplikace jsou také popsány přístupy k zpracování digitálních signálů ve vestavěných zařízeních, definované standardy a již známé přístupy optimalizace sítí na HW úrovni.

Jako součást diplomové práce byla navržena síť s 5 vrstvami, kde první dvě konvoluční vrstvy jsou následované dvěma plně propojenými, všechny využívající ReLU jako aktivační funkci, a jednou SoftMax vrstvou. Tato síť se stovkami neuronů poskytuje velmi dobré výsledky v řádech 99 % při klasifikaci MNIST číslic, samozřejmě míra úspěšnosti také záleží na způsobu a době trénování. Velikost sítě byla navržena s ohledem na omezenou paměť STM32F429 Discovery. Trénování samotné bylo realizováno knihovnou Theano v Pythonu na klasickém PC. Učením získané hodnoty vah a biasů sítě byly poté extrahovány a konvertovány do formy vhodné pro reimplementaci sítě ve vestavěném zařízení v jazyce C.

Specifikovaná pěti-vrstvá síť, konkrétně její část vykonávající dopředné vyhodnocení, tedy samotnou klasifikaci, byla převedena do vestavěného zařízení bez použití specializovaných knihoven nebo frameworků. Následoval proces optimalizace se zaměřením na maximální využití DSP knihovny, zejména pro funkce konvoluce a skalárního součinu. Tyto funkce tvoří většinu výpočtu při vyhodnocení sítě. Výsledky použití knihovny ukázaly, že velmi záleží na nastavení stupně optimalizace překladačem. Knihovnu se pro konvoluci vyplatí použít, pokud je optimalizace překladačem vypnutá, zrychlení je dvojnásobné, při zapnuté optimalizace to již výhodné není. Ostatní funkce při nahrazení jejich knihovními variantami nedosahují významného zlepšení v kontextu testované sítě.

Další zkoumané vylepšení efektivity bylo použít v síti místo 32 bitového typu s pohyblivou řádovou čárkou typ s pevnou řádovou čárkou Q7.9. Konvertovaná síť sice nevykazuje zlepší v rychlosti, při jednoduchém nahrazení typu, ale ani neztrácí přesnost ve významné míře, všech 35 testovacích čísel stále určí korektně. Přitom však potřebná velikost paměti pro uložení hodnot sítě klesla na polovinu.

Potenciál menších datových typů ale rozhodně není v této práci vyčerpán úplně. Knihovna DSP sice neimplementuje své funkce pro specifický typ Q7.9 ale určitě by bylo možné její funkce blíže prozkoumat a navrhnout podle nich i velmi efektivní SIMD variantu pro Q7.9. Zajímavé by také mohlo být vyzkoušet na mikrokontroleru mnohem větší síť s daleko více

vrstvami. Parametry sítě by bylo možné uložit do pomalejší SDRAM paměti, která má ale kapacitu 8 MB.

Datový typ parametrů sítě by dále určitě šel posunout ještě níže z 16 bitů například na 8. Bylo by však nutné sít s takovýmto omezením již trénovat. Jak ukazují jiné práce, tímto přístupem je možné používat i binární váhy. Malá přesnost vah zabraňuje přetrénování sítě.

Literatura

[1] ARM, [Online; navštíveno 16.01.2018].
URL https://www.arm.com/assets/images/compare-Cortex-M-diagramLG.png

[2] *Definition of overfitting in English*. Oxford dictionaries, [Online; navštíveno 18.12.2017].
URL https://en.oxforddictionaries.com/definition/overfitting

[3] *Embedded System*. Techopedia, [Online; navštíveno 18.12.2017].
URL https://www.techopedia.com/definition/3636/embedded-system

[4] *Understanding Activation Functions in Neural Networks*. medium.com, [Online; navštíveno 01.01.2018].
URL https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

[5] *X86*. TechTerm, 2018, [Online; navštíveno 29.04.2018].
URL https://techterms.com/definition/x86

[6] Abadi, M.; Agarwal, A.; Barham, P.; aj.: TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. 2015, software available from tensorflow.org.
URL https://www.tensorflow.org/

[7] Andri, R.; Cavigelli, L.; Rossi, D.; aj.: *YodaNN: An Architecture for Ultra-Low Power Binary-Weight CNN Acceleration*. [Online; navštíveno 16.01.2018].
URL https://arxiv.org/abs/1606.05487?context=cs

[8] ARM: *Developer Suite (v1.2) AXD and armsd Debuggers Guide*, DUI 0066D. Listopad 2001.

[9] Cavigelli, L.; Benini, L.: *Origami: A 803 GOp/s/W Convolutional Network Accelerator*. [Online; navštíveno 16.01.2018].
URL https://arxiv.org/abs/1512.04295

[10] Chaitanya, C. R. A.; Kaplanyan, A.; Schied, C.; aj.: *Interactive Reconstruction of Monte Carlo Image Sequences using a Recurrent Denoising Autoencoder*. [Online; navštíveno 20.12.2017].
URL http://research.nvidia.com/publication/interactive-reconstruction-monte-carlo-image-sequences-using-recurrent-denoising

[11] Courbariaux, M.; Bengio, Y.; David, J.-P.: *BinaryConnect: Training Deep Neural Networks with binary weights during propagations*. [Online; navštíveno 16.01.2018].
URL https://arxiv.org/abs/1511.00363v3

[12] Dieleman, S.; Schlüter, J.; Raffel, C.; aj.: *Lasagne: First release.* Srpen 2015, [Online; navštíveno 13.05.2018].
URL http://dx.doi.org/10.5281/zenodo.27878

[13] Fisher, T.: *What Is Firmware?* Lifewire, Březen 2018, [Online; navštíveno 29.04.2018].
URL https://www.lifewire.com/what-is-firmware-2625881

[14] George B. Thomas, J.; Finney, R. L.: *Calculus and Analytic Geometry.* Addison-Wesley Publishing Company, 1998, ISBN 0-201-40015-4.

[15] Gomez, R. G.: *Deep Learning frameworks: a review before finishing 2016.* Prosinec 2016, [Online; navštíveno 07.03.2018].
URL https://buzzrobot.com/deep-learning-frameworks-a-review-before-finishing-2016-5b3ab4010b06

[16] ISO/IEC: *9899:TC2 WG14/N1124.* Květen 2005.

[17] Jia, Y.; Shelhamer, E.; Donahue, J.; aj.: Caffe: Convolutional Architecture for Fast Feature Embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[18] LeCun, Y.: *LeNet-5, convolutional neural networks.* [Online; navštíveno 01.01.2018].
URL http://yann.lecun.com/exdb/lenet/

[19] Mehrotra, K.; Mohan, C. K.; Ranka, S.: *Elements of Artificial Neural Networks.* Cambridge : MIT Press, 1997, ISBN 0-262-13328-8.

[20] Microsoft: *Microsoft Cognitive Toolkit (CNTK), an open source deep-learning toolkit.* [Online; navštíveno 11.04.2018].
URL https://www.microsoft.com/en-us/cognitive-toolkit/

[21] Nielsen, M.: *Neural Networks and Deep Learning.* [Online; navštíveno 18.12.2017].
URL http://neuralnetworksanddeeplearning.com

[22] Nomi, T.: *tiny-dnn documentations.* 2016, [Online; navštíveno 17.05.2018].
URL http://tiny-dnn.readthedocs.io/en/latest/index.html#

[23] Russell, S. J.; Norvig, P.: *Artificial Intelligence A Modern Approach Third Edition.* Prentice Hall, 2010, ISBN 0-13-604259-7.

[24] Sirio, G. D.: *RTOS Concepts.* 2017, [Online; navštíveno 29.04.2018].
URL http://www.chibios.org/dokuwiki/doku.php?id=chibios:articles:rtos_concepts

[25] Stallman, R. M.; aj.: *Using the GNU Compiler Collection.* 2018.
URL https://gcc.gnu.org/onlinedocs/gcc/

[26] STMicroelectronics: *STM32F4DISCOVERY, Discovery kit with STM32F407VG MCU*, DocID022204 Rev 6. Říjen 2016.

[27] STMicroelectronics: *RM0090, Reference manual, STM32F405/415, STM32F407/417, STM32F427/437 and STM32F429/439 advanced ARM ® -based 32-bit MCUs*, DocID0 18909 Rev 15. Červenec 2017.

[28] STMicroelectronics: *UM1725, User Manual, Description of STM32F4 HAL and LL drivers*, DocID025834 Rev 5. Únor 2017.

[29] Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, ročník abs/1605.02688, Květen 2016. URL http://arxiv.org/abs/1605.02688

[30] Yiu, J.: *The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors* . Oxford : Elsevier ; Newnes, 2014, ISBN 0-12-408082-0.

Příloha A

Obsah přiloženého paměťového média

Přiložené CD obsahuje veškeré zdrojové kódy pro SW i Latex v následující struktuře:

- **/doc** – text práce a zdrojové kódy pro program LaTeX
- **/src** – zdrojové soubory:
  - **/Python_PC** – neuronová síť a skript pro export vah a biasů
  - **/C_network_MCU** – neuronová síť, profilace funkcí a optimalizace
  - **/STM32Cuve_FW_F4_V1.19.0** – knihovny pro vestavěné zařízení
- **/bin** – přeložené zdrojové kódy pro vestavěné zařízení, různé verze sítě
- **/tools** – pomocné softwarové nástroje
- **/MNIST** – data databáze čísel MNIST

Příloha B

Měření postupné optimalizace

Úroveň optimalizace sítě	**Čas vykonání (-o1/-o2/-o3)[ms]**
Neoptimalizovaná síť	4 892 / 3 972 / 3 974
l_0 a l_1 - konvoluce, bez alokace, l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin l_0, l_1 - hledání maxima uvnitř poolingu	6 588 / 5 831 / 5 817
l_0 a l_1 - konvoluce, bez alokace, l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin l_0, l_1 - hledání maxima uvnitř poolingu, l_0, l_1, l_2, l_3, l_4 offset/součet biasů + aktivační funkce (bez l_4)	6 602 / 5 842 / 5 805
- l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin l_0, l_1 - hledání maxima uvnitř poolingu	4 521 / 3 887 / 3 877
- l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin l_0, l_1 - hledání maxima uvnitř poolingu, l_0, l_1, l_2, l_3, l_4 offset/součet biasů + aktivační funkce (bez l_4)	4 525 / 3 889 / 3 879

Tabulka B.1: Dopředné vyhodnocení 36 čísel, měřeno vnitřním čítačem

Úroveň optimalizace sítě	**Čas vykonání** [ms]
Neoptimalizovaná síť	40 866
l_0 - konvoluce, s alokací paměti (*float_32* x1000)	39 656
l_0 - konvoluce, bez alokace	34 785
l_0 a l_1 - konvoluce, bez alokace	23 823
l_0 a l_1 - konvoluce, bez alokace, l_2 - skalární součin	23 542
l_0 a l_1 - konvoluce, bez alokace, l_2 a l_3 - skalární součin	23 542
l_0 a l_1 - konvoluce, bez alokace, l_2, l_3 a l_4- skalární součin	23 490
l_0 a l_1 - konvoluce, bez alokace, l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin	23 376
l_0 a l_1 - konvoluce, bez alokace, l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin l_0 - hledání maxima uvnitř poolingu	23 379
l_0 a l_1 - konvoluce, bez alokace, l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin l_0, l_1 - hledání maxima uvnitř poolingu, l_0, l_2 offset/součet biasů + aktivační funkce	23 611
l_0 a l_1 - konvoluce, bez alokace, l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin l_0, l_1 - hledání maxima uvnitř poolingu, l_0, l_1, l_2 offset/součet biasů + aktivační funkce	23 607
l_0 a l_1 - konvoluce, bez alokace, l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin l_0, l_1 - hledání maxima uvnitř poolingu, l_0, l_1, l_2, l_3 offset/součet biasů + aktivační funkce	23 610
l_0 a l_1 - konvoluce, bez alokace, l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin l_0, l_1 - hledání maxima uvnitř poolingu, l_0, l_1, l_2, l_3, l_4 offset/součet biasů + aktivační funkce (bez l_4)	23 609
l_0 a l_1 - konvoluce, nutná alokace, na 1 vrstvu jen 1 l_2 (spojen i vnitřní cyklus), l_3 a l_4 - skalární součin l_0 - hledání maxima uvnitř poolingu	55 431

Tabulka B.2: Dopředné vyhodnocení 36 čísel, měřeno vnitřním čítačem, bez optimalizace překladačem, postupné nahrazení funkcí jejich verzemi z knihovny DSP