Realizace základních typů neuronových sítí v mikrokontrolerech

Michal Raška

Bakalářská práce
2023

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
Ústav informatiky a umělé inteligence

Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Michal Raška**
Osobní číslo:	**A20076**
Studijní program:	**B0613A140020 Softwarové inženýrství**
Forma studia:	**Kombinovaná**
Téma práce:	**Realizace neuronových sítí na mikrokontrolerech**
Téma práce anglicky:	**Implementation of Neural Networks on Microcontrollers**

Zásady pro vypracování

1. Vypracujte literární rešerši na zvolené téma a zaměřte se na číselné formáty a zvolené HW architektury.
2. Zvolte vhodné architektury neuronových sítí a proveďte jejich popis.
3. Realizujte neuronové sítě na zvolených mikrokontrolerech.
4. Zvolte vhodné datasety případně vhodnou praktickou testovací úlohu a implementujte testovací program.
5. Proveďte vyhodnocení s důrazem na přesnost a výkon.

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. SILVA, Ivan Nunes Da, Danilo Hernane SPATTI, Rogerio ANDRADE FLAUZINO, Luisa Helena Bartocci LIBONI a Silas Franco dos REIS ALVES. *Artificial neural networks: a practical course*. Switzerland: Springer, [2016], 1 online resource (xx, 307 pages). Dostupné z: doi:9783319431628
2. GRAUPE, Daniel. *Principles of artificial neural networks*. 2nd ed. Singapore: World Scientific, 2007, xv, 303 s. Advanced series on circuits and systems. ISBN 9812706240.
3. MANDIC, Danilo P. a Jonathon A. CHAMBERS. *Recurrent neural networks for prediction: learning algorithms, architectures and stability*. Chichester: John Wiley, c2001, xxi, 285 s. Wiley series in adaptive and learning systems for signal processing, communications, and control. ISBN 0471495174. Dostupné také z: http://www.loc.gov/catdir/toc/onix06/2001033418.html
4. HAYKIN, Simon S. *Neural networks and learning machines*. Third edition. Upper Saddle River: Pearson, [2009], 934 s. Pearson international edition. ISBN 9780131293762.
5. GRAUPE, Daniel. *Deep learning neural networks: design and case studies*. New Jersey: World Scientific, [2016], xvi, 263 s. ISBN 9789813146457.
6. DEAN, Alexander G. *Embedded systems fundamentals with ARM Cortex-M based microcontrollers: a practical approach*. Cambridge: ARM Education Media, [2017], xx, 292 s. ISBN 9781911531036.

Vedoucí bakalářské práce: **doc. Ing. Roman Šenkeřík, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání bakalářské práce: **2. prosince 2022**
Termín odevzdání bakalářské práce: **26. května 2023**

UNIVERZITA TOMÁŠE BATI VE ZLÍNĚ
Fakulta aplikované informatiky
L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 7. prosince 2022

Prohlašuji, že

- beru na vědomí, že odevzdáním bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že bakalářské práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky. Univerzity Tomáše Bati ve Zlíně;
- byl/a jsem seznámen/a s tím, že na moji bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – bakalářskou práci nebo poskytnout licenci k jejímu využití jen připouští-li tak licenční smlouva uzavřená mezi mnou a Univerzitou Tomáše Bati ve Zlíně s tím, že vyrovnání případného přiměřeného příspěvku na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše) bude rovněž předmětem této licenční smlouvy;
- beru na vědomí, že pokud bylo k vypracování bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu využití), nelze výsledky bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji,

- že jsem na bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze bakalářské práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne .....

podpis studenta

ABSTRAKT

Bakalářská práce se zabývá možnostmi uplatnění neuronových sítí v oblasti mikrokontrolerů s omezeným výpočetním výkonem a pamětí s využitím čísel s pevnou řádovou čárkou. Cílem je zvýšení schopností autonomních systémů v oblasti robotiky, Internetu věcí (IoT), průmyslových, zabezpečovacích a dalších aplikací. Práce má dvě části, teoretickou a praktickou. V teoretické části jsou popsány dvě možnosti reprezentace reálných čísel v počítačové technice - formáty s pohyblivou a pevnou řádovou čárkou a jejich vlastnosti. Následuje základní popis neuronových sítí typu perceptron, Adaline, dopředná (feedforward) a konvoluční neuronová síť. Nakonec je popsána platforma procesorových jader ARM Cortex M0+ a M4F, historie jejího vzniku společnosti ARM a popis vybraných mikrokontrolerů založených na těchto jádrech. Praktická část se věnuje tvorbě knihovny v jazyce C pro základní matematické operace a datové struktury, které jsou využity pro implementaci knihoven realizujících vybrané typy neuronových sítí a učících algoritmů. Nakonec byly provedeny testy rychlosti a přesnosti, jak matematických funkcí, tak i vlastních naprogramovaných neuronových sítí. Na základě těchto testů byla vyhodnocena vhodnost a podmínky pro jejich budoucí využití v praktických aplikacích.

Klíčová slova: Neuronové sítě, ARM, jazyk C, pevná řádová čárka, plovoucí řádová čárka, strojové učení

ABSTRACT

The bachelor thesis deals with the possibilities of applying neural networks in the field of microcontrollers with limited computing power and memory using fixed-point numbers. The aim is to enhance the capabilities of autonomous systems in robotics, Internet of Things (IoT), industrial, security and other applications. The thesis has two parts, theoretical and practical. In the theoretical part, two options for representing real numbers in computing are described - floating-point and fixed-point formats and their properties. This is followed by a basic description of perceptron, Adaline, feedforward and convolutional neural networks. Finally, the ARM Cortex M0+ and M4F processor core platform is described, the history of its creation by ARM, and a description of

selected microcontrollers based on these cores. The practical part is devoted to the creation of a C library for basic mathematical operations and data structures, which are used to implement libraries implementing selected types of neural networks and learning algorithms. Finally, tests of speed and accuracy of both the mathematical functions and the actual programmed neural networks were performed. Based on these tests, the suitability and conditions for their future use in practical applications were evaluated.

Keywords: Neural networks, ARM, C language, fixed point, floating point, machine learning

Rád bych touto cestou vyjádřil poděkování vedoucímu mé bakalářské práce doc. Ing. Romanu Šenkeříkovi, Ph.D., za podnětné rady, metodickou a odbornou pomoc. Taktéž bych rád poděkoval konzultantovi Ing. Peteru Janků, Ph.D., za vstřícnost a ochotu. Poděkování patří i mé rodině za podporu, toleranci a obětavost po celou dobu mého studia. A děkuji také svým kolegům, kteří byli zdrojem inspirace, podpory a porozumění.

OBSAH

ÚVOD

Cílem této bakalářské práce je zkoumat možnosti využití čísel s pevnou řádovou čárkou (FX) a souvisejících matematických operací v oblasti umělých neuronových sítí. Práce se zaměřuje na porovnání výhod a nevýhod takových čísel oproti číslům s plovoucí řádovou čárkou, která jsou běžně používána v současných neuronových sítích.

Hlavním zaměřením práce je zkoumání možností implementace čísel s pevnou řádovou čárkou a příslušných operací v prostředí mikrokontrolerů s jádry ARM Cortex řady M. Tyto mikrokontrolery jsou široce používány v embedded systémech a představují značný potenciál pro efektivní provoz umělých neuronových sítí v zařízeních typu IoT, nebo nositelné elektroniky.

V rámci studie jsou porovnány různé metody implementace čísel s pevnou řádovou čárkou a jejich operací. Dále byl hodnocen jejich výkon, přesnost a účinnost v různých aplikacích umělých neuronových sítí oproti variantě s čísly s plovoucí řádovou čárkou. Práce také poskytuje přehled o dostupných nástrojích, knihovnách a metodách pro vývoj a optimalizaci neuronových sítí na mikrokontrolerech s jádry ARM Cortex řady M.

Výsledkem této bakalářské práce je přehled o možnostech a omezeních použití čísel s pevnou řádovou čárkou v umělých neuronových sítích na mikrokontrolerech, poskytuje užitečný základ pro další vývoj efektivních a energeticky úsporných neuronových sítí na těchto platformách.

I. TEORETICKÁ ČÁST

1 PREZENTACE ČÍSEL V POČÍTAČOVÉ TECHNICE

Pro prezentaci reálných čísel v oblasti výpočetní techniky existuje několik variant a jsou uvedeny v normách IEEE 754, IEC 60559, IEC 559. Základními variantami prezentace reálných čísel jsou formáty s plovoucí řádovou čárkou (Float Point dále FP) a s pevnou desetinou čárkou (Fixed Point dále FX).

Oba formáty se liší přesností, dynamickým rozsahem a rychlostí výpočtu. Výpočty ve formátu FP mohou mít velký dynamický rozsah, nicméně výpočet je obecně pomalejší, oproti tomu dynamický rozsah formátu FX je obecně menší, než je tomu u formátu FP, nicméně výpočet je téměř stejně rychlý, jako při použití celých čísel.

1.1 Formát s plovoucí řádovou čárkou

Formát FP definuje norma IEEE 754[1)]. Základní norma IEEE 754-1985[2)] definuje čtyři základní formáty:

- jednoduchá přesnost(32 bitů)
- dvojitá přesnost (64 bitů)
- rozšířená přesnost (>48 bitů)
- dvojitá rozšířená přesnost (79 bitů, v praxi 80 bit) $a_n < a_{n+1}$

Čísla v jednoduchém a dvojitém formátu se skládají z následujících tří polí:

1. Bit znaménka s.
2. Pozměněný exponent $e = E + bias$, kde bias je hodnota zajišťující, že exponent je vždy kladný.
3. Zlomkový segment f, tvořený hodnotami - $b_1, b_2 ... b_{p-1}$.

Organizace jednotlivých polí a bitů zobrazuje tabulka 1.1.

Tabulka 1.1. Reprezentace FP čísla s jednoduchou přesností

bit	31	30, 29 ... 24, 23	22, 21 ... 3, 2, 1, 0
Význam	s - znaménko	E - exponent (8 bitů)	f - mantisa (23 bitů)

Vybraná podmnožina racionálních čísel je pro formát FP vyjádřena vztahem (1.1).

[1)] https://ieeexplore.ieee.org/document/8766229
[2)] https://ieeexplore.ieee.org/document/30711

$$x_{fp} = (-1)s^{2E+bias}(b_0 - b_1, b_2...b_{p-1}) \tag{1.1}$$

kde s je bit znaménka nabývající hodnotu s = 0,1, E libovolné kladné číslo, pro které platí: $E \in < E_{min}, E_{max} >$, b_i jsou jednotlivé bity.

Dynamický rozsah pro jednoduchou přesnost (v jazyce C typ float) je $-3.40282347 \times 10^38 < x_{fp} < 3.40282347 \times 10^38$ a nejmenší možná zobrazitelná hodnota je $1,17549 \times 10^{-38}$.

U FP existují, kromě standardních hodnot, také specifické hodnoty, které zahrnují nedefinované hodnoty, zápornou a kladnou nulu a také záporné a kladné nekonečno. Jednotlivé typy jsou shrnuty v tabulce 1.2.

Tabulka 1.2. Reprezentace FP čísla s jednoduchou přesností

Význam hodnoty	s - znaménko	E - exponent	f - mantisa
Kladná nula	0	0	0
Záporná nula	1	0	0
Kladné nekonečno	0	255	0
Záporné nekonečno	1	255	0
Not a number (NaN)	0	255	f > 0
NaN	1	255	f > 0

Změna hodnoty způsobená konečnou velikostí mantisy není lineární, ale naopak se zvětšuje s rostoucí vzdáleností od počátku souřadnicového systému, což vede k poklesu rozlišení čísel s plovoucí řádovou čárkou (FP) čísel. Tento fenomén způsobuje nepřesnosti v matematických operacích s hodnotami, jejichž řády se výrazně liší. V důsledku toho, není tento formát čísel vhodný pro použití v určitých oblastech, jako je finančnictví a pro výpočty s velkým dynamickým rozsahem, kde se chyba postupně zvyšuje s každou další matematickou operací.

Pro tyto specifické scénáře existují v zásadě dvě možné cesty:

1. Zvýšit počet bitů pro reprezentaci čísla FP.
2. Použít matematiku s pevnou desetinnou čárkou.

Pomocí těchto opatření je možné eliminovat, nebo alespoň minimalizovat negativní dopad ztráty přesnosti.

1.2 Formát čísel s pevnou řádovou čárkou

Výhodou formátu čísel s pevnou řádovou čárkou FX je zvýšená rychlost matematických operací, menší složitost výpočetních jednotek, nižší náklady na systém a snížená

spotřeba elektrické energie. Tento formát nachází uplatnění v oblasti digitálního zpracování signálů, a to zejména ve specializovaných digitálních signálových procesorech (Digital Signal Processing dále DSP), matematických DSP jednotkách programovatelných hradlových polí (Field Programming Gate Array dále FPGA), a také v mikrokontrolerech, které nejsou vybaveny jednotkou pro výpočty s plovoucí řádovou čárkou (Float Point Unit dále FPU). V těchto případech formát FX poskytuje výrazně vyšší výpočetní rychlost ve srovnání s softwarově emulovanou FPU.

Dalším typem výkonných výpočetních jednotek, ve kterých se uplatní formát FX jsou hybridní systémy na čipu (System on Chip dále SoC), které kombinují jedno nebo více procesorových jader (Central Processing Unit dále CPU) a programovatelná hradlová pole FPGA. Použití aritmetiky s pevnou řádovou čárkou (FX) umožňuje dosáhnout optimálního výkonu při co nejnižší ceně, neboť realizované výpočetní jednotky jsou jednodušší, což umožňuje volbu menších a tedy levnějších variant těchto SoC nebo naopak dosažení vyššího výkonu než při použití aritmetiky s plovoucí řádovou čárkou FP.

Kombinace CPU a FPGA také umožňuje využití hardwarově akcelerovaných výpočtů prostřednictvím integrovaných jednotek s plovoucí řádovou čárkou (FPU) v CPU, které lze využít například pro normalizaci hodnot, kde se uplatní vyšší dynamický rozsah. Tyto hodnoty mohou být následně převedeny do FX formátu a zpracovány v jednotkách digitálního signálového zpracování (DSP) realizovaných pomocí FPGA části. Příkladem mohou být SoC od společnosti AMD (Xilinx) řady Zynq 7000, které kombinují jedno nebo dvě jádra ARM Cortex A9 s programovatelným hradlovým polem Artix-7.

Tyto SoC jsou používány v mnoha oblastech, jako jsou například:

- měřící přístroje (osciloskopy, spektrální analyzátory a další),
- signálové generátory a softwarově definované rádiové systémy,
- lékařské přístroje,
- audio - video technice,
- obrazové mapovací systémy u letadel, vrtulníků a řízených střel,
- radarové systémy jak ve vojenském, tak i civilním sektoru

Jednou z velkých výhod pevné řádové čárky FX aritmetiky je možnost výrazného zrychlení výpočtů funkcí s jedním vstupním parametrem prostřednictvím předvypočítaných tabulek (Look up Table LUT) v systému s dostatečným množstvím paměti. V tomto případě vstupní hodnota odpovídá adrese plus pevnému posunutí v tabulce,

kde je uložena výsledná hodnota. Pro 16bitové číslo s pevnou řádovou čárkou (Fixed point FXP16) to představuje $2^{16} = 65535$ kombinací. Skutečná alokace paměti je však dvojnásobek těchto kombinací, neboť jedna hodnota zabírá 2 bajty, což odpovídá 128 KiB paměti. Rychlost výpočtu je pak dána pouze časem potřebným k přečtení hodnoty z paměti. Tímto způsobem lze provádět poměrně složité výpočty s konstantní složitostí za cenu vyšších nároků na paměť pro data. Mapování vstupní proměnné na adresu v paměti v tabulce 1.3.

Tabulka 1.3. Mapování vstupní hodnoty funkce na $y = e^x$ výstupní hodnotu pro FX hodnotu Q5.10

vstupní hodnota		výstupní hodnota	
šestnáctkově	dekadicky	šestnáctkově	dekadicky
0x0000	0.00000	0x4000	1.00000
0x0001	0,00097	0x4000	1.00097
0x0002	0.00195	0x4002	1.00195
...			
0x8000	-32.00000	0x0000	0.00000
...			
0xA000	-24.00000	0x0000	0.00000
...			
0xB000	-20.00000	0x0000	0.00000
...			
0xE000	-7.99902	0x0000	0.00033
...			
0xF000	-3.99902	0x0013	0.01833
...			
0xFFFF	-0.00097	0x03FF	0.99903

Z tabulky označené jako 1.3 je patrné, že můžeme optimalizovat pro některé funkce využití paměti tím, že v určitém rozsahu funkčních hodnot je výstupní hodnota nižší než je rozlišovací schopnost zvoleného formátu. Takže pro tento interval je dostatečné určit, zda se vstupní hodnota nachází uvnitř daného intervalu. V příkladu s parametry uvedenými v tabulce 1.3 je jasné, že alespoň pro rozsah adres od 0x8000 do 0xB000 je hodnota funkce rovna 0x0000.

1.2.1 Formy zápisu čísel FX

Formát s pevnou řádovou čárkou se obvykle zapisuje jako $m.n + 1$ pro čísla se znaménkem nebo $m.n$ pouze pro kladné hodnoty. Alternativní značení, které používají například společnosti Texas Instruments, ARM a další [39], pro číslo se znaménkem má tvar:

$$Qm.n \tag{1.2}$$

a pro číslo bez znaménka pak:

$$Um.n \tag{1.3}$$

kde m je počet bitů celočíselné části, n počet bitů zlomkové části.

Celkový počet bitů w je dán vztahem pro číslo se znaménkem:

$$w = m + n + 1 \tag{1.4}$$

a pro číslo bez znaménka pak:

$$w = m + n \tag{1.5}$$

kde w je celkový počet bitů.

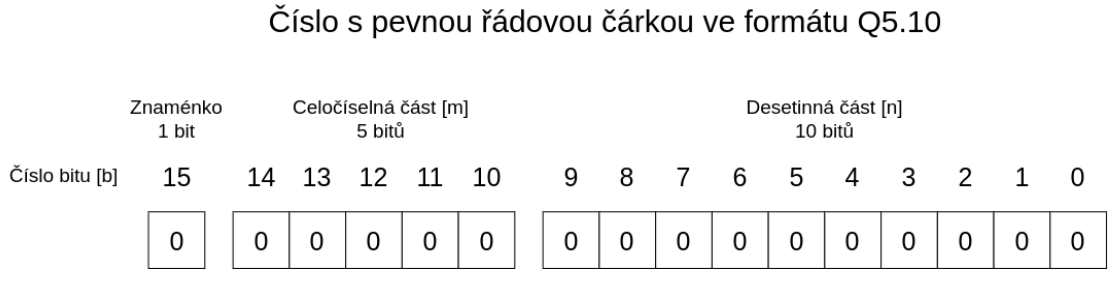
Poloha řádové čárky ovšem není součástí samotného čísla, protože je známa v rámci programu, nebo funkce a proto je možno při stejném počtu bitů dosáhnout většího rozlišení, než v případě čísel ve formátu FP. Příklad zápisu FX čísel:

Q8.7 představuje: 1 bit je znaménko, 8 bitů je celočíselná část a posledních 7 bitů patří zlomkové části. Celkem tedy 16 bitů.

U5.11 představuje: 5 bitů celočíselné části a 11 bitů zlomkové části, celkem 16 bitů.

1.2.2 Vlastnosti čísel ve formátu FX

Při použití formátu FX je klíčové zvolení správného rozlišení pro zajištění optimálního poměru mezi výpočetní přesností a zdroji potřebnými pro výpočet. To zahrnuje pečlivé zvážení počtu bitů vyhrazených pro celočíselnou a zlomkovou část čísla, aby byly zachovány důležité informace při minimalizaci zaokrouhlovacích chyb a saturace. Uspořádání bitů pro FX16 je na obrázku 1.1



Obrázek 1.1. Uspořádání bitů čísel FX16 formátu Q5.10

Váha bitu je dána vztahem (1.6):

$$W_x = 2^{x-n} \tag{1.6}$$

kde W_x je váha bitu, x pořadové číslo bitu zprava a n je počet bitů ve zlomkové části.

Pro hodnotu čísla daného v binárním tvaru při známém celkovém počtu bitů a polohy řádové čárky platí vztah (1.7).

$$Y_w = \sum_{x=0}^{w-2} b_x * 2^{x-n} \tag{1.7}$$

kde Y_w je hodnota FX, w celkový počet bitů, x pořadové číslo bitu zprava, b_x je hodnota bitu na pozici x a n je počet bitů ve zlomkové části.

V tabulce 1.4 jsou uvedeny minimální nenulové hodnoty a maximální hodnota v závislosti na počtu bitů ve zlomkové části čísla pro šestnácti bitové slovo.

Tabulka 1.4. Minimální nenulové hodnoty a rozsah kladné a záporné hodnoty v závislosti na pozici řádové čárky formátu FP [celá část. zlomková]

Pozice řádové čárky	Minimální nenulová hodnota	Maximální kladná hodnota	Maximální záporná hodnota
15.0	1.00000	32767.00000	-32768.00000
14.1	0.50000	16383.50000	-16384.00000
13.2	0.25000	8191.75000	-8192.00000
12.3	0.12500	4095.87500	-4096.00000
11.4	0.06250	2047.93750	-2048.00000
10.5	0.03125	1023.96875	-1024.00000
9.6	0.01562	511.98438	-512.00000
8.7	0.00781	255.99219	-256.00000
7.8	0.00391	127.99609	-128.00000
6.9	0.00195	63.99805	-64.00000
5.10	0.00098	31.99902	-32.00000
4.11	0.00049	15.99951	-16.00000
3.12	0.00024	7.99976	-8.00000
2.13	0.00012	3.99988	-4.00000
1.14	0.00006	1.99994	-2.00000
0.15	0.00003	0.99997	-1.00000

Poloha řádové čárky v závislosti na účelu použití hraje klíčovou roli. U operace násobení nedochází k přetečení, neboť výsledné číslo je po násobení normalizováno zpět na 16 bitů. V případě podílu, pokud výsledná hodnota je nižší, než nejmenší možná hodnota formátu Qm.n pak je výsledná hodnota nulová. Situace se však liší u operací sčítání a odečítání, kde je třeba zohlednit potenciální maximální hodnotu součtu nebo rozdílu, neboť při přetečení nedojde k saturaci hodnoty v případě požadavku na co

nejrychlejší operace.

Pokud se sčítají normalizovaná čísla v rozsahu od $x_i \in (-1, 1)$, poloha řádové čárky záleží na celočíselné části, která určuje maximální možnou hodnotu součtů. V případě, že všechny sčítané hodnoty dosahují extrémních hodnot (1 pro sčítání, -1 pro odečítání), je maximální hodnota součtu nebo rozdílu dána celkovým počtem těchto operací.

Důležité je, aby maximální hodnota součtu nebo rozdílu byla menší nebo rovna maximální hodnotě vybrané varianty Qm.n, aby nedošlo k překročení kapacity zvoleného formátu dle vztahu (1.8):

$$y_{max} \geq \sum_{i=1}^{k} (x_i) \qquad x_i \in (-1, 1) \tag{1.8}$$

kde y_{max} je maximální reprezentovatelná hodnota, k počet iterací součtu a $x_i = 1$

Platí tedy vztah pro potřebný počet bitu celočíselné části dle vztahu (1.9):

$$Qm = \frac{\ln(y_{max})}{\ln(2)} \tag{1.9}$$

kde Q_m je počet bitů celočíselné části čísla Qm.n a y_{max} je maximální hodnota součtu. Pokud Q_m je desetinné číslo musí se použít nejbližší vyšší celočíselná hodnota.

Pro počet bitu zlomkové části Qn čísel FX16 platí vztah (1.10):

$$Qn = 16 - Qm - 1 \tag{1.10}$$

kde Qn je počet bitů zlomkové části a Qm je počet bitů celočíselné části.

2 NEURONOVÉ SÍTĚ

V reálném světě existují rozmanité metody pro řešení problémů, které s sebou nesou různé výhody a nevýhody. Pro účely této studie je klíčové rozlišovat mezi deterministickými a stochastickými algoritmy. Následující dvě kapitoly se věnují diskuzi o rozdílech mezi těmito dvěma přístupy a představují několik příkladů těchto metod. Významným zástupcem stochastických metod jsou neuronové sítě, které v posledních desetiletích procházejí prudkým vývojem a pro tuto práci mají zásadní význam.

Deterministický přístup
Tato metoda spočívá v konstrukci matematického modelu systému a následné validaci pomocí matematických důkazů. Validovaný model může být poté implementován ve formě mechanického, hydraulického, analogového či digitálního zařízení, nebo v současnosti jako program pro výpočetní systém. Tento model poté simuluje chování reálného systému v daném kontextu a poskytuje přesná řešení problémů. Výhodou deterministických algoritmů je skutečnost, pro stejné vstupní hodnoty vždy dostaneme identický výstup a jeho chování je známo. Příkladem deterministického algoritmu jsou například Dijikstrův, A*, Primův a další algoritmy.

Stochastický přístup
Druhým přístupem je stochastická metoda, která, na rozdíl od deterministické, nenabízí přesná, opakovatelná či optimální řešení. Tato nevýhoda je však vyvážena schopností poskytnout řešení v akceptovatelném časovém rámci, i když existuje deterministický model, nebo v případě, že jej v daném okamžiku nelze sestavit. Stochastická metoda tedy představuje pružnější řešení, které se hodí zejména pro situace, kdy je potřeba rychle reagovat na měnící se podmínky, nebo kdy je deterministický model obtížně realizovatelný.

Stochastické algoritmy jsou používány v oblasti optimalizace a klasifikace. Příkladem stochastických optimalizačních algoritmů jsou například Monte Carlo, Genetický, Evoluční, Random forest, Reinforcement Learning a další.

Další oblastí, je klasifikace jevů, kde opět s rostoucí dimenzionalitou prudce roste náročnost a navíc nemusí být znám exaktní vztah mezi jednotlivými parametry vstupního vektoru. Zde naleznou uplatnění algoritmy typu Bayesovy sítě, metoda nejliššího souseda, Random Forest a podobné.

Asi nejvýraznější oblastí použití je oblast počítačového vidění, rozpoznávání hlasu, analýza textu a jeho kontextu, predikce vývoje veličin v čase a také rychle se rozvíjející oblast generativních neuronových sítí, ať již v oblasti grafiky DALL·E [13], Stable Diffusion, videa, hudby, nebo konverzačních jazykových modelů, jako je ChatGPT 4 [14],

LLaMA [15], Bard [16] a další. Zde se uplatní různé varianty velmi komplexních neuronových sítí ve formě transformerů [19][17].

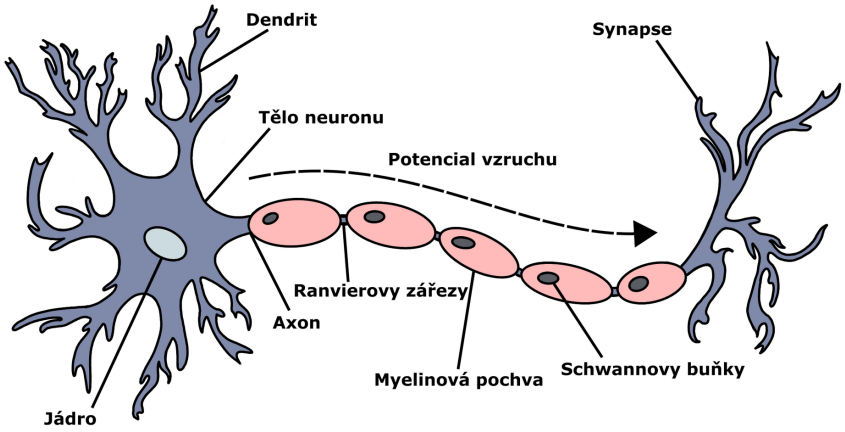
Rozvoj neuronových sítí se však neodehrává pouze v oblasti velkých modelů zmíněných v předchozím odstavci, ale také v oblasti nasazení v mikrokontrolerech, kde nalézají uplatnění v oblasti autonomních systémů v oblasti robotiky, bezpečnostních systému, předvídání budoucích poruch, zdravotnické technice a také v oblasti nositelné elektroniky a Internetu věcí (Internet of Thinks dále IoT) a mnoha dalších [18].

2.1 Perceptron

V této kapitole je popsán jeden z prvních prvků v oblasti neuronových sítí. Postupně je popsán základní princip, struktura a použití. Nedílnou součástí je také jeho učení.

2.1.1 Popis perceptronu

Nejjednodušším prvkem neuronových sítí je matematický model, který odpovídá jednomu neuronu. Tento model byl inspirován biologickými neurony obrázek 2.1, které se nacházejí v neuronových gangliích jednodušších, nebo mozcích pokročilejších živočichů.

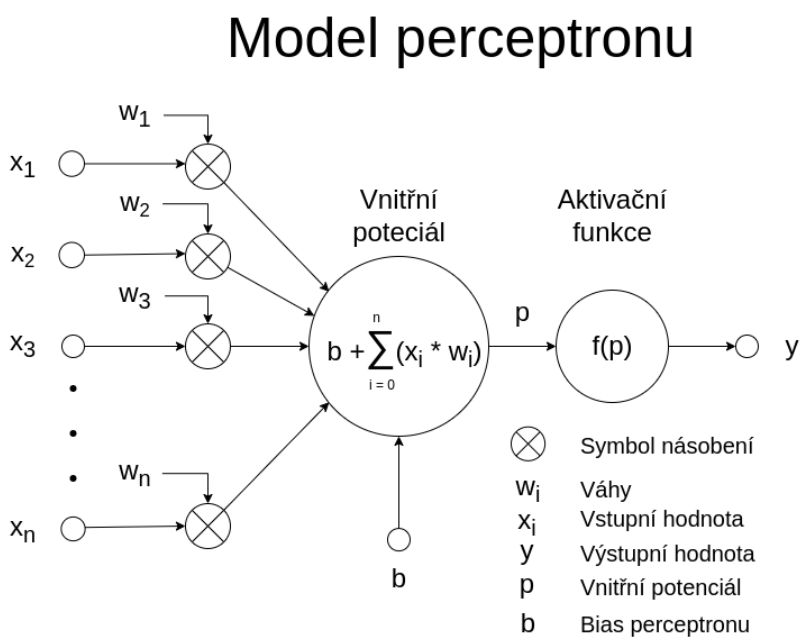


Obrázek 2.1. Biologický neuron

Tento model neuronu se nazývá Perceptron viz obrázek 2.2.

Vstup perceptronu tvoří vektor vstupních hodnot $x_1 ... x_n$ a jednoho výstupu y. Vstupní vektor je zpracován pomocí součtu součinů vstupních hodnot x_n s jim odpovídajícími váhami w_n. Tato část představuje takzvaný vnitřní potenciál neuronu. Matematicky je tedy vyjádřen vztahem (2.1):

$$p = b + \sum_{i=1}^{n} (x_i * w_i) \tag{2.1}$$



Obrázek 2.2. Model Perceptronu

kde p je vnitřní potenciál perceptronu, n je počet vstupů, x_i vstupní hodnota i-tého vstupu, w_i k němu příslušející váha a b je bias.

Vstup označený jako b a jemu odpovídající váha b je bias neuronu a posouvá práh aktivace u ostře nelineární aktivační funkce, nebo posouvá úroveň výstupu u nelineární aktivační funkce.

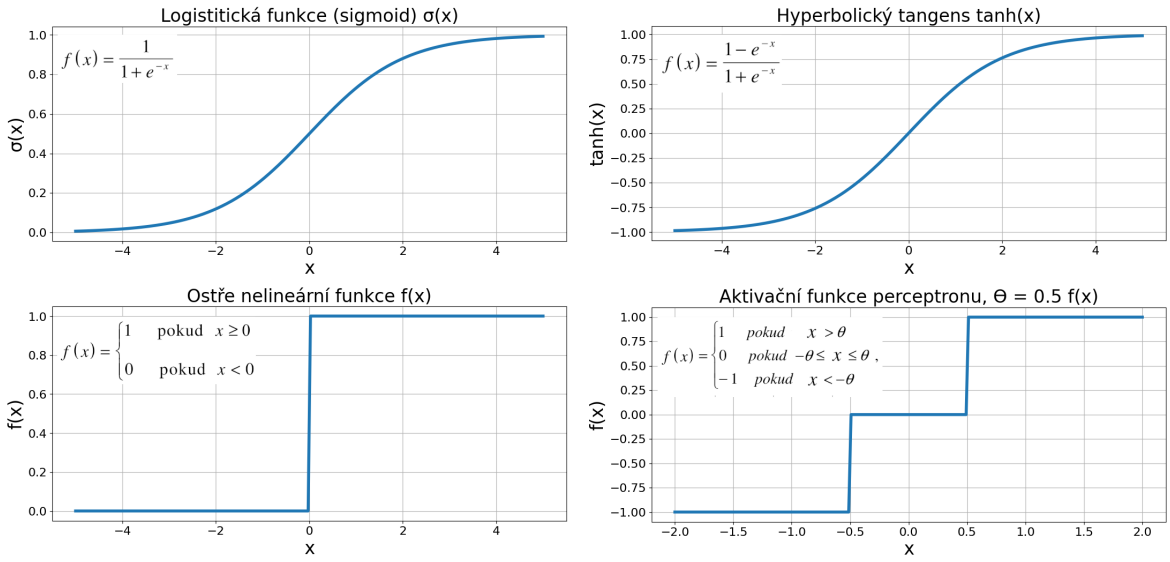
Další částí je aktivační funkce perceptronu. Tato funkce transformuje vnitřní potenciál na výstupní hodnotu pomocí aktivační funkce perceptronu. Pokud je funkce ostře nelineární, pak vyjadřuje, zda je perceptron aktivovaný, či nikoliv a vyjadřuje, zda vstupní hodnoty spadají do dané kategorie. V případě nelineární funkce, jako je *sigmoid*, *ReLu* a její varianty, nebo hyperbolický tangens *tanh*, pak vyjadřuje míru aktivace perceptronu a určuje tak míru příslušnosti k dané kategorii.

Některé vybrané funkce jsou zobrazeny na obrázku 2.3

Celkový vztah pro výpočet výstupní funkce je tedy uveden ve vztahu (2.2):

$$y = f(b + \sum_{i=1}^{n} (x_i * w_i)) \tag{2.2}$$

kde $f()$ je aktivační funkce d) dle obrázku 2.3, n je počet vstupů, x_i vstupní hodnota i-tého vstupu, w_i k němu příslušející váha a b je bias.



Obrázek 2.3. Vybrané aktivační funkce použité v neuronových sítích

Velmi podobný model perceptronu je Adaline (Adaptive Linear Neuron). Přijímá také bipolární bipolární signál na vstupu a také je často bipolární signál na výstupu. Hlavní rozdíl mezi nimi je aktivační funkce v neuronu. Zatímco perceptron má aktivační variantu d) dle obrázku 2.3, Adaline má lineární aktivační funkci [6] a je dána (2.3):

$$y = p \tag{2.3}$$

Z toho plyne, že celková přenosová funkce Adaline neuronu je dána vztahem (2.4):

$$y = b + \sum_{i=1}^{n} (x_i * w_i) \tag{2.4}$$

kde y je výstupní signál a p je vnitřní potenciál Adaline neuronu dle obrázku 2.2, n je počet vstupů, x_i vstupní hodnota i-tého vstupu, w_i k němu příslušející váha a b je bias.

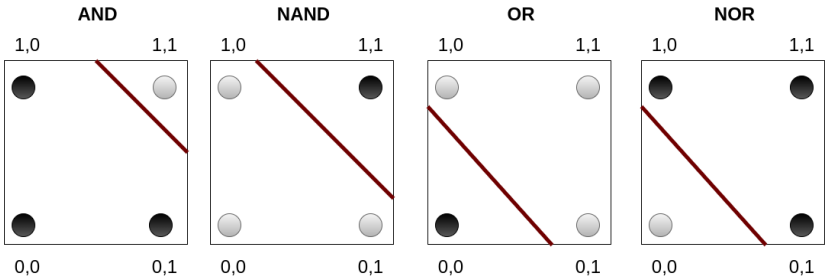
Perceptron a Adaline jsou klasifikátory, které jsou vhodné jen pro úlohy, které splňují podmínky lineární separability. Příkladem jsou logické funkce AND, NAND, OR a NOR. Zobrazení rozdělení na poloprostory zobrazuje obrázek 2.4.

Toto omezení je poněkud v praxi limitující, neboť většina reálných problémů tuto podmínku nesplňuje. Typickou úlohou, která je jedním perceptronem neřešitelná, je logická funkce XOR a XNOR a je zobrazena na obrázku 2.5.

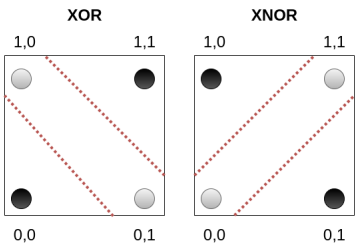
Mezi body nelze vést dělící rovinu tak, aby rozdělil prostor na dva poloprostory se správnou příslušností k daným třídám. Tento problém způsobil výrazné zpomalení vývoje neuronových sítí, neboť prostředky a pozornost byly přesunuty do jiných oblastí

Tabulka 2.1. Vybrané aktivační funkce používané v neuronových sítích

Aktivační funkce	$f(x)$	$f'(x)$
Logistická sigmoida	$\sigma(x) = \frac{1}{1+e^{-x}}$	$\sigma'(x) = f(x)(1 - f(x)$
Hyperbolický tangens	$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$\tanh'(x) = 1 - f(x)^2$
Ostrá nelinearita	$y = \begin{cases} 1 & \text{pokud } p > 0 \\ 0 & \text{pokud } p \leq 0 \end{cases}$	-
Aktivační funkce perceptronu	$y = \begin{cases} 1 & \text{pokud } p > \Theta \\ 0 & \text{pokud } -\Theta \leq p \leq \Theta \\ -1 & \text{pokud } p < \Theta \end{cases}$	-



Obrázek 2.4. Lineárně separabilní funkce AND,NAND, OR a NOR jedním perceptronem



Obrázek 2.5. Lineárně neseparabilní funkce XOR a XNOR jedním perceptronem

strojového učení.

2.1.2 Trénování perceptronu

Postup adaptace perceptronu[20]:

1. Inicializace vah $w_i \quad i = (1...n)$ náhodnými hodnotami.
   Přiřazení inicializační hodnoty koeficientu učení $\eta(0 < \eta < 1)$

2. - Pro každý tréninkový pár $\vec{\mathbf{d}} : t$ z datasetu
     aktivovat vstupní neurony $x_i = d_i$
   - Výpočet vnitřního potenciálu perceptronu:
     $p = b + \sum_{i=1}^{n} (x_i * w_i)$
   - Následně výpočet výstupní hodnoty:
     $$y = \begin{cases} 1 & \text{pokud } p > \Theta \\ 0 & \text{pokud } -\Theta \leq p \leq \Theta \\ -1 & \text{pokud } p < \Theta \end{cases}$$

3. Pokud je $y \neq t$ aktualizace hodnot vah a biasu
   $w_i(m+1) = w_i(m) + \eta t x_i \quad i = (1...n)$
   $b(m+1) = b(m) + \eta t$
   jinak se hodnoty vah nemění
   $w_i(m+1) = w_i(m)$
   $b(m+1) = b(m)$

4. Pokud se hodnota vah a biasu nemění, nebo trénovací dataset je vyčerpán, pak konec učení.
   Jinak pokračování krokem 2.

kde $\vec{\mathbf{d}} : t$ je datový trénovací pár tvořený vstupním vektorem s dimenzionalitou odpovídající vstupním datům ($\vec{\mathbf{d}}$) a výstupní požadovanou skalární hodnotou, neboť perceptron má pouze jeden výstup (t), w_i je váhový vektor, y je výstup neuronu, a p je vnitřní potenciál neuronu, b je bias perceptronu.

2.1.3 Trénování Adaline neuronové sítě

Postup trénování je stejný jako u perceptronu: každou váhu w_i je třeba aktualizovat tak, aby v následující iteraci zvýšila množství správně předpovězených výstupů - tuto hodnotu aktualizace nazýváme δ_i. Tato aktualizační veličina se však počítá jiným způsobem, a to pomocí algoritmu známého jako gradientní sestup (Gradient descent)[6].

Gradientní sestup je optimalizační algoritmus, který hledá globální minimum derivace chybové funkce pomocí nastavení parametrů vah w_i, které jsou na počátku učení

nastaveny na náhodné hodnoty. Tento proces probíhá iterativním způsobem pomocí trénovacích dat, kde se porovnává požadovaný výstupní signál se skutečně obdrženým. Rozdíl mezi těmito veličinami se následně aplikuje na váhy neuronu. Proces učení se zastaví v okamžiku, kdy dalším učením nedojde ke zlepšení a bylo dosaženo minima chybové funkce. Postup adaptace sítě Adaline:

1. Inicializace vah $w_i \quad i = (1...n)$ náhodnými hodnotami.
   Přiřazení inicializační hodnoty koeficientu učení $\eta(0 < \eta < 1)$
2. - Pro každý tréninkový pár $\vec{\mathbf{d}} : t$ z datasetu
     aktivovat vstupní neurony$x_i = d_i$
   - výpočet vnitřního potenciálu perceptronu:
     $p = b + \sum_{i=1}^{n} (x_i * w_i)$
   - následně výpočet výstupní hodnoty:
     $y = p$
3. Pokud je $y \neq t$ aktualizace hodnot vah a biasu
   $w_i(m+1) = w_i(m) + \eta(t-p)x_i \quad i = (1...n)$
   $b(m+1) = b(m) + \eta(t-p)$
4. Pokud je změna hodnot vah a biasu menší, než maximální povolená chyba, nebo trénovací dataset je vyčerpán, pak konec učení.
   Jinak pokračování krokem 2.

kde $\vec{\mathbf{d}} : t$ je datový trénovací pár tvořený vstupním vektorem s dimenzionalitou odpovídající vstupním datům ($\vec{\mathbf{d}}$) a výstupní požadovanou skalární hodnotou, neboť neuron má pouze jeden výstup (t), w_i je váhový vektor, y je výstup neuronu, a p je vnitřní potenciál neuronu.

V případě Adaline je jako chybová funkce použita střední kvadratická chyba MSE nebo varianta RMSE dle vztahu (2.5):

$$t = \sum_{i=1}^{n} (x_i * w_i)^2 \tag{2.5}$$

Proces učení ovšem nemusí být vždy úspěšný a to v případě, že funkce není lineárně separabilní. V tomto případě je nutno použít neuronovou síť složenou z více neuronů ve více vrstvách, případně neurony vyšších řádů.

2.2 Feedforward neuronová síť s backpropagation učením

Jde o typ neuronové sítě s dopředným šířením signálu. Její schéma je na obrázku 2.7. Oproti síti Adaline má tato síť více vrstev s více neurony. Feedforward síť je tvořená třemi hlavními částmi:

Vstupní vrstva

Je část sítě, do které vstupují informace, které chceme zpracovat. Počet těchto vstupních hodnot je dán podobně, jako u Perceptronu dimenzionalitou vstupních dat. Tato vrstva většinou převádí vstupní hodnoty do další vrstvy. Někdy může být také označena za neurony se specifickou funkcí. Tato vrstva se neúčastní samotného učení a pokud má váhy, tak plní roli preprocesoru dat, například normalizace, škálování, transformace a podobně.

Skryté vrstvy

Skrytá vrstva může být jedna, nebo jich je více. Obvykle mohou mít každá jiný počet neuronů v závislosti na požadované komplexitě rozeznávaných vzorů. Tato část se účastní procesu učení a hraje při něm důležitou roli. Každý vstup neuronu je obvykle napojený na každý výstup předchozí vrstvy neuronové sítě a každý vstup má také své váhy w.

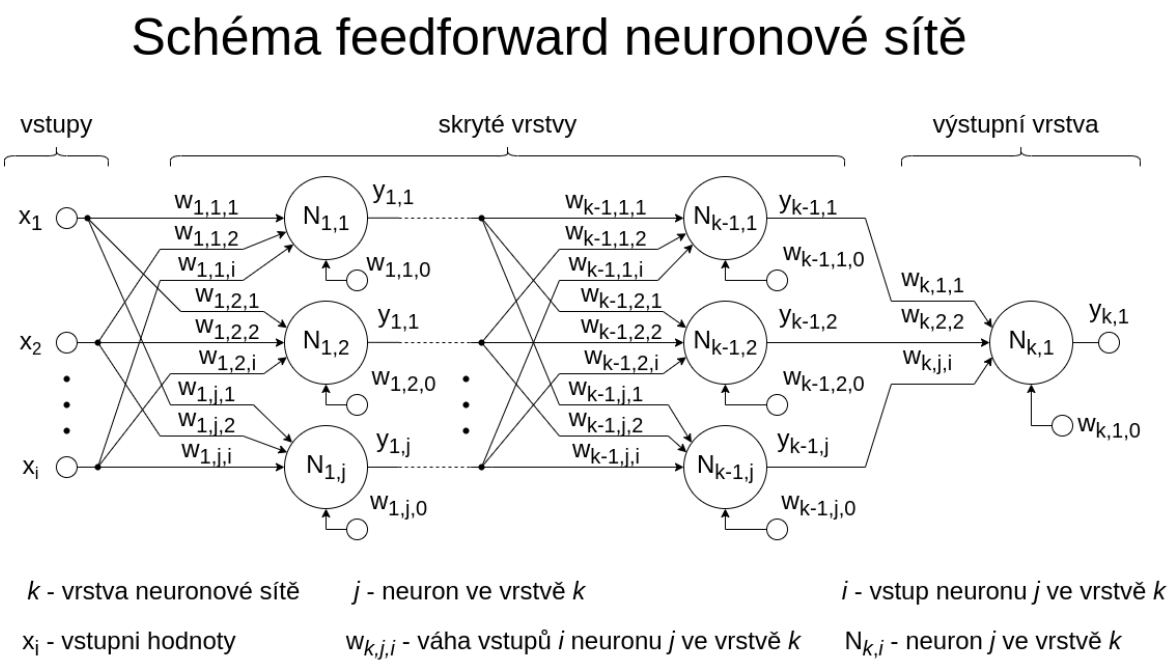
Výstupní vrstva

Výstupní vrstva převádí výstupy předchozích vrstev na požadovaný výstup může mít stejnou aktivační funkci jako předchozí vrstva pro účely regrese, ne s výhodou použije funkci Softmax pro klasifikaci vstupních dat. Účastní se procesu učení.

2.2.1 Učící dataset

Pro učení neuronové sítě je nutno použít dataset s označenými vzory. To znamená, že data obsahují vstupní a požadované výstupní hodnoty. U datasetu pro učení je důležité, aby data skutečně reprezentovala vlastnosti, které má naučená neuronová síť zpracovávat. Neméně důležité je příslušné vzory označit tak, aby byly správně zařazeny do příslušných kategorií. Získání datasetu může být nejsložitějším úkolem, neboť je třeba brát v úvahu nejen správnost dat, ale také etické a právní okolnosti jejich získání a týká se to zejména osobních dat lidí a firem a v případě textových, obrazových a zvukových dat, také práva autorská. Datasety je také nutno chránit před zneužitím.

Správně sestavený dataset, spolu se správně označenými kategoriemi, je pro účely učení rozdělen na několik stejných částí. Jedna část bude sloužit jako validační pro ověření kvality výsledného naučení neuronové sítě. Kromě rozdělení do bloků jsou data v jejich rámci náhodně promíchána. Tímto způsobem snížíme riziko přeučení.



Obrázek 2.6. Feedforward neuronová síť

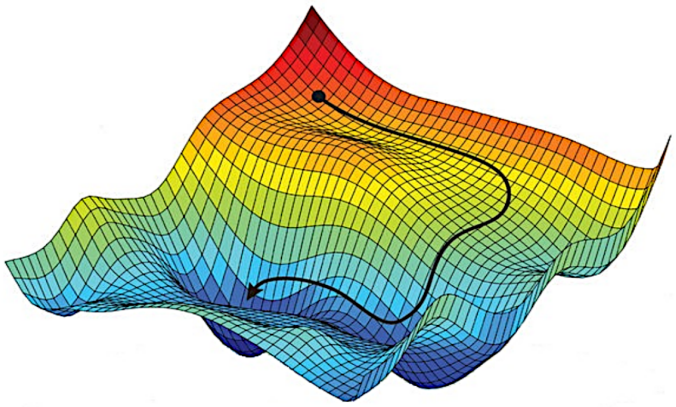
2.2.2 Algoritmus učení backpropagation

Backpropagation, neboli zpětné šíření chyby (gradient descent), je učící algoritmus s učitelem pro adaptaci neuronové sítě tak, aby plnila požadovanou činnost, ať už regresní, nebo klasifikační v souladu s požadavky.

Princip spočívá v porovnání výstupní hodnoty neuronové sítě pro daný vstupní vektor dat s požadovanou hodnotou. V případě, že se výstup neliší, cyklus učení pokračuje s dalším vstupním vektorem. Pokud se ovšem liší, výsledný rozdíl se postupně propaguje zpět do první skryté vrstvy, a ovlivňuje váhy nejen u vstupních dat, ale i u biasu jednotlivých neuronů.

V kontextu parciálních derivací, celková derivace chyby je rozdělena na parciální derivace, které následně ovlivňují jednotlivé váhy vstupů u všech neuronů ve všech vrstvách. Parciální derivace v tomto kontextu reprezentují měřítko, jak moc se chyba změní, když se mírně změní daná váha. Při procesu učení se tyto derivace využívají k tomu, aby se váhy postupně upravovaly směrem, který nejvíce snižuje chybu. Tento proces zajišťuje, že každý neuron je přizpůsoben tak, aby minimalizoval chybu v předpovědích sítě v souvislosti s jeho specifickými vstupy a váhami.

1. Inicializace vah $w_{k,j,i}$
   kde $k = (1...m)$ je vrstva neuronové sítě, $j = (1...n)$ je neuron ve vrstvě a $i = (1...o)$ je vstup neuronu i náhodnými hodnotami.
   Přiřazení inicializační hodnoty koeficientu učení (momentum) $\eta(0 < \eta < 1)$



Obrázek 2.7. Gradientní sestup (Gradient descent)[21]

Feedforward:

2.
- Pro každý tréninkový pár $\vec{\mathbf{d}} : \vec{\mathbf{t}}$ z datasetu
  aktivovat vstupní neurony $y_{0,i} = x_i = d_i$, které jen kopírují hodnotu ze vstupu na výstup.
- Výpočet vnitřního potenciálu neuronů ve skryté vrstvě:
  $p_{k,j} = b_{k,j} + \sum_{i=1}^{n} (y_{(k-1),j,i} * w_{k,j,i})$
  kde k je index skryté vrstvy, j je index neuronu a i je index vstupu neuronu.
- Následně výpočet výstupní hodnoty pomocí aktivační funkce $f(p_{k,j})$:
  $y_{k,j} = f(p_{k,j})$.

Backpropagation:

3. Výstupní vrstva:

- Každému $y_{k,j}$ přiřadíme t_j kde k je poslední vrstva v neuronové síti.
- Spočítat chybu $\delta_j = (t_j - y_{k,j}) f'(p_{k,j})$
- Následně váhové korekce $\Delta w_{k,j,i} = \eta \delta_j y_{k-1,j}$.

Skryté vrstvy:

- $\delta_j = f'(p_{k,j} \sum_{i=1}^{n} \delta_j w_{k,j,i}$
- $\Delta w_{k,j,i} = \eta \delta_j y_{k-1,j}$

Aktualizace vah a prahů:

4.
- Každý neuron j ve vrstvě k aktualizuje váhy na vstupech i.
- $w_{k,j,i}(\text{new}) = w_{k,j,i}(\text{old}) + \Delta w_{k,j,i}$

5. Pokud se hodnota vah a biasu nemění, nebo trénovací dataset je vyčerpán, pak konec učení.
   Jinak pokračování krokem 2.

kde $\vec{\mathbf{d}} : \vec{\mathbf{t}}$ je datový trénovací pár tvořený vstupním vektorem s dimenzionalitou odpovídající vstupním datům ($\vec{\mathbf{d}}$) a v tomto případě výstupním vektorem s požadovaným zařazením do příslušné třídy, nebo hodnotami ($\vec{\mathbf{t}}$). Jeho velikost odpovídá počtu kategorií a současně i počtu výstupních neuronů u sítě typu fastforward. $w_{k,j,i}$ je váhová matice, případně vektor, pokud byla na něj transformována, $y_{k,j}$ jsou výstupy příslušných neuronů, a $p_{k,j}$ je vnitřní potenciál neuronu.

2.2.3 Přeučení neuronové sítě

Přeučení je stav, kdy neuronová síť vykazuje minimální, nebo dokonce nulovou chybovost na učících datech, ale na testovacích datech se následně chybovost prudce zvětší. Takto přeučená NS ztratila schopnost generalizace.

Pokud k tomuto jevu dojde, bude nutno učení NS provést znovu, a to s nově promíchaným a rozděleným datasetem a také s nově nastavenými náhodnými váhami u neuronů. Není vhodné je pro jednoduchost nastavovat na identické váhy, protože to s velkou pravděpodobností může vést k selhání učení.

2.2.4 Význam funkce jednotlivých vrstev

Jednotlivé vrstvy neuronové sítě feedforward plní specifickou funkci. Pro další popis je definována struktura sítě typu [2, 2, 2, 1] se dvěma vstupními parametry, dvěma skrytými vrstvami, každá po dvou neuronech a výstupní vrstva s jedním neuronem. Na obrázku 2.8 jsou zobrazeny separační hyperplochy jednotlivých vrstev Feedforward neuronové sítě.

Grafy pro ilustraci byly vytvořeny pomocí programu Wolfram Matematica na základě testu feedforward neuronové sítě s využitím datasetu Occupancy Detection Data Set [24]. Její pracovní list je uveden v přiložených souborech. Výsledek učení je v tabulce 2.2.

Funkcionalita jednotlivých vrstev FF

První skrytá vrstva obrázek 2.8a Každý neuron v první skryté vrstvě dělí prostor na dva pomocí separačních hyperploch. Vzhledem k tomu, že první skrytá vrstva obsahuje dva neurony, je prostor rozdělen na čtyři poloprostory. Každý z těchto poloprostorů sdružuje body s podobnými vlastnostmi, které náleží do stejné kategorie.

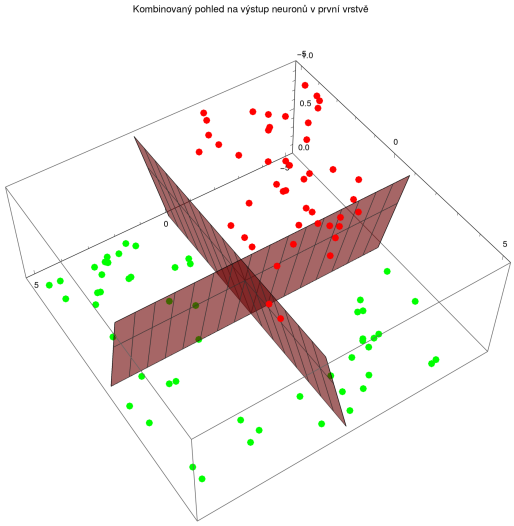
Tabulka 2.2. Výsledky učení FF sítě na testovacím datasetu

Validate al weights on test dataset	Count	Percent
Test dataset size	20	-
Correct results	20	100
Wrong results	0	0

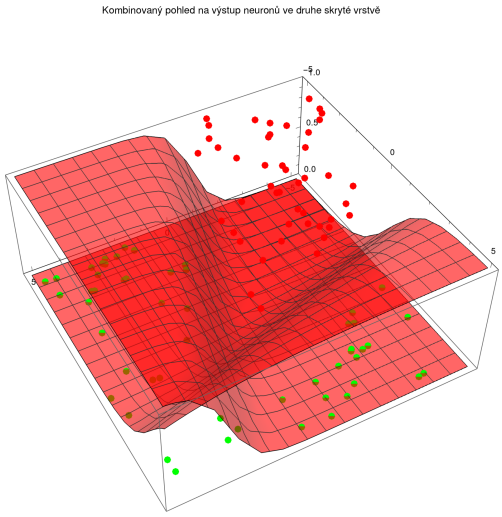
První skrytá vrstva obrázek 2.8b Druhá skrytá vrstva se opět snaží rozdělit prostor pomocí hyperploch na dva poloprostory, avšak s vyšší úrovní abstrakce, která se zaměřuje na komplexnější vztahy mezi body v prostoru. Tím pádem se druhá skrytá vrstva pokouší separovat jednotlivé body efektivněji než první vrstva.

Výstupní vrstva obrázek 2.8c Výstupní vrstva následně kombinuje výsledky z druhé skryté vrstvy a vytváří konečné rozdělení prostoru pomocí komplexní separační hyperplochy s další přidanou abstrakcí. Toto rozdělení vede k vytvoření dvou poloprostorů, ve kterých body jedné kategorie leží nad komplexní hyperplochou, zatímco body druhé kategorie leží pod ní.

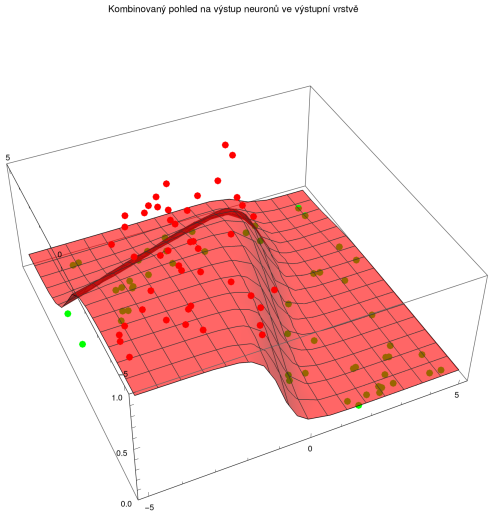
Sloučené hyperplochy všech vrstev Vzájemný vztah dělících hyperploch jednotlivých vrstev demonstruje poslední obrázek 2.8d, ve kterém jsou sdruženy.



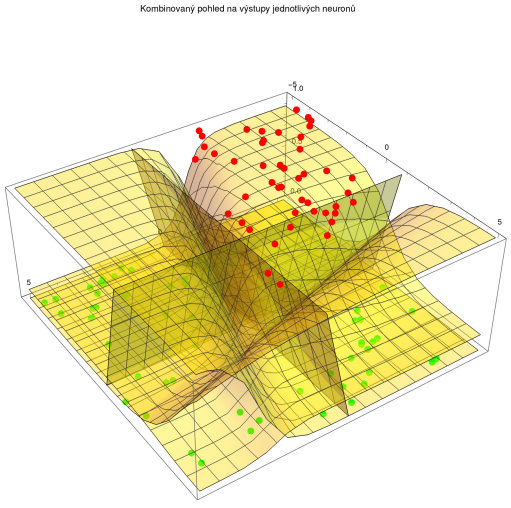
(a) První vrstva



(b) Druhá vrstva



(c) Výstupní vrstva



(d) Sloučené hyperplochy všech vrstev

Obrázek 2.8. Separační hyperplochy vícevrstvé neuronové sítě

2.3 Konvoluční neuronové sítě

Následuje popis konvolučních sítí, které jsou základem pro oblast strojového vidění, klasifikace obrazu a rozpoznávání zvuku.

2.3.1 Historie vzniku Konvolučních neuronových sítí

V oblasti poštovních služeb se v důsledku nárůstu korespondence objevila potřeba automatizace třídění zásilek. Tento proces byl ztížen skutečností, že adresy a směrovací čísla na zásilkách byly často psány ručně. Proto bylo nezbytné vyvinout systém pro automatické rozpoznávání ručně psaných písmen a čísel, které tvoří doručovací adresy.

Jedním z průkopníků v této oblasti byla síť Neocognitron od Kunihiko Fukushimy, která byla představena v roce 1980 a sloužila k rozpoznávání ručně psaného japonského písma a jiných vzorů [11] [10]. Neocognitron byl inspirován prací Davida H. Hubela a Torstena N. Wiesela, kteří popsali strukturu oblasti mozkové kůry, kde jednotlivé neurony, za předpokladu nehybnosti očí, reagují na malé oblasti zorného pole [12].

Tato oblast se nazývá receptivní pole neuronu a zároveň se překrývá se sousedními neurony. Velikost a umístění receptivního pole se neustále mění. V roce 1968 byly identifikovány dva typy vizuálních neuronů. [11].

Jednoduché buňky (S buňky) jsou excitovány hranami s určitou orientací v jejich receptivním poli.

Komplexní buňky (C buňky) mají širší receptivní pole a nejsou citlivé na přesnou polohu hran.

Tyto dva druhy buněk lze střídavě uspořádat za sebou, což umožňuje zpracování rozsáhlejších abstrakcí ve vstupních obrazcích.

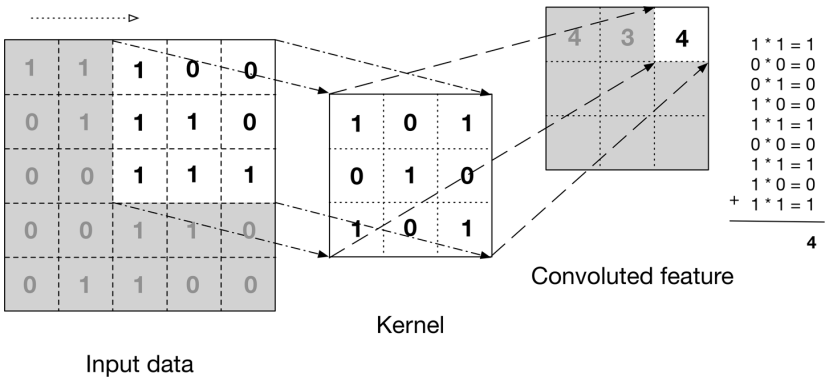
Konvoluční neuronové sítě (CNN) byly uvedeny roce 1990 pod původním názvem Space Invariant Artificial Neural Networks (SIANN) a byly velmi úspěšné v klasifikaci objektů v obrazových datech, kde vykazovaly nezávislost na poloze a úhlu pohledu klasifikovaných objektů[7].

2.3.2 Popis funkce CNN

Při zpracování velkého množství vstupních dat, například obrazových, velmi rychle roste nutný počet neuronů nejen ve vstupní vrstvě, ale i ve skrytých vrstvách. Například vstupní obrázek o rozměru 25 x 25 pixelů s jedním jasovým kanálem představuje vstupní vektor o velikosti 625 hodnot. Pro skrytou vrstvu o velikosti 128 neuronů to představuje již 80 000 vah. V poslední vrstvě, kde je jen 10 výstupních neuronů, například pro

klasifikaci ručně psaných čísel, je nutno přidat dalších 1 280 vah. Pro takto malý obrázek tedy potřebujeme pro naučenou neuronovou síť spočítat 81 280 vah a stejné množství je nutno určit při učení. Takto vytvořená síť je velmi výpočetně i paměťově náročná nejen pro proces učení, ale i pro samotnou, již naučenou síť.

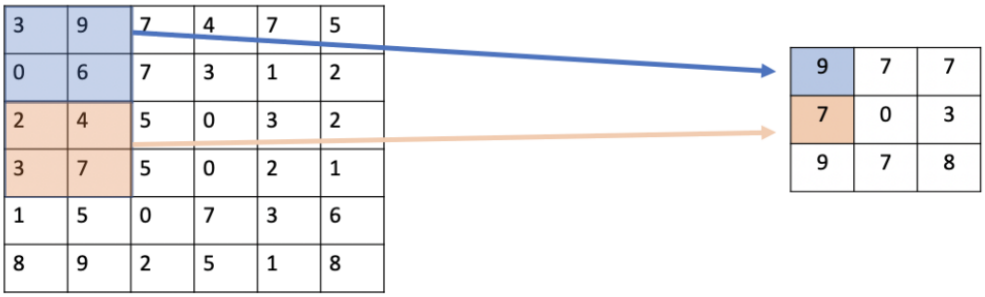
Vstupní vrstvu tvoří obrazová data o velikosti $m \times n$. Tato vrstva může mít jeden nebo více kanálů. Na tuto vrstvu navazuje první konvoluční vrstva, která primárně rozeznává hrany a jejich orientaci. Tato konvoluční vrstva je tvořena takzvaným konvolučním jádrem. Toto jádro má většinou velikost 3×3, 5×5 a 9×9. Toto jádro obsahuje parametry, které představují váhy podobně, jako u neuronových sítí. Toto jádro se postupně posouvá po ploše s posuvem (stride) 1 a více. Pokud je posunutí větší než jedna, dojde současně ke zmenšení výstupní vrstvy. Toto konvoluční jádro není v dané vrstvě jediné a tvoří takzvanou sadu filtrů. Výsledkem je tedy nová vrstva s menší šířkou a výškou, která je dána velikosti posunutí a současně okrajem. Hloubka této vrstvy je dána počtem použitých filtrů. Každý filtr se zaměřuje na jiný typ hran a s růstem jejich počtu se zlepšuje množství zachycených vzorů. Na druhou stranu ovšem hrozí riziko přeučení.



Obrázek 2.9. Konvoluce s jádrem 3×3 [22]

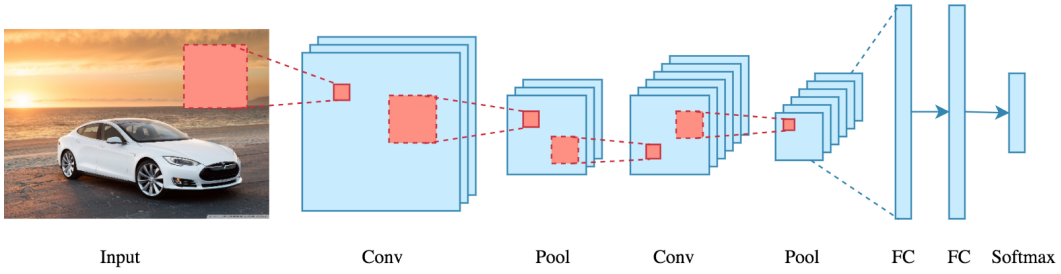
Následující vrstvu může tvořit pooling vrstva, jejíž cílem je zredukovat dimenzionalitu vstupních dat. Tato vrstva tvoří jádro o velikosti minimálně 2×2 a posun 2. V případě užití max pooling vrstvy se tedy vybere z pole 2×2 nejvyšší hodnota a ta se vloží do nové vrstvy a velikost vstupních dat se tedy sníží u každého filtru na polovinu. Dle rozměrů vstupních obrazových dat lze samozřejmě zvolit posun i velikost pooling jádra větší, nicméně pokud je příliš velké, dojde ke ztrátě informace a tím, k možnému snížení přesnosti odhadů sítě.

Další možnou variantou místo funkce max pooling je average pooling vrstva, která předá střední hodnotu poolingového jádra. Častěji se používá varianta max pooling, neboť má lepší výsledky.



Obrázek 2.10. Max pooling vrstva 2×2 s posunutím 2 [23]

Tato dvojice konvoluční-pooling vrstvy může být řazena za sebou podle požadavků na počtu výstupních tříd. Po poslední vrstvě následuje plně propojená neuronová síť typu feedforward, která nalezené abstrakce vzorů zpracovává a následně pomocí poslední vrstvy s softmax aktivační funkcí rozdělí do jednotlivých tříd.



Obrázek 2.11. Struktura CNN [22]

Učení CNN probíhá obdobně, jako u sítě typu FF, nicméně váhy konvolučních jader jsou pro daný filtr a vrstvu identické a na počátku učení náhodné. Tato vlastnost výrazně snižuje potřebný výkon při učení, protože místo stovek až deseti tisíců vah stačí učením určit jen tolik, kolik jich obsahuje konvoluční jádro, což znamená pro rozměr 3×3 a 10 filtrech je to jen 90 vah pro celou vstupní vrstvu s jedním kanálem.

Proces učení konvoluční neuronové sítě CNN začíná s přípravou datasetu podobně jako u vícevrstvých dopředných neuronových sítí. Avšak tentokrát je dataset tvořen obrazovými daty pro dvourozměrné sítě, nebo například zvukovými záznamy pro jednorozměrné sítě. Stejně jako u feedforward sítě, je to typ učení s učitelem, a proto je nutné data přiřadit k odpovídajícím třídám.

Proces začíná tím, že se načte vzorek dat a zpracuje se pomocí CNN. Výstup klasifikace se poté porovná s očekávanou hodnotou. Pokud je zjištěn rozdíl, následuje zpětná propagace chyby, která se přenese přes plně propojenou neuronovou síť, všechna konvoluční jádra jednotlivých filtrů a všechny vrstvy, případně kanály, pokud se zpracovávají barevné obrázky.

Váhy se poté upraví pomocí metody gradientního sestupu, která se zaměřuje na ty, které se nejvíce podílely na chybě. Tento cyklus se opakuje pro celý dataset určený pro učení. Pokud nebylo dosaženo minimální chyby nebo, nebyl proveden požadovaný počet iterací, dataset se promíchá a rozdělí znovu.

Po dokončení učícího procesu se na validačním datasetu testuje kvalita učení. Pokud CNN vykazuje vysoké procento chyb, je nutné zjistit, zda konfigurace sítě odpovídá požadovanému úkolu. Příčinou chyb může být nedostatečný počet filtrů, které jsou schopné abstrahovat vzory ze vstupních dat, nebo naopak, síť může být příliš složitá a ztratila schopnost generalizovat.

Kvalita datasetu pro učení je klíčovou součástí jakékoliv neuronové sítě, jak již bylo zmíněno v kapitole věnované dopředným sítím (FF). Je také nezbytné zajistit ochranu proti neoprávněným, poškozeným nebo podvrženým datům, pokud zdrojová data pocházejí od třetích stran, jako jsou jednotlivci, organizace, nebo jsou získána z internetu. To je důležité, aby se zabránilo vytvoření skryté zranitelnosti v konvoluční neuronové síti (CNN) během procesu učení. Tento druh datasetu může být vytvořen úmyslně, a v takovém případě se jedná o takzvaný „Data Poisoning Attacks“ [25]. Současná praxe hledá a analyzuje metody pro detekci a možnosti obrany proti těmto druhům útoků nejen při učení, ale také při samotném nasazení CNN.

3 ARCHITEKTURA A POPIS MIKROKONTROLERŮ S JÁDRY ARM CORTEX ŘADY M

Akronym ARM byl poprvé použit v roce 1986 společností Acorn Computers v souvislosti s procesorem osobního počítače Acorn Archimedes, kde značil "Acorn RISC Machine"[27]. Dne 27. listopadu 1990 byla založena firma Advanced RISC Machines Ltd., což způsobilo změnu významu akronymu ARM na „Advanced RISC Machine“. V roce 1998 došlo k další změně názvu společnosti na ARM Holdings, a to v souvislosti s IPO (první veřejnou nabídkou akcií). V současné době je společnost známá pouze jako ARM, přičemž došlo také ke změně loga společnosti a názvu z velkých písmen zkratky na malá písmena [26].

Společnost ARM, která je zodpovědná za architekturu se stejným názvem, neprodukuje finální výrobky. Místo toho licencuje své návrhy výrobcům, kteří mohou po získání licence tyto návrhy použít přímo nebo upravit podle svých potřeb (jako například Qualcomm, Apple či Samsung) pro finální produkty [28]. ARM jádra pokrývají široké spektrum aplikací, od malých a levných MCU s jádry Cortex-M0/M0+, která postupně nahrazují 8- a 16-bitové procesory, přes jádra optimalizovaná pro reálný čas Cortex-R82 až po výkonná jádra Cortex-X2 a Cortex-A715.

V aplikačních oblastech se jedná o produkty, které zahrnují malé mikrokontrolery pro řízení zařízení (řady Cortex M a R), vícejádrové procesory pro mobilní telefony a tablety (jako Qualcomm Snapdragon 8 generace 2 či Apple A16), mobilní a stolní počítače (Apple M1 a M2), servery (Graviton, Neoverse V2) až po superpočítače (Fujitsu A64FX, Graviton 3).

Jak akronym ARM naznačuje, architektura instrukčních sad (ISA) jader Cortex je založena na principu RISC (Reduced Instruction Set Computer), což je protějšek CISC (Complex Instruction Set Computer). Tyto dvě kategorie ISA představují rozdílné filozofie návrhu, které ovlivňují výkon a efektivitu procesorů. RISC architektura staví na jednoduchých, rychle prováděných instrukcích, které často vyžadují menší počet taktů hodin. Na druhou stranu, CISC architektura zahrnuje komplexní instrukce, které mohou být prováděny za více taktů hodin, avšak poskytují vyšší úroveň abstrakce pro programátory.

Vzhledem k tomu, že RISC procesory pracují s jednoduchými instrukcemi, obvykle mají menší počet tranzistorů, což může vést k nižší spotřebě energie a vyšší rychlosti zpracování. Naopak, CISC procesory mohou obsahovat více tranzistorů a hardwarové

logiky pro provádění složitějších operací [42].

Mezi instrukčními sadami RISC a CISC neexistuje striktně vymezená hranice. CISC neprovádí implementaci kompletní instrukční sady v hardwaru, ale částečně nebo zcela využívá mikroinstrukce. Tento přístup může být motivován snahou o snížení počtu tranzistorů, zmenšení plochy čipu, snížení spotřeby energie nebo nutností emulovat instrukční sadu CISC z důvodu kompatibility s jinými sadami instrukcí.

Významnou výhodou tohoto přístupu je možnost modifikace, zlepšení a oprav implementované funkcionality v průběhu životnosti procesoru prostřednictvím změny mikrokódu. Oprava chyb v hardwarovém návrhu je totiž nákladná a může vést k finančním ztrátám, způsobeným například stažením vadné výrobní série od zákazníků. Na druhou stranu, procesory realizované tímto způsobem mohou trpět vyšší latencí a celkově nižším výkonem. Tuto nevýhodu lze částečně eliminovat optimalizací architektury [41].

Hlavními zástupci RISC architektury jsou v současnosti PowerPC(IBM), ARM(ARM), RISC-V(Open Hardware), MIPS(MIPS), AVR(Atmel/Microchip).

3.1 Mikrokontroler RP2040

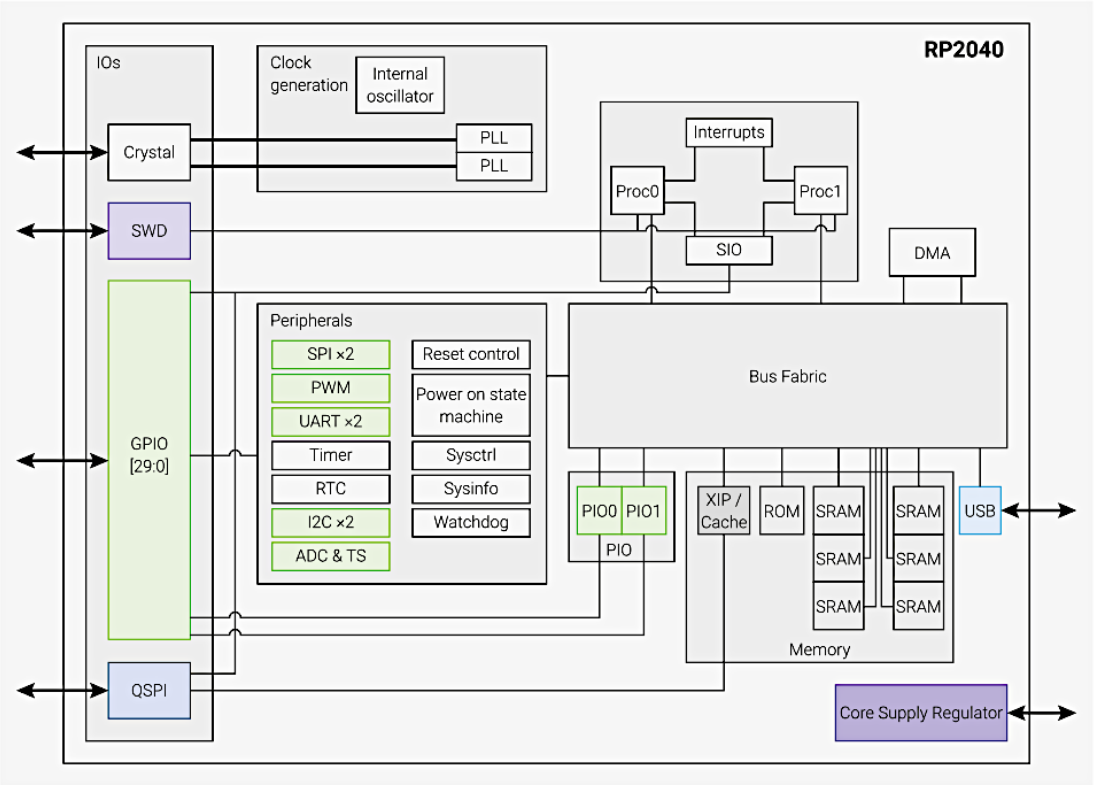
Mikrokontroler Raspberry RP2040 je prvním a zatím jediným mikrokontrolerem navrženým společností Raspberry [29]. Kromě tohoto typu mikrokontroleru firma nabízí vývojové a výpočetní moduly založené na mikroprocesorech od společnosti Broadcom. Jedná se o BCM2711 s čtyřmi jádry ARM A72 o taktu 1,5 GHz a pamětí od 1 GiB do 8 GiB, které se prodávají pod názvem Raspberry Pi 4, Pi 4 CM [31][32]. Dále nabízí Pi Zero 2 W s procesorem Broadcom BCMBCM2710A1, který obsahuje čtyři jádra Cortex-A53 [33].

RP2040 je k dispozici jak samostatně ve formě čipu v pouzdru QNF-56, tak i jako součást modulu (SoM) Raspberry Pico, který lze přímo instalovat do cílového zařízení, ať už doplněním pinů a vložením do patice, nebo přímo připájením na cílovou desku. Tento modul je k dispozici ve čtyřech variantách, a to s WiFi a Bluetooth modulem (BT) nebo bez bezdrátových modulů. Obě varianty jsou navíc dodávány s připájenými piny pro instalaci do kompatibilní patice, nebo bez pinů pro pájení přímo na základní desku platformy.

Samotný mikrokontroler je vybaven dvěma jádry ARM M0+ s taktovací frekvencí 133 MHz bez FP jednotky. Nicméně, každé jádro je vybaveno hardwarovou celočíselnou

děličkou, programovatelným konečným automatem (FSM) pro akceleraci vstupně/výstupních operací a dvěma celočíselnými interpolátory pro každé jádro.

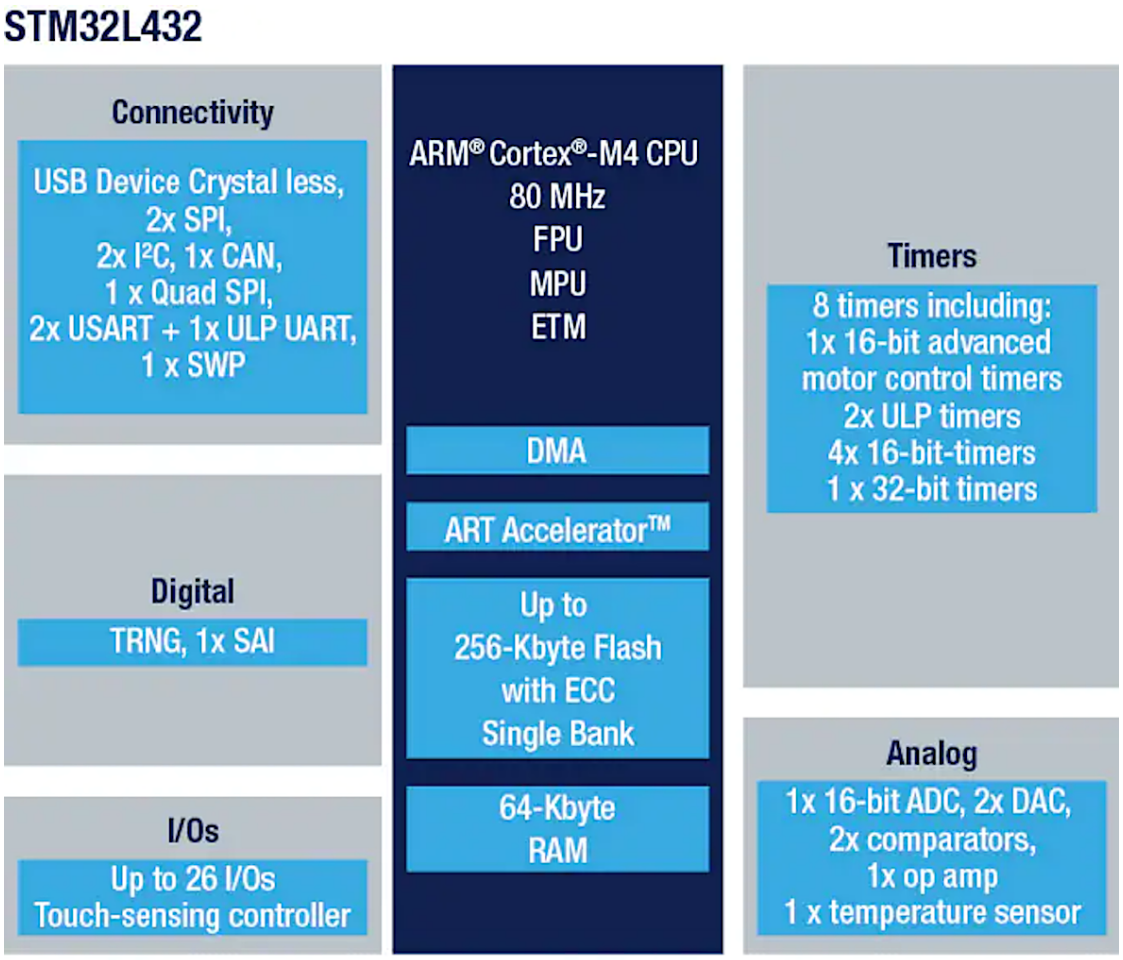
Paměť mikrokontroleru je trojího typu. Prvním typem je integrovaná RAM o velikosti 264 kiB, která je rozdělená do šesti bank. Druhým typem paměti je externí sériová paměť Flash připojená přes čtyřkanálové rozhraní QSPI s kapacitou až 16 MiB. Posledním typem je vestavěná ROM, která obsahuje bootloader a emulátor USB diskové jednotky pro snadné vložení programu ve formátu UF2.



Obrázek 3.1. Blokové schéma mikrokontroleru RP2040 [30]

3.2 Mikrokontroler STM32L432KC

Mikrokontroler STM32L432KC od společnosti STMicroelectronics je obvod s nízkou spotřebou a je součástí vývojové platformy NUCLEO-L432KC [34]. Jedná se o jednojádrový SoC s jádrem ARM Cortex M4F a maximální taktovací frekvencí 80 MHz. Tento mikrokontroler je vybaven FPU s jednoduchou přesností, ART akcelerátorem pro rychlý přístup do paměti FLASH, kompletním sadou DSP instrukcí a MMU jednotkou pro správu paměti. Jádro komunikuje s integrovanými pamětí typu Flash o velikosti 256 KiB a dále 64 KiB RAM.



Obrázek 3.2. Blokové schéma mikrokontroleru STM32L432KC [34]

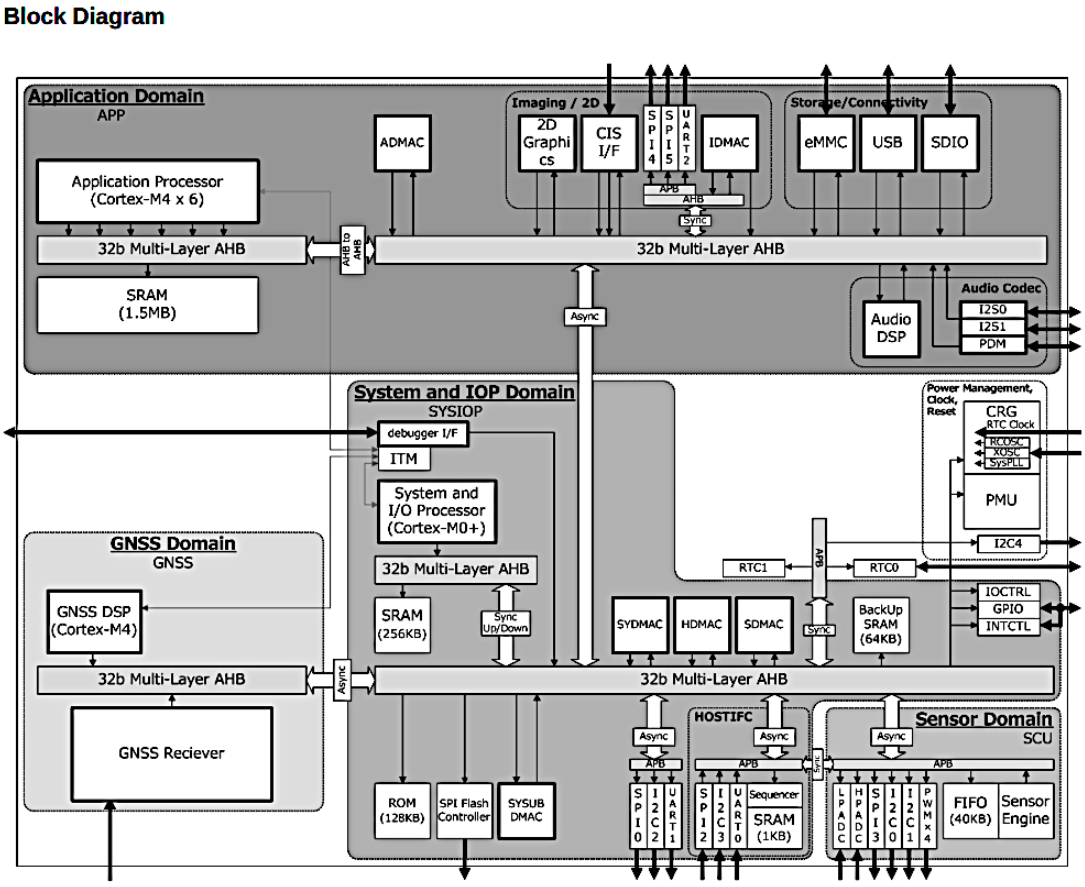
Pro vývoj aplikací poskytuje kompletní sadu nástrojů:

- Sadu standartních knihoven (SDK) STM32CubeL4.
- Vývojové prostředí s editorem zdrojového kódu Integrated Development Environment (IDE) STM32CubeIDE.
- Konfigurátor s počátečním nastavením mikrokontroleru STM32CubeMX.
- Run-time nástroj pro ladění programů STM32CubeMonitor.
- Software pro programování STM32CubeProg.
- Podporovány jsou kompilátor jazyka C/C++ GCC, Keil a IAR.

Kromě těchto vývojových nástrojů poskytuje společnost STMicroelectronics prostředí pro automatizované strojové učení s názvem NanoEdge AIStudio [35], které je primárně určeno pro mikrokontrolery STM32. Tento nástroj zahrnuje oblasti použití, jako je detekce anomálií, detekce hodnot mimo stanovené meze, regrese a klasifikace. Vedle NanoEdge AI Studia firma nabízí další nástroje pro optimalizaci naučených sítí pro mikrokontrolery a pro vytváření datasetů z různých typů čidel.

3.3 Mikrokontroler CXD5602

Mikrokontroler společnosti Sony CXD5602 je komplexní systém na čipu (SoC), který je součástí aplikačního modulu (SoM) Spresense [36]. Tento mikrokontroler je založen na šesti uživatelsky dostupných jádrech ARM Cortex M4F s taktovací frekvencí 156 MHz. Další jádro tohoto typu se nachází v bloku satelitní navigace, kde se stará o výpočet souřadnic a řízení rádiového bloku. Posledním jádrem je ARM Cortex M0+, které je určeno pro rychlé vstupně / výstupní operace.



Obrázek 3.3. Blokové schéma mikrokontroleru CXD5602 [37]

Paměť je integrovaná součást SoC a zahrnuje 8 MiB FLASH a 1.5 MiB RAM. Na SoM Spresense je také pomocný obvod Sony CXD5247, který je určený pro zpracování signálů pole čtyř analogových nebo osmi digitálních mikrofonů. Dále převodník digitálně analogového signálu s vysokým rozlišením a koncový zesilovač třídy D pro sluchátka a reproduktory.

Tento obvod zajišťuje rovněž napájení celého systému, včetně nabíjení externího Li-Pol akumulátoru maximálním proudem 500 mA. Spresense s originální základní deskou tuto vlastnost ovšem nepodporuje. Při aplikaci tohoto modulu není možné použít stan-

dardní 3,3V logické úrovně, ale je nutné je upravit na 1,8V pomocí vhodných převodníků napěťových úrovní.

Ačkoli je modul schopen samostatného provozu, pro plné využití jeho potenciálu, je vhodné jej instalovat na základní desku, ať už přímo od výrobce Sony, nebo na desku dle vlastního návrhu. Osazením na základní desku získáme kromě napěťových převodníků také výstupy s pulsně šířkovou modulací (PWM), vstupy analogově digitálních převodníků (ADC), rozhraní pro připojení paměťových karet, pole mikrofonů, výstup pro sluchátka a další. Sony uvádí velmi nízkou spotřebu okolo 100 mW [38].

II. PRAKTICKÁ ČÁST

4 PROGRAMOVACÍ JAZYK A POMOCNÉ NÁSTROJE.

Pro vývoj a testování bylo využito následující softwarové a hardwarové vybavení:

Programovací jazyky

Programy jsou psány v jazyce C, protože je momentálně jediný jazyk, jehož SDK existuje pro všechny použité platformy. Kompilátor byl použit GCC gcc-arm-none-eabi-10.3-2021.10, Cmake a Ninja.

IDE a SDK:

- Clion, základní IDE pro vývoj od společnosti JetBrains – Pro vývoj knihovny a s SDK od Raspberry pro moduly na bázi Raspberry Pico a deriváty s mikrokontrolerem RP2040.
- STM32Cube a STM32CubeMX IDE spolu s SDK pro mikrokontrolery řady STM32 určeno pro modul Nucleo s mikrokontrolerem STM32L432KC od společnosti STMicroelectronics.
- Arduino IDE 2.0.2 pro modul Spresense s mikrokontrolerem CXD5602 a SDK pro IDE Arduino od společnosti Sony.

Moduly s kontrolery na bázi jader ARM Cortex M4F a M0+

Raspberry Pico, SPRESENSE, NUCLEO-L432KC.

Měřící přístroje

Stolní voltmetr Voltcraft VC-655BT a digitální dvoukanálový osciloskop Sigilent SDS 1202X-E.

5 REALIZACE KNIHOVNY PRO OPERACE S ČÍSLY S PEVNOU ŘÁDOVOU ČÁRKOU (FX) V JAZYCE C

Matematická knihovna je implementována ve formě zdrojových kódů jazyka C (soubory s příponou .c) a příslušnými hlavičkovými soubory (s příponou .h) za účelem maximalizace přenositelnosti a flexibility při použití na různých platformách. Pro zahrnutí těchto zdrojových kódů do projektu, je nutné přidat je do sekce externích zdrojů nebo přímo do zdrojových kódů projektu.

Takový přístup umožňuje jednoduché úpravy, opravy a rozšíření funkcionalit knihovny podle potřeb uživatelů. Na druhou stranu může zapříčinit delší dobu překladu, zejména u větších projektů, což může být nevýhodou.

Celá knihovna je rozdělena do několika samostatných celků, dle funkcionality. Jsou realizovány tyto oblasti matematické knihovny ve FX formátu čísel:

- Základní matematické funkce sčítání, odčítání, násobení, dělení, mocnina a operace sdružující násobení a přičtení hodnoty k předešlé hodnotě.
- Funkce pro textový vstup a výpis hodnot srozumitelný pro člověka.
- Funkce pro textový vstup a výstup do souborů a funkcí pro textové zpracování dat.
- Vektor jednorozměrná dynamická datová struktura, pro uchování a práci s daty.
- LUT funkce pro urychlení výpočtů složitějších funkcí.

5.1 Definice a popis datových struktur a funkcí pro prezentaci FX

V následujících kapitolách jsou popsány základní matematické operace, jako je sčítání, odčítání, násobení a dělení, což tvoří základní kameny knihovny pro práci s FX aritmetikou.

5.1.1 Základní operace sčítání, odčítaní násobení a dělení

```
static inline fxp16_t fxpMulInLine(fxp16_t numA, fxp16_t numB)
static inline fxp16_t fxpDivInLine(fxp16_t numA, fxp16_t numB)
static inline fxp16_t fxpAddInLine(fxp16_t numA, fxp16_t numB)
static inline fxp16_t fxpSubInline(fxp16_t numA, fxp16_t numB)
```

V počáteční fázi implementace byly základní aritmetické operace, jako jsou násobení, dělení, sčítání a odčítání, realizovány ve formě funkcí. Pro násobení byla vytvořena

specifická funkce s názvem numFxp fxpMul(numFxp a, numFxp b), která přijímá dvě čísla ve formátu FX jako vstupní parametry.

Po úspěšném vytvoření funkcí pro násobení čísel ve formátu FX bylo provedeno první testování funkčnosti na osobním počítači. Pro ověření správnosti a efektivity implementace byla zvolena operace násobení. Testování spočívalo v porovnání výsledků získaných pomocí nově vytvořené funkce s očekávanými hodnotami, což umožnilo identifikovat případné chyby nebo nedostatky v algoritmu.

Následně proběhly testy rychlosti a byla zvolena operace násobení. Test spočíval v opakovaném násobení stejných čísel v každé iteraci. Pro měření byla vytvořena funkce pro získání časového razítka viz výpis 1:

Listing 1. Funkce pro získání časového razítka

```
double getMs() {
  gettimeofday(&tv, NULL);

  double time_in_mill =
  (tv.tv_sec) * 1000 + (tv.tv_usec) / 1000;
  return time_in_mill;
}
```

Funkce ve výpisu 1 byla realizována pouze pro potřeby měření času pro osobní počítače s procesory Amd a Intel. V kontrolerech byly realizovány jinými prostředky danými jejími standardními knihovnami (Standard Development Kit dále SDK). Většinou platí, že každý výrobce má vlastní verzi SDK a tyto časové funkce realizuje jinak. Je proto nutno funkce pro měření času konzultovat s manuálem k danému SDK. Je také třeba věnovat pozornost úrovni abstrakce, neboť výsledky měření nemusí být správné ať už z důvodu velikosti, nebo pozice měření v kódu vlivem optimalizací.

Testování odhalilo nevhodnost použití obvyklé funkce v programovacím jazyce C pro operaci násobení z hlediska rychlosti. Volání funkce v tomto kontextu vykazuje významnou časovou režii. Pro zlepšení rychlosti výpočtů existují dva hlavní přístupy k realizaci kódu:

1. Využití maker v jazyce C viz výpis 3. Preprocesor vkládá takto definovanou funkci přímo na místo v kódu, kde je makro voláno, čímž eliminuje časovou režii spojenou s voláním funkce. Při použití maker je však nezbytné postupovat obezřetně, neboť opomenutí zásad tvorby maker může vést ke vzniku těžko detekovatelných chyb [43][44].

2. Využití inline funkcí ve formě "static inline foo()" viz výpis 5. Tento přístup je preferován, neboť nepředstavuje problémy související s nevhodně použitými vstupními parametry, které se mohou objevit u maker. Inline funkce mají minimální

časovou ztrátu ve srovnání se standardními funkcemi, což je ideální pro provádění rychlých operací. Podpora pro tento typ funkcí byla zavedena ve specifikaci jazyka C od verze C99 [45] [46].

Listing 2. Standardní funkce pro operaci násobení FX

```
/**
* fixed point multiply
* @param numA
* @param numB
* @return
*/
fxp16_t fxpMul(fxp16_t numA, fxp16_t numB) {
  numA.num = (int16_t) (((numA.num * numB.num)) >> numA.frac);
  return numA;
}
```

Listing 3. Makro jazyka C pro operaci násobení FX

```
#dee FX_MUL3(C, A, B) { \
  C.num =(int16_t) (((A.num * B.num)) >> A.frac); \
  C.frac = A.frac; \
}
```

Listing 4. Static inline funkce pro operaci násobení FX

```
/**
* fixed point multiply
* @param numA
* @param numB
* @return
*/
static inline fxp16_t fxpMulInLine(fxp16_t numA, fxp16_t numB) {
  numA.num = (int16_t) (((numA.num * numB.num)) >> numA.frac);
  return numA;
};
```

Při vývoji matematické knihovny je tedy důležité zvážit oba způsoby realizace kódu a zvolit ten, který nejlépe splňuje požadavky na rychlost a současně odolnost vůči chybám, zejména v případě použití maker.

Použití nastavení „Release“ namísto „Debug“ při zkoušení bylo zvoleno pro dosažení nejvyššího výkonu kódu. Nastavení „Debug“ totiž často zahrnuje dodatečné informace pro ladění a zaznamenávání. To způsobuje nežádoucí zpomalení kódu. V produkčním prostředí se proto vždy používá konfigurace „Release“. Pokud využíváme in-line funkce, je nutná opatrnost při testování v nastavení „Debug“. Důvodem je, že in-line funkce mohou být převedeny na standardní funkce s navýšenou časovou režií. Tento jev může

negativně ovlivnit výkonnost testovaného kódu a vést k nesprávným závěrům o jeho rychlosti a efektivitě.

Postup testování rychlosti operace násobení v prostředí kontroléru RP2040. Pro zamezení použití optimalizací kompilátorem jazyka C je potřebné zajistit skutečné provedení požadovaného počtu testovaných operací. K tomu byl využit výsledek násobení v následujícím cyklu testované operace. Je tedy realizována operace dle vztahu (5.1):

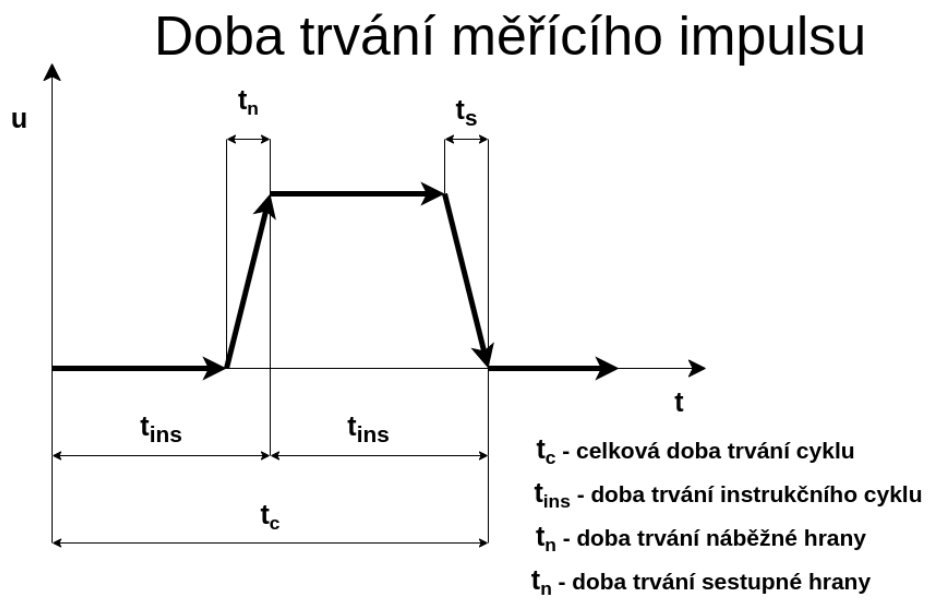
$$y_n = fxpMul(y_{n-1}, b) \tag{5.1}$$

Testovací program byl spuštěn na jednom jádře ze dvou dostupných v mikrokontroleru RP2040 s taktovací frekvencí 133 MHz. Počátek testované operace byl označen náběžnou hranou na portu GP22 a ukončení operace sestupnou hranou na stejném portu. Doba trvání byla změřena za použití digitálního osciloskopu, která však není dána pouze testovanou operací samotnou, ale také časem potřebným pro změnu stavu na portu GP22. Pro účely srovnání bylo toto zvýšení času zanedbáno, neboť bylo součástí všech měření.

Pro přesné určení doby trvání vstupně-výstupní (I/O) operace lze využít postupné změny úrovně na zkoumaném portu GP22, přičemž doba trvání je poté dána následujícím vztahem (5.2):

$$t_{ins} = \frac{t_c}{2} \tag{5.2}$$

kde t_{ins} je čas potřebný pro nastavení hodnoty vstupně výstupního portu (Input/Outpu dále I/O), t_c je celková doba trvání pulsu viz obrázek 5.1.



Obrázek 5.1. Doba trvání impulsu

5.1.2 Operace násobení čísel ve formátu FX a FP

Celkem byly testovány čtyři varianty implementující násobení. První používá standardní funkci pro FX čísla označenou jako `fxpMul`. Druhá používá in-line funkci pro FX čísla označenou jako `fxpMulInLine`. Třetí varianta je vytvořena pomocí maker preprocesoru jazyka C (MC) je označena jako `FX_MUL3(C, A, B)`. Poslední variantou je pak násobení realizované pomocí FP čísel a je označena jako *Float Point A * B*. Každá varianta operace násobení byla v cyklu opakována 1000x dle metodiky uvedené na straně 49.

Celkové časy jsou pak uvedeny ve výpisu 5:

Listing 5. Výkonové testy násobení

```
************************************************************
*** Multiplication
************************************************************
-
*** Time test multiplication [fxpMulInLine] math ...
************************************************************
A: 30.23438, B: 0.99903
Iteration count j: 1000
Last value: 11.06836
Total [ Multiplication fxpMulInLine ] time 82 uS
Average time for multiplication [fxpMulInLine]: 82 uS
-
-
*** Time test multiplication [FX_MUL3(C, A, B)] math ...
```

```
**********************************************************
A: 30.23438, B: 0.99903
Iteration count j: 1000
Last value: 11.06836
Total [ Multiplication FX_MUL3(C, A, B) ] time 83 uS
Average time for multiplication [FX_MUL3(C, A, B)]: 83 uS
-
-
*** Time test multiplication [fxpMul] math ...
**********************************************************
A: 30.23438, B: 0.99903
Iteration count j: 1000
Last value: 11.06836
Total [ Multiplication fxpMul ] time 600 uS
Average time for multiplication [fxpMul]: 600 uS
-
-
*** Time test multiplication [Float Point] math ...
**********************************************************
A: 30.233999, B: 0.999000
Iteration count j: 1000
Last value: 11.117043
Total [ Multiplication Float Point ] time 660 uS
Average time for multiplication [Float Point]: 660 uS
```

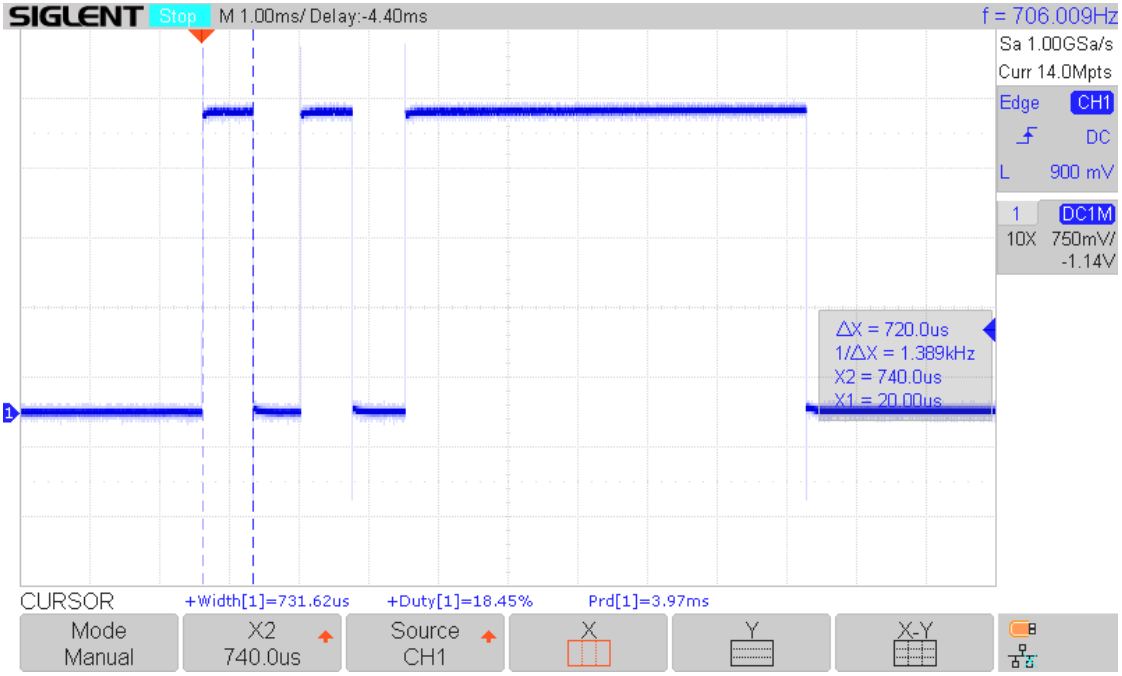
Tabulka 5.1. Porovnání časů potřebných pro provedení 1000 iterací násobení

Metoda násobení	**Celkový čas**	**Násobek času fxpMulInLine**
fxpMul	615 us	8,42
fxpMulInLine	73 us	1
FX_MUL3(C, A, B)	75 us	1,03
*Float Point A * B*	645 us	8,84

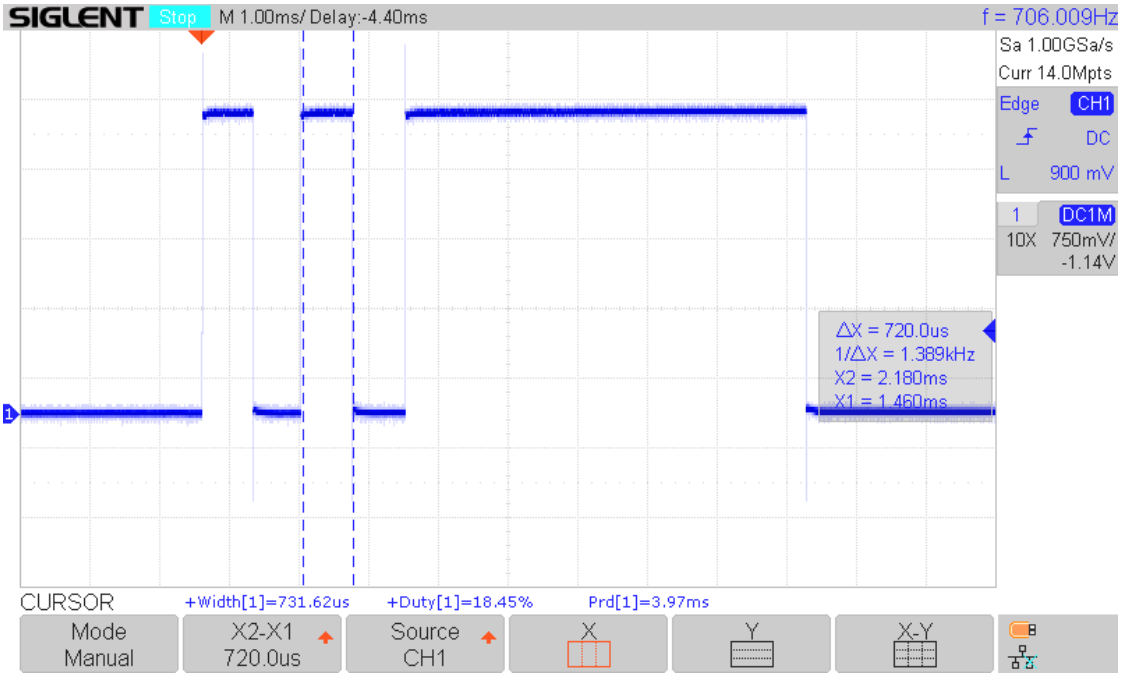
Z tabulky 5.1, která zobrazuje celkové časy pro jednotlivé varianty násobení, plyne, že rozdíl mezi in-line verzí a verzí realizovanou pomocí maker preprocesoru jazyka C (MC) je zanedbatelný a je dán spíše fluktuací podmínek v procesoru během vykonávání programu. Jako referenční čas byla zvolena nejrychlejší operace označena `fxpMulOpt`. Oproti tomu operace násobení se stejným počtem opakování byla 8x pomalejší.

Vzhledem k optimalizacím, které kompilátor jazyka C aplikuje, byla doba trvání

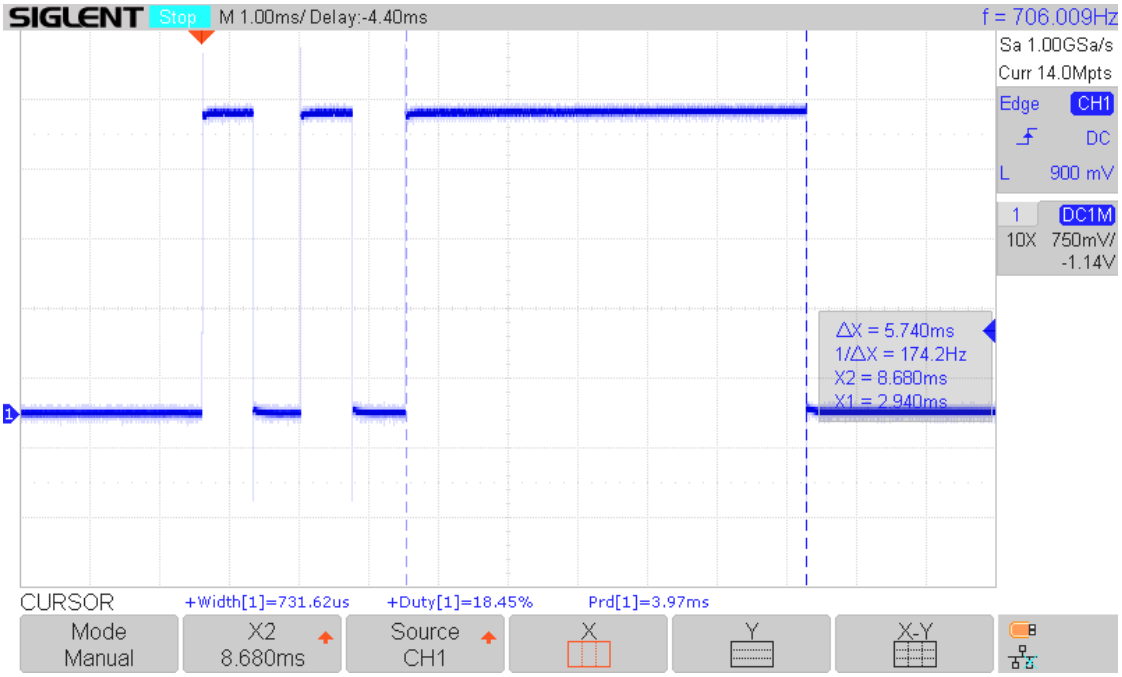
ověřena i pomocí osciloskopu. Měřila se doba trvání impulsu, jehož náběžná hrana byla vygenerována na I/O portu GP22 na počátku násobení a sestupná hrana po ukončení cyklu násobení. Jednotlivé délky pulsu byly změřeny pomocí funkce osciloskopu `Cursor`. Doba trvání jednotlivých operací jsou změřena na obrázcích 5.2, 5.3 a 5.4.



Obrázek 5.2. Ilustrace doby provedení operace fxMul()



Obrázek 5.3. Ilustrace doby provedení operace FX_MUL3(C, A, B)



Obrázek 5.4. Ilustrace doby provedení operace (float)A * (float)B

5.1.3 Operace sčítání čísel ve formátu FX a FP

Operace sčítání byla provedena obdobně, jako v případě násobení 50 s tím, že byla vynechána varianta funkce realizované pomocí makro jazyka, protože doba trvání je přibližně stejná.

Listing 6. Výkonové testy sčítání

```
**********************************************************
*** Addition
**********************************************************
-
*** Time test addition [fxpAddInLine] math
**********************************************************
A: 0.00000, B: 0.00098
Iteration count j: 1000
Last value: 0.97656
Total [ Addition fxpAddInLine ] time 3 uS
Average time for addition [fxpAddInLine]: 3 uS
-
-
*** Time test addition [fxpAdd] math
**********************************************************
A: 0.00000, B: 0.00098
Iteration count j: 1000
Last value: 0.97656
Total [ Addition fxpAdd ] time 263 uS
```

```
Average time for addition [fxpAdd]: 263 uS
-
-
*** Time test addition [Float Point]
*********************************************************
A: 0.000000, B: 0.001000
Iteration count j: 1000
Last value: 0.999991
Total [ Addition Float Point ] time 726 uS
Average time for addition [Float Point]: 726 uS
```

Doba trvání softwarově emulované operace odčítání čísel FP pro 1000 iterací byla nejpomalejší a byla 726 us. Oproti tomu dělení pomoci funkce `fxpDivInLine()` byla 3 us. Funkce realizována pomocí normálních funkcí `fxpMul()` 263 us. výsledky jsou shrnuty v tabulce 5.2

Tabulka 5.2. Porovnání časů potřebných pro provedení 1000 iterací sčítání

Metoda sčítání	**Celkový čas**	**Násobek času fxpAddInLine**
fxpAdd	263 us	87,67
fxpAddInLine	3 us	1
Float Point A + B	726 us	242

5.1.4 Operace odčítaní čísel ve formátu FX a FP

Operace odčítaní byla provedena obdobně, jako v případě násobení 50 s tím, že byla vynechána varianta funkce realizovaná pomocí makro jazyka.

Listing 7. Výkonové testy odčítání

```
*********************************************************
*** Subtraction
*********************************************************
-
*** Time test subtraction [fxpSubInLine] math
*********************************************************
A: 30.00000, B: 0.00098
Iteration count j: 1000
Last value: 29.02344
Total [ Subtraction fxpSubInLine ] time 4 uS
Average time for subtraction [fxpSubInLine]: 4 uS
-
-
*** Time test subtraction [fxpSub] math
*********************************************************
A: 30.00000, B: 0.00098
Iteration count j: 1000
Last value: 29.02344
Total [ Subtraction fxpSub ] time 255 uS
Average time for subtraction [fxpSub]: 255 uS
-
-
*** Time test subtraction [Float Point]
*********************************************************
A: 30.233999, B: 0.001000
Iteration count j: 1000
Last value: 29.234549
Total [ Subtraction Float Point ] time 757 uS
Average time for subtraction [Float Point]: 757 uS
```

Doba trvání softwarově emulované operace odčítání čísel FP pro 1000 iterací byla nejpomalejší a činila 757 us. Oproti tomu dělení pomocí funkce `fxpDivInLine()` byla 4 us. Funkce realizována pomocí normálních funkcí `fxpMul()` byla 255 us. Výsledky jsou shrnuty v tabulce 5.3.

Tabulka 5.3. Porovnání časů potřebných pro provedení 1000 iterací odčítání

Metoda odčítání	**Celkový čas**	**Násobek času fxpSubInLine**
fxpSub	255 us	63,75
fxpSubInLine	4 us	1
Float Point A - B	757 us	189,25

5.1.5 Operace dělení čísel ve formátu FP

Operace odčítaní byla provedena obdobně, jako v případě násobení 50 s tím, že byla vynechána varianta funkce realizovaná pomocí makro jazyka.

Listing 8. Výkonové testy dělení

```
**********************************************************
*** Division
**********************************************************
-
*** Time test division [fxpDivInLine] math ...
**********************************************************
A: 30.23438, B: 1.00098
Iteration count j: 1000
Last value: 11.39161
Total [ Division fxpDivInLine ] time 351 uS
Average time for division [fxpDivInLine]: 351 uS
-
-
*** Time test division [fxpDiv] math ...
**********************************************************
A: 30.23438, B: 1.00098
Iteration count j: 1000
Last value: 11.07911
Total [ fxpDiv ] time 746 uS
Average time for division [fxpDiv]: 746 uS
-
-
*** Time test division [Float Point] math ...
**********************************************************
```

```
A: 30.233999, B: 1.001000
Iteration count j: 1000
Last value: 11.127662
Total [ Division Float Point ] time 649 uS
Average time for division [Float Point]: 649 uS
```

Doba trvání softwarově emulované operace dělení čísel FP pro 1000 iterací byla 649 us. Oproti tomu dělení pomocí funkce `fxpDivInLine()` byla 351 us. Funkce realizována pomocí normálních funkcí `fxpMul()` byla nejpomalejší a byla 746 us. Výsledky jsou shrnuty v tabulce 5.4.

Tabulka 5.4. Porovnání časů potřebných pro provedení 1000 iterací dělení

Metoda dělení	**Celkový čas**	**Násobek času fxpDivInLine**
fxpDiv	255 us	63,75
fxpDivInLine	4 us	1
Float Point A / B	757 us	189,25

5.2 Operace pro textový vstup a výstup z / do souborů

Pro efektivní prezentaci číselných hodnot, zejména v kontextu maticových zobrazovačů, tiskáren a terminálových výstupů, byly vytvořeny funkce pro konverzi FX formátu do textového řetězce. Klíčovou součástí bylo také zpracování vstupů z terminálů a jiných vstupních zařízení, jako jsou klávesnice, čtečky QR kódů a podobně.

```
void printFxp(fxp16_t numA);
void sprintFxp(char *string, size_t strSize, fxp16_t numA);
fxp16_t strToFxp(const char *string, int8_t frac);
```

```
void printFxp(fxp16_t numA);
```

Vytiskne na standardní výstup textovou prezentaci hodnoty fxp16_t numA.

```
void sprintFxp(char *string, size_t strSize, fxp16_t numA);
```

Vloží textovou prezentaci do řetězce pro další použití v složených řetězcích a sestavách.

```
fxp16_t strToFxp(const char *string, int8_t frac);
```

Převod z řetězcové proměnné je použitelný pro vkládání dat z terminálu a textových datových zdrojů.

Tento typ funkcí byl testován pomocí pseudonáhodně vygenerovaných hodnot a porovnán s požadovaným výstupem. Rychlostní testy, vzhledem k jejich účelu, nebyly realizovány.

5.3 Konverzní funkce FX na FP a FP na FX

Pro konverzi hodnot mezi FP a FX byly vytvořeny dvě funkce:

```
fxp16_t floatToFxp(float input);
float fxpToFloat(fxp16_t fxp);
```

```
//
```

```
fxp16_t floatToFxp(float input);
```

Funkce je navržena pro konverzi čísel z formátu s plovoucí řádovou čárkou (FP) na formát s pevnou řádovou čárkou (FX). Tato funkce může být využita pro konverzi datových souborů ve formátu FP na interní formát FX, pro vstupy z datových zdrojů, které nepodporují formát s pevnou řádovou čárkou. Slouží také programátorům pro snazší a přehlednější definici konstant.

```
float fxpToFloat(fxp16_t fxp);
```

Funkce zajišťuje převod v opačném směru, zejména pro výstupní soubory, ale i pro případné využití matematických funkcí, které nejsou implementovány v doméně FX čísel.

Testování bylo provedeno obdobným způsobem jako v případě předchozích funkci, nicméně s tím rozdílem, že výkonové testy již provedeny byly. Tento typ může být totiž využit ke konverzi tabulek, nebo generování tabulek funkcí pro jednu hodnotu (Look Up Table dále LUT), což může být časově náročná činnost.

6 DATOVÉ STRUKTURY A FUNKCE PRO UCHOVÁNÍ A PŘEDÁVÁNÍ DAT

Pro manipulaci s formátem FX byly vytvořeny nové modifikace datových typů.

6.1 Základní datový typ fxp16_t

Základní typ pro uložení FX o velikosti 16 bitů. Je realizován pomocí struktury:

```
typedef struct {
    int16_t num; // fixed point number
    int8_t frac; // fraction bits
} fxp16_t;
```

kde `int16_t num` je vlastní hodnota datového typu a má velikost 16 bitů,
`int8_t frac` je počet bitů za desetinnou čárkou a má velikost 8 bitů.

6.2 Datový typ pole a operace s ním

Pole je jednou ze základních datových struktur, která v kontextu FX matematiky plní funkci kontejneru pro stejný format Qm.n, ale je v něm možné uchovat i různé jeho varianty.

6.2.1 Datové struktury

Tento typ uchovává sekvenci hodnot typu `fxp16_t`, která nemusí být stejného typu v prezentaci Qm.n:

```
typedef struct {
    fxp16_t num; // fixed point number
    int32_t size; // array size
} fxArray;
```

kde `fxp16_t num` je hodnota datového typu `fxp16_t`,
`int32_t size` je maximální velikost tohoto pole.

6.2.2 Funkce pro práci s datovým typem pole

```
fxArray *createrArrayFx(int16_t size, bool force);
```

Funkce alokuje paměť pro specifický datový typ dané velikosti, přidělí plný paměťový prostor pro zrychlení práce s polem.

```
freeArrayFx(fxArray *pArray);
```

Funkce smaže data i samotnou strukturu z paměti a uvolní ji, jde v podstatě o destruktor.

6.3 Datový typ vektor a operace s ním

Datový typ vektor je jednou z nejpoužívanějších struktur v knihovně, slouží k uchování a předání jednorozměrných dat mezi funkcemi.

6.3.1 Datové struktury

Datový typ `fxVect` je variantou typu `fxArray`, který ukládá sekvenci hodnot s konzistentní Qm.n prezentací. Tento model je kompaktnější a nabízí lepší paměťovou efektivitu, jelikož nevyžaduje uchování informace o počtu bitů za řádovou čárkou pro každou hodnotu. Rychlost operací je také zvýšená, jelikož funkce nemusí konvertovat jednotlivé varianty Qm.n na stejný typ. Oproti tomu, hlavní nevýhodou `fxVect` je jeho nižší flexibilita ve srovnání s `fxArray`. Tento typ reprezentuje následující struktura:

```
typedef struct {
  int16_t num;  // fixed point number
  int8_t frac;  // fraction bits
  int_t size;   // array size
} fxp16_Vect;
```

kde `int16_t num` je vlastní hodnota datového typu a má velikost 16 bitů,
`int8_t frac` je počet bitů za desetinnou čárkou a má velikost 8 bitů,
`int32_t size` je maximální velikost tohoto vektoru a má velikost 32 bitů.

6.3.2 Funkce pro práci s datovým typem vektor

```
fpVect *createVectFp(uint16_t size);
fxVect *createVectFx(uint16_t size, uint8_t frac);
```

Funkce alokuje paměť pro specifický datový typ dané velikosti a funkce přidělí plný paměťový prostor pro zrychlení práce s vektorem. Verze pro FP a FX čísla.

```
void printVectFp(fpVect const *vect);
void printVectFx(fxVect const *vect);
```

Funkce vytiskne obsah vektoru.Verze pro FP a FX čísla.

```
void freeVectFp(fpVect *vec);
void freeVectFx(fxVect *vec);
```

Funkce smaže data i samotnou strukturu z paměti a uvolní ji. Verze pro FP a FX čísla.

7 ZÁKLADNÍ STRUKTURY NEURONOVÝCH SÍTÍ PRO FORMÁT FX A FP

V následujících kapitolách budou postupně diskutovány neuronové sítě typu perceptron a feedforward a s nimi souvisejícími datovými strukturami.

7.1 Specifický datový typ pro oblast umělých neuronových sítí nnTopology

Popis datových struktury a funkce specifické pro neuronové sítě. Jde zejména o struktury, které umožňují definovat strukturu neuronových funkcí, umožňují práci s váhami.

7.1.1 Datové struktury

Datový typ `nnTopology` je specifickou strukturou pro definici struktury Feedforward neuronové sítě. Tato struktura obsahuje informaci o počtu vrstev, pole s počtem neuronů v každé vrstvě a maximální počet vah. Tento typ reprezentuje následující struktura:

```
typedef struct {
  uint16_t *neuronCount;  // array with counts neurons in layer
  uint16_t maxWeightsInLayer; // maximum weights in layers
  uint16_t layers;  // layer counts
} nnTopology;
```

kde `int16_t *neuronCount` je pole s počtem neuronů v každé vrstvě a má velikost 16 bitů,
`int16_t maxWeightsInLayer` obsahuje informaci o největším počtu vah ve vrstvách a má velikost 16 bitů,
`int16_t layers` je počet vrstev neuronové sítě včetně vstupní a má velikost 16 bitů.

7.1.2 Funkce pro práci s datovým typem nnTopology a váhami

```
nnTopology *createTopology(uint16_t layers, const uint16_t
  neuronsInLayer[]);
```

Funkce alokuje paměť pro specifický datový typ dané velikosti a vytvoří strukturu s daným počtem neuronů v každé vrstvě a uloží do ní příslušný počet vrstev. Tato struktura slouží dalším funkcím pro definici neuronové sítě, jako jsou váhy a podobně. Verze je identická pro FX i FP čísla.

```
void freeTopology(nnTopology *topology);
```

Funkce smaže data i samotnou strukturu z paměti a uvolní ji. Verze je identická pro FX i FP čísla.

```
fpVect *createWeightArray(nnTopology const *topology);
fxVect *createWeightArray(nnTopology const *topology);
```

Funkce vygeneruje podle definiční struktury `nnTopology` vektor s váhami a tyto váhy naplní náhodnými čísly. Verze pro FP a FX čísla.

```
void printWeightArray(nnTopology const *topology, const fpVect *
  weightArray);
void printWeightArray(nnTopology const *topology, const fxVect *
   weightArray);
```

Funkce vytiskne formátovaně do terminálu váhy vygenerované funkcí `*createWeightArray()`, nebo upravené funkcemi pro učení neuronových sítí. Verze pro FP a FX čísla.

```
void printWeightParam(nnTopology const *topology, const fpVect *
  weightArray);
void printWeightParam(nnTopology const *topology, const fxVect *
   weightArray);
```

Vytiskne strukturu vah bez hodnot. Verze pro FP a FX čísla.

7.1.3 Funkce pro práci s LUT pro funkce Sigmoid() a Tanh()

```
static inline int16_t sigmoidLut(int16_t numA);
static inline fxp16_t sigmoidLutFx(fxp16_t numA)
```

Funkce vrací hodnotu matematické funkce *Sigmoid*() pomocí LUT. První verze pracuje pouze s 16 bitovou hodnotou se znaménkem `int16_t`. Druhá funkce je podobná, pouze pracuje s úplným FX číslem, `fxp16_t`.

```
static inline int16_t tanhLut(int16_t numA);
static inline fxp16_t tanhLutFx(fxp16_t numA)
```

Funkce vrací hodnotu matematické funkce *Tanh*() pomocí LUT. První verze pracuje pouze s 16 bitovou hodnotou se znaménkem `int16_t`. Druhá funkce je podobná, pouze pracuje s úplným FX číslem, `fxp16_t`.

```
int16_t generateTable(fceTypes type, int8_t frac)
```

Funkce je generátorem tabulek LUT. Prozatím podporuje pouze funkce *Sigmoid*() a *Tanh*() a automaficky generuje hlavičkové soubory `tanhLut.h` a `sigmoidLut.h` pro jazyka C. Tyto hlavičkové soubory jsou nezbytné pro funkci LUT funkcí a automaticky se importují.

7.2 Perceptron

Perceptron je implementován v modulu perceptron.h a perceptron.c. Je realizován pomocí static inline funkcí kvůli rychlostní optimalizaci.

```
float perceptron(fpVect *pDataset, fpVect *pWeights);
int16_t perceptronFx(fxVect *pDataset, fxVect *pWeights);
```

Funkce implementuje algoritmus neuronové sítě typu perceptron. Vstupem jsou dva vektory, kde `*pDataset` jsou vstupní data pro zpracování a `*pWeights` jsou váhy. Velikost vektoru odpovídá vstupní dimenzi dat a váhový vektor je vždy delší o jednu hodnotu váhy biasu. Síť je navržena pro oba formáty, tedy pro FP a FX.

```
void perceptrontrain(fpVect *pDataset, fpVect *pWeights, uint32_t
  epochs);
void perceptrontrainFx(fxVect *pDataset, fxVect *pWeights, uint32_t
    epochs)
```

Funkce realizuje učící algoritmus pro perceptron, který byl popsán v teoretické části. Vstupem je promíchaný dataset `*pDataset` a váhový vektor pro neuronové sítě typu feedforward. Sít je navržena pro oba formáty, tedy pro FP a FX.

7.3 Feedforward síť s backpropagation učením

Feedforward neuronová síť je jednou z nejpoužívanějších sítí. Postupně jsou probrány implementační detaily, následované výčtem a popisem funkcí jak pro samotnou inferenci, tak i učení.

7.3.1 Implementace

Síť typu feedforward (FF) je realizována v souborech backpropag.c a backpropag.h spolu s backpropagation učícím algoritmem. V tomto případě je použita implementace s využitím standardních funkcí, neboť časová náročnost volání podprogramu ve strojovém kódu je v porovnání s výpočetní náročností neuronové sítě zanedbatelná. Tato síť je navržena pro formáty čísel FP a FX.

7.3.2 Funkce realizující neuronovou typu feedforward a vlastní učení sítě backpropagation

```
void fpFeedForwardNetworkSig(nnTopology *topology, fpVect *weights,
    fpVect *inData, fpVect *outData)
void fpFeedForwardNetworkSigFx(nnTopology *topology, fxVect *
    weights, fxVect *inData, fxVect *outData)
```

Funkce implementuje algoritmus neuronové sítě typu feedforward. Vstupem jsou čtyři vektory, kde `*topology` je použitá topologie sítě, `*inData` jsou vstupní data pro zpracování a `*weights` jsou váhy a `*outData` je výstupní vektor. Síť je navržena pro oba formáty, tedy pro FP a FX.

```
void BackPropagationTrain(nnTopology *topology, fpVect *weights,
   fpVect *trainingData, fpVect *labels, uint32_t epochs, float
   learningRate)
void BackPropagationTrainFx(nnTopology *topology, fxVect *weights,
    fxVect *trainingData, fpVect *labels, uint32_t epochs, fxp16_t
    learningRate)
```

Funkce implementuje učící algoritmus neuronové sítě typu feedforward. Vstupem jsou čtyři vektory, kde v `*topology` je definována topologie sítě, `*trainingData` jsou vstupní data pro zpracování, `*weights` jsou váhy a `*labels` je vektor s označením trénovacích dat, `epochs` počet trénovacích epoch a `learningRate` je koeficient učení. Síť je navržena pro oba formáty, tedy pro FP a FX.

8 DATASETY PRO TESTY A TESTOVÁNÍ JEDNOTLIVÝCH TYPŮ NEURONOVÝCH SÍTÍ

Jelikož kvalita datasetu je klíčovou složkou pro úspěšný proces učení v kontextu umělých neuronových sítí, bylo nezbytné pečlivě zvolit vhodné datasety. Hlavními kritérii pro výběr byly: (1) využití v oblasti verifikace modelů a výuky, (2) kompatibilita s vybranými modely neuronových sítí, a to nejen z hlediska tematického zaměření, ale také velikosti dat. Na základě těchto kritérií byly pro účely této bakalářské práce vybrány dva datasety: Iris a Occupancy Detection.

8.1 Iris data set

Prvním vybraným datasetem je Iris data set. Tento dataset je jedním z nejznámějších a nejčastěji využívaných datasetů v oblasti strojového učení a je dostupný na webu UCI Machine Learning Repository[1)]. Obsahuje 150 záznamů, které reprezentují tři druhy kosatců: kosatec setý (Iris setosa), kosatec sibiřský (Iris virginica) a kosatec zlatohlavý (Iris versicolor). Každý záznam má čtyři atributy: délku a šířku okvětních lístků a délku a šířku kališních lístků.

8.2 Occupancy Detection

Druhým vybraným datasetem je Occupancy Detection dataset, který je rovněž dostupný na UCI Machine Learning Repository[2)]. Tento dataset je zaměřen na detekci obsazenosti místnosti na základě environmentálních faktorů jako je teplota, vlhkost, světelnost a úroveň CO2. Obsahuje přes 20 000 záznamů s 7 atributy. Je ideální pro analýzu časových řad a detekci vzorců.

Oba datasety jsou ideální pro verifikaci modelů neuronových sítí a jejich použití v této práci umožní nejen ověřit použitelnost zvolených metod, ale také poskytnout srovnatelné výsledky s jinými pracemi v oblasti strojového učení.

8.3 Realizace testů neuronových sítí a jejich provedení

Z povahy aritmetiky ve formátu FX čísel plyne vysoká rychlost operací, ale dynamický rozsah je horší než u čísel ve formátu FP. Proto jsou testy zaměřeny primárně tímto směrem.

[1)] `https://archive.ics.uci.edu/ml/datasets/iris`

[2)] `https://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+`

8.3.1 Test přesnosti zpracování dat perceptronem realizovaný pomocí formátu FX a FP

Tato síť nemá žádný konfigurační soubor, neboť se jedná o jediný neuron. Informaci o velikosti dimenze dat si nese vstupní vektor. V tomto případě je jeho velikost nastavena dva parametry. Test byl proveden v pěti iteracích, kdy na počátku byly vygenerovány náhodné váhy a následně proběhl samotný test. Každá iterace měla náhodně vygenerovaný vstupní vektor. Výsledek testu je v následujícím výpisu:

```
************************************************************
*** PERCEPTRON TEST
************************************************************
Weights FP
-0.540949,-0.307857,-0.016528
Weights FX
-0.54101, -0.30762, -0.01661

Iterace 0
FP > 0.292923,-0.294392 | 0.000000
FX > 0.29297, -0.29395 | 0

Iterace 1
FP > 0.265315,-0.824149 | 0.000000
FX > 0.26563, -0.82422 | 0

Iterace 2
FP > -0.560111,0.667472 | 1.000000
FX > -0.56055, 0.66699 | 1

Iterace 3
FP > 0.956512,0.967570 | 0.000000
FX > 0.95606, 0.96778 | 0

Iterace 4
FP > -0.157336,0.898158 | 1.000000
FX > -0.15723, 0.89844 | 1

*** PERCEPTRON TEST ISH
************************************************************
```

Jelikož jsou výsledné hodnoty binární, nelze určit míru shody přesněji. Z výpisu plyne, že v obou případech jsou, v reakci na stejná data, shodné výstupní hodnoty. Přesnost je tedy dobrá.

8.3.2 Test přesnosti zpracování dat feedforward sítí realizované pomocí formátu FX a FP

Testovaná síť je v konfiguraci `testNN2T[3] = {2, 3, 2}`, což znamená, že tato síť má celkem 3 vrstvy. První je vstupní vrstva se dvěma vstupy, následuje jedna skrytá vrstva se 6 neurony a poslední je výstupní vrstva opět se 6 neurony. Váhy byly náhodně vygenerovány pomocí funkce `createWeightArray(testNN2T)` pro formát FP a byla převedena na formát FX a zkopírována pomocí funkce `copyVectFpToFx( ... );` do váhového vektoru pro FX formát FF neuronové sítě. Ve výstupním výpisu testovacího programu je zvolen jen jeden vstupní vektor pro ilustraci správnosti funkce.

```
************************************************************
*** TEST FEEDFORWARD NEURAL NETWORK
************************************************************

*** Network config
************************************************************
Layer 0, weights 2:
Layer 1, weights 6:
Layer 2, weights 6:
Total weights: 15, Total layers: 3, Mem: 64 B


*** Weights FP
************************************************************
Layer 0, weights 2:
    1.000000 1.000000
Layer 1, weights 6:
    0.852761 -0.980697 -0.926578 -0.552949 0.693315 -0.608510
Layer 2, weights 6:
    -0.682327 -0.146927 -0.913226 0.749857 0.398071 0.758813
Total weights: 15, total layers: 3


*** Weights FX copy
************************************************************
Layer 0, weights 2:
    1.00000 1.00000
Layer 1, weights 6:
    0.85254 -0.98047 -0.92677 -0.55274 0.69336 -0.60840
Layer 2, weights 6:
    -0.68262 -0.14649 -0.91309 0.75000 0.39844 0.75879
Total weights: 15, total layers: 3
```

```
*** Input vector FP
*********************************************************
0.092000,-0.990000
*** Input vector FX
*********************************************************
0.09180, -0.99024


*** Output vector FP
*********************************************************
0.231683,0.785990
*** Output vector FX
*********************************************************
0.23145, 0.78516


*** TEST FEEDFORWARD NEURAL NETWORK ISH
*********************************************************
```

Z výpisu je zřejmé, že obě sítě dávají velmi podobný výsledek, a lze tak konstatovat, že z hlediska přesnosti, je prezentace čísel ve formátu FX dostatečná pro použití v neuronových sítích. Z praktických testů ovšem vyplynulo, že je vhodné použít normalizovaný dataset, aby se zabránilo přetečení. Pro formát čísel Q5.10 je maximální velikost vstupní dimenze rovna 31, neboť i v nejhorším případě bude při použití normovaných dat maximální součet právě 31. Jednotlivé varianty jsou zobrazeny v tabulce 1.4.

Další podmínkou je správná volba aktivační funkce. Měla by mít funkční hodnotu omezenu na interval $y \in < -1, 1 >$ (tanh()), případně $y \in < 0, 1 >$ (sigmoid()) viz obrázky 2.3. Důvodem je skutečnost, že takto zvolené aktivační funkce současně normalizují své výstupní hodnoty, které jsou vhodné pro zpracování v další vrstvě.

8.4 Test učení sítě perceptron

Pro trénování byl vytvořen základní dataset reprezentující funkci NOR. Počet epoch byl 10000, dataset měl 17 vstupních vektorů se dvěma hodnotami. Rychlost učení byla stanovena na 0.82. Výsledky a testovaní jednotlivých bloků jsou v následujícím výpisu:

```
   ************************************************************
*** PERCEPTRON LEARN TEST
************************************************************

*** Vector inDataPl:
************************************************************
0.000000,0.000000 | 0.000000
0.000000,1.000000 | 1.000000
1.000000,0.000000 | 1.000000
1.000000,1.000000 | 1.000000
0.000000,1.000000 | 1.000000
0.000000,0.000000 | 0.000000
1.000000,0.000000 | 1.000000
1.000000,0.900000 | 1.000000
0.350000,0.750000 | 1.000000
0.000000,0.000000 | 0.000000
0.000000,1.000000 | 1.000000
0.000000,0.100000 | 0.000000
0.000000,0.800000 | 1.000000
1.000000,0.000000 | 1.000000
0.250000,0.000000 | 0.000000
0.900000,0.450000 | 1.000000
1.000000,0.100000 | 1.000000


*** Vector weightsPl:
************************************************************
-0.389214,0.021957,0.436712
Training stopped after 1 epochs with error 0.117647

*** Vector updated:
************************************************************
0.323286,0.686957,0.436712
*** Vector updated:
************************************************************
1.000000,0.100000 > 1.000000  | 1.000000
0.900000,0.450000 > 1.000000  | 1.000000
1.000000,0.000000 > 1.000000  | 1.000000
1.000000,0.000000 > 1.000000  | 1.000000
0.250000,0.000000 > 1.000000  | 0.000000
```

```
0.000000,1.000000 > 1.000000   | 1.000000
0.000000,0.800000 > 1.000000   | 1.000000
0.000000,0.000000 > 0.000000   | 0.000000
1.000000,0.000000 > 1.000000   | 1.000000
0.000000,1.000000 > 1.000000   | 1.000000
0.000000,0.000000 > 0.000000   | 0.000000
Total mean square error : 0.090909
************************************************************
*** PERCEPTRON LEARN TEST END
************************************************************
```

Z výpisu je zřejmé, že síť se adaptovala velmi rychle. Nicméně je to způsobeno vlastnostmi učícího datasetu.

ZÁVĚR

Tato bakalářská práce zkoumala možnosti implementace neuronových sítí na mikrokontrolerech s omezenými výpočetními a paměťovými kapacitami, přičemž byla použita čísla s pevnou řádovou čárkou. Analýza výsledků ukázala, že tato technologie může přispět ke zlepšení schopností autonomních systémů v řadě oblastí, včetně robotiky, Internetu věcí (Internet of Thinks IoT), průmyslových a zabezpečovacích aplikací.

V teoretické části byly zkoumány různé formy implementace čísel s pevnou řádovou čárkou a koncepty různých typů neuronových sítí, což poskytlo potřebný teoretický rámec pro praktickou část práce.

V praktické části byla vytvořena knihovna v jazyce C pro základní matematické operace a datové struktury, ta byla dále využita pro implementaci vybraných typů neuronových sítí a učících algoritmů pro síť typu feedforward. Testování prokázalo funkčnost této knihovny a poskytlo základ pro její další rozšiřování a zlepšování, zejména v oblasti bezpečnosti kódu a efektivity.

Výsledky této práce naznačují, že i přes omezené výpočetní kapacity a paměť je možné na mikrokontrolerech provádět operace s neuronovými sítěmi. To otevírá nové cesty pro využití neuronových sítí v systémech s nízkou spotřebou energie, v modulárních systémech a také lze opětovně využít recyklované mikrokontrolery. Další výzkum by měl tato zjištění prohloubit a rozšířit na další typy mikrokontrolerů (zejména s jádry RISC-V) a neuronových sítí.

SEZNAM POUŽITÉ LITERATURY

[1] *754-2019 - IEEE Standard for Floating-Point Arithmetic*. IEEE Xplore [online]. New York: Electrical and Electronics Engineers, 2020, 8.5.2020 [cit. 2023-05-21]. Dostupné z: https://ieeexplore.ieee.org/document/8766229

[2] *754-1985 - IEEE Standard for Binary Floating-Point Arithmetic*. IEEE Xplore [online]. New York: Electrical and Electronics Engineers, 2020, 8.5.2020 [cit. 2023-05-21]. Dostupné z: https://ieeexplore.ieee.org/document/30711

[3] SILVA, Ivan Nunes Da, Danilo Hernane SPATTI, Rogerio ANDRADE FLAUZINO, Luisa Helena Bartocci LIBONI a Silas Franco dos REIS ALVES. *Artificial neural networks: a practical course*. Switzerland: Springer, [2016], 1 online resource (xx, 307 pages). Dostupné z: doi:9783319431628

[4] GRAUPE, Daniel. *Principles of artificial neural networks* 2nd ed. Singapore: World Scientific, 2007, xv, 303 s. Advanced series on circuits and systems. ISBN 9812706240.

[5] MANDIC, Danilo P. a Jonathon A. CHAMBERS. *Recurrent neural networks for prediction: learning algorithms, architectures and stability.* Chichester: John Wiley, c2001, xxi, 285 s. Wiley series in adaptive and learning systems for signal processing, communications, and control. ISBN 0471495174. Dostupné také z: http://www.loc.gov/catdir/toc/onix06/2001033418.html

[6] VINHAS, Adriano. *Adaline neural networks: the origins of gradient descent* Medium [online]. medium.com, 2021 [cit. 2023-05-02]. Dostupné z: https://towardsdatascience.com/adaline-neural-networks-the-origin-of-gradient-descent-783ed05d7c18

[7] BHATNAGA, Daksh. *History of CNN & its impact in the field of Artificial Intelligence* Medium [online]. medium.com, 2023 [cit. 2023-05-20]. Dostupné z: https://medium.com/international-school-of-ai-data-science/history-of-cnn-its-impact-in-the-field-of-artificial-intelligence-2b1efb7d99e5

[8] HAYKIN, Simon S. *Neural networks and learning machines. Third edition.* Upper Saddle River: Pearson, [2009], 934 s. Pearson international edition. ISBN 9780131293762.

[9] GRAUPE, Daniel. *Deep learning neural networks: design and case studies*. New Jersey: World Scientific, [2016], xvi, 263 s. ISBN 9789813146457.

[10] DRAELOS, Rachel. *The History of Convolutional Neural Networks.* Glass Box [online]. Durham, United States: Glass Box, 2019 [cit. 2023-05-10]. Dostupné z: https://glassboxmedicine.com/2019/04/13/a-short-history-of-convolutional-neural-networks/

[11] FUKUSHIMA, Kunihiko. *Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position.* Biol. Cybernetics [online]. Tokyo: Springer-Verlag, 1980, 36(36), 193-202 [cit. 2023-05-10]. Dostupné z: https://www.rctn.org/bruno/public/papers/Fukushima1980.pdf

[12] CARANDINI, Matteo. *What simple and complex cells compute. In: The Journal of Physiology* [online]. Hoboken: John Wiley, 2006, s. 463-466 [cit. 2023-05-20]. ISSN 00223751. Dostupné z: doi:10.1113/jphysiol.2006.118976

[13] DALL·E [online]. San Francisco: OpenAI, 2023 [cit. 2023-05-18]. Dostupné z: https://labs.openai.com

[14] *OpenAI* [online]. San Francisco: OpenAI, 2023 [cit. 2023-05-18]. Dostupné z: https://openai.com

[15] *Introducing LLaMA: A foundational, 65-billion-parameter large language model.* Meta AI [online]. Menlo Park: Meta Platforms, 2023 [cit. 2023-05-18]. Dostupné z: https://ai.facebook.com/blog/large-language-model-llama-meta-ai/

[16] *Google Bard* [online]. Mountain View: Alphabet, 2023 [cit. 2023-05-18]. Dostupné z: https://bard.google.com

[17] KULSHRESTHA, Ria. *Transformers.* Medium [online]. medium.com, 2020 [cit. 2023-05-18]. Dostupné z: https://towardsdatascience.com/transformers-89034557de14

[18] LEIBSON, Steve. *Adding Low-Power AI/ML Inference to Edge Devices.* EE Times [online]. Cambridge: AspenCore, 2023, 2023 [cit. 2023-05-18]. Dostupné z: https://www.eetimes.com/adding-low-power-ai-ml-interference-to-edge-devices/

[19] DUBEY, Avinava. *Constructing Transformers For Longer Sequences with Sparse Attention Methods.* Google Research [online]. Mountain View: Alphabet, 2021 [cit. 2023-05-18]. Dostupné z: https://ai.googleblog.com/2021/03/constructing-transformers-for-longer.html

[20] VOLNÁ, Eva. *Neuronové sítě 1* [online]. Ostrava, 2008 [cit. 2023-05-19]. Dostupné z: https://web.osu.cz/ Volna/Neuronove_site_skripta.pdf. Studijní materiály pro distanční kurz. Ostravská univerzita v Ostravě.

[21] COHEN, Nadav. *Optimization and Generalization for Deep Linear Neural Networks via Trajectories of Gradient Descent* [online]. Princeton, 2019 [cit. 2023-05-20]. Dostupné z: https://www.cs.princeton.edu/courses/archive/fall19/cos597B/lecnotes/linearnetsbyNC.pdf. Studijní materiály. Princeton University.

[22] MANDAL, Manav. *Introduction to Convolutional Neural Networks (CNN).* Analytics Vidhya [online]. Gurgaon: Analytics Vidhya, 2021 [cit. 2023-05-20]. Dostupné z: https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/

[23] Sebastian. *What is Pooling in a Convolutional Neural Network (CNN): Pooling Layers Explained.* Programmathically [online]. Vienna: Programmathically, 2021, 5.12.2021 [cit. 2023-05-20]. Dostupné z: https://programmathically.com/what-is-pooling-in-a-convolutional-neural-network-cnn-pooling-layers-explained/

[24] *UCI Machine Learning Repository: Occupancy Detection Data Set.* UCI Machine Learning Repository [online]. Irvine: UCI Donald Bren School of Information & Computer Sciences, 2021, 29.2.2016 [cit. 2023-05-21]. Dostupné z: https://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+

[25] TANG„ Ruixiang, Mengnan DU, Ninghao LIU, Fan YANG a Xia HU. *An Embarrassingly Simple Approach for Trojan Attack in Deep Neural Networks* [online]. Texas, 2020 [cit. 2023-05-23]. Dostupné z: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9855872. Vědecký článek. Texas A&M University.

[26] *Defining the Future of Computing – Arm®* [online]. Cambridge: Arm Limited, 2023 [cit. 2023-03-16]. Dostupné z: https://www.arm.com/

[27] DREIMER, Jeremy. *A history of ARM, part 1: Building the first chip.* Ars Technica [online]. New York City: WIRED Media Group, 2022 [cit. 2023-05-23]. Dostupné z: https://arstechnica.com/gadgets/2022/09/a-history-of-arm-part-1-building-the-first-chip/

[28] DREIMER, Jeremy. *A history of ARM, part 2: Everything starts to come together.* Ars Technica [online]. New York City: WIRED Media Group, 2022 [cit. 2023-05-23]. Dostupné z: https://arstechnica.com/gadgets/2022/11/a-history-of-arm-part-2-everything-starts-to-come-together/

[29] *Raspberry Pi Pico.* Raspberry Pi [online]. Cambridge: Raspberry Pi, 2023 [cit. 2023-03-14]. Dostupné z: https://www.raspberrypi.com/products/raspberry-pi-pico/

[30] *RP2040 Datasheet*. Raspberry Pi [online]. Cambridge: Raspberry Pi, 2023 [cit. 2023-03-14]. Dostupné z: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

[31] *Raspberry Pi 4*. Raspberry Pi [online]. Cambridge: Raspberry Pi, 2023 [cit. 2023-03-14]. Dostupné z: https://www.raspberrypi.com/products/raspberry-pi-4-model-b/

[32] C*ompute Module 4*. Raspberry Pi [online]. Cambridge: Raspberry Pi, 2023 [cit. 2023-03-14]. Dostupné z: https://www.raspberrypi.com/products/compute-module-4/?variant=raspberry-pi-cm4001000

[33] *Raspberry Pi Zero 2 W*. Raspberry Pi [online]. Cambridge: Raspberry Pi, 2023 [cit. 2023-03-14]. Dostupné z: https://www.raspberrypi.com/products/raspberry-pi-zero-2-w/

[34] *STM32L432KC*. STMicroelectronics [online]. Geneva: STMicroelectronics, STMicroelectronics [cit. 2023-03-14]. Dostupné z: https://www.st.com/en/microcontrollers-microprocessors/stm32l432kc.html

[35] *Automated Machine Learning (ML) tool for STM32 developers*. NanoEdgeAIStudio [online]. Geneva: STMicroelectronics, 2023 [cit. 2023-03-14]. Dostupné z: https://www.st.com/en/development-tools/nanoedgeaistudio.html#overview

[36] *Overview - Spresense - Sony Developer World* [online]. Tokio: Sony, 2023 [cit. 2023-03-14]. Dostupné z: https://developer.sony.com/develop/spresense/

[37] *CXD5602 User Manual* [online]. Tokio: Sony Semiconductors, 2022 [cit. 2023-05-21]. Dostupné z: https://www.sony-semicon.com/files/62/pdf/p-28_CXD5602_user_manual.pdf

[38] *Spresense Hardware Documents*. Overview - Spresense - Sony Developer World [online]. Tokio: Sony, 2023 [cit. 2023-03-14]. Dostupné z: https://developer.sony.com/develop/spresense/docs/hw_docs_en.html#_differences_between_spresense_and_arduino_uno

[39] ALLAIN, Alex. *TMS320C64x DSP Library Programmer's Reference*. Ti.com [online]. Dallas: Texas Instruments Incorporated, 2003 [cit. 2023-03-09]. Dostupné z: http://focus.ti.com/lit/ug/spru565b/spru565b.pdf

[40] DEAN, Alexander G. *Embedded systems fundamentals with ARM CortexM based microcontrollers: a practical approach*. Cambridge: ARM Education Media,[2017], xx, 292 s. ISBN 9781911531036.

[41] RAMZI, Dinda, Audra BAIHAQI, Bima PASARIBU, Anastasya AUDRYA a Dahlan SITOMPUL. *A COMPARISON BETWEEN RISC VS CISC PROCESSOR ARCHITECTURE*. ResearchGate [online]. Berlin: ResearchGate, 2021 [cit. 2023-03-20]. Dostupné z: https://www.researchgate.net/publication/352644720_A_COMPARISON_BETWEEN_RISC_VS_CISC_PROCESSOR_ARCHITECTURE

[42] FLYNN, Michael J. *Computer Architecture: Pipelined and Parallel Processor Design*. Burlington: Jones & Bartlett Publishers, 1995. ISBN 9780585251035.

[43] ALLAIN, Alex. *The C Preprocessor*. Cprogramming.com [online]. San Rafael: Cprogramming.com, 2019 [cit. 2023-02-09]. Dostupné z: https://www.cprogramming.com/tutorial/cpreprocessor.html

[44] HEATHCOTE, Jonathan. *C Pre-Processor Magic*. Jhnet [online]. Great Britain: Jhnet, 2023 [cit. 2023-02-09]. Dostupné z: Dostupné z: http://jhnet.co.uk/articles/cpp_magic

[45] *An Inline Function is As Fast As a Macro*. GCC, the GNU Compiler Collection [online]. Boston, USA: Free Software Foundation, 2007 [cit. 2023-02-13]. Dostupné z: https://gcc.gnu.org/onlinedocs/gcc/Inline.html

[46] PEDAMKAR, Priya. *Inline Function in C*. EDUCBA [online]. Maharashtra, Mumbai: EDUCBA, 2022 [cit. 2023-02-13]. Dostupné z: https://www.educba.com/inline-function-in-c/

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

AI	Artificial Inteligence
CNN	Convolution Neural Network
CPU	Central Processing Unit
DSP	Digital signal Processing
FF	Feedforward neural network
FP	Float Point
FPGA	Field Programming Gate Array
FPU	Float Point Unit
FX	Fixed Point
IDE	Integrated Development Environment
IEEE	Institute of Electrical and Electronics Engineers
I/O	Input / Output operations
IoT	Internet of Thinks
MCU	Microcontroller unit
NN	Neural Network
RISC	Reduced Instruction Set Computer
SDK	Standard Development Kit

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I. ALGORITMUS FEEDFORWARD NEURONOVÉ SÍTĚ A STRUKTURA KONFIGURACE

```
/**
* Topology structure
* [ inputs, cnl1, cnl2 ....]
* example [5,3,2,1] means
* 5 input values per perceptron
* 3 neurons in layer 1
* 2 neurons in layer 2
* last layer has only one neuron
* maxWeightsInLayer = 15
*/

typedef struct {
uint16_t *neuronCount;  // array of neurons in layer
uint16_t maxWeightsInLayer; // maximum weights in layers
uint16_t layers;  // layer counts
} nnTopology;



/**
* Fastforward neural multilayer network with logistic activation
* - tested
* @param weights
* @param inData
* @return
*/
void fpFeedForwardNetworkSig(nnTopology *topology, fpVect *weights,
   fpVect *inData, fpVect *outData) {
    uint16_t layers = topology->layers;
    uint16_t maxNeurons = topology->maxWeightsInLayer;
    float out = 0; // for safe
    // [buffer][maxNeurons]
    float Y[2][maxNeurons];
    int buffPage = 0;
    //start with input data
    float *pX = NULL;
    float *pY = Y[0];

    uint16_t index = 0;
    // layers loop
    for (int i = 0; i < layers; ++i) {
        // maxNeurons loops [i+1] -> next layer
        for (int j = 0; j < topology->neuronCount[i]; ++j) {
            // ** single neuron
```

```
            // input data loop - neuron potential
            if (i != 0) {
                    out = 0;
                    for (int k = 0; k < topology->neuronCount[i -
                       1]; ++k) {
                    out = out + weights->data[index] * pX[k];
                    ++index;
                }
                if (i == (layers - 1)) {
                    // output layer
                    outData->data[j] = (float) (1 / (1 + pow(M_E, -1
                        * out)));
                } else {
                    // hidden layer
                    pY[j] = (float) (1 / (1 + pow(M_E, -1 * out)));
                }
                // ** single neuron end
            } else {
                pY[j] = inData->data[j];
                ++index;
            }
        }

        // switch buffer - swap old output to next input
        if (i != (layers - 1)) {
            if (buffPage == 0) {
                pY = Y[1];
                pX = Y[0];
                buffPage = 1;
            } else {
                pY = Y[0];
                pX = Y[1];
                buffPage = 0;
            }
        }
    }
}
```

PŘÍLOHA P II. KOMPLETNÍ VÝPIS TESTŮ FX ČÍSEL PRO MCU RP2040

```
Program aiCore start...

Fixed Point math test...
Delay 0x end...


************************************************************
*** TIME TEST MATH
************************************************************

************************************************************
*** Multiplication
************************************************************

*** Time test multiplication [fxpMulInLine] math ...
************************************************************
A: 30.23438, B: 0.99903
Iteration count j: 1000
Last value: 11.06836
Total [ Multiplication fxpMulInLine ] time 82 uS
Average time for multiplication [fxpMulInLine]: 82


*** Time test multiplication [FXP_MUL3(C, A, B)] math ...
************************************************************
A: 30.23438, B: 0.99903
Iteration count j: 1000
Last value: 11.06836
Total [ Multiplication FXP_MUL3(C, A, B) ] time 83 uS
Average time for multiplication [FXP_MUL3(C, A, B)]: 83


*** Time test multiplication [fxpMul] math ...
************************************************************
A: 30.23438, B: 0.99903
Iteration count j: 1000
Last value: 11.06836
Total [ Multiplication fxpMul ] time 600 uS
Average time for multiplication [fxpMul]: 600


*** Time test multiplication [Float Point] math ...
************************************************************
```

```
A: 30.233999, B: 0.999000
Iteration count j: 1000
Last value: 11.117043
Total [ Multiplication Float Point ] time 660 uS
Average time for multiplication [Float Point]: 660


**********************************************************
*** Division
**********************************************************

*** Time test division [fxpDivInLine] math ...
**********************************************************
A: 30.23438, B: 1.00098
Iteration count j: 1000
Last value: 11.39161
Total [ Division fxpDivInLine ] time 351 uS
Average time for division [fxpDivInLine]: 351


*** Time test division [fxpDiv] math ...
**********************************************************
A: 30.23438, B: 1.00098
Iteration count j: 1000
Last value: 11.07911
Total [ fxpDiv ] time 746 uS
Average time for division [fxpDiv]: 746


*** Time test division [Float Point] math ...
**********************************************************
A: 30.233999, B: 1.001000
Iteration count j: 1000
Last value: 11.127662
Total [ Division Float Point ] time 649 uS
Average time for division [Float Point]: 649


**********************************************************
*** Addition
**********************************************************

*** Time test addition [fxpAddInLine] math
**********************************************************
A: 0.00000, B: 0.00098
Iteration count j: 1000
```

```
Last value: 0.97656
Total [ Addition fxpSub ] time 3 uS
Average time for addition [fxpAddInLine]: 3


*** Time test addition [fxpAdd] math
**********************************************************
A: 0.00000, B: 0.00098
Iteration count j: 1000
Last value: 0.97656
Total [ Addition fxpSub ] time 263 uS
Average time for addition [fxpAdd]: 263


*** Time test addition [Float Point]
**********************************************************
A: 0.000000, B: 0.001000
Iteration count j: 1000
Last value: 0.999991
Total [ Addition Float Point ] time 726 uS
Average time for addition [Float Point]: 726


**********************************************************
*** Subtraction
**********************************************************

*** Time test subtraction [fxpSubInLine] math
**********************************************************
A: 30.00000, B: 0.00098
Iteration count j: 1000
Last value: 29.02344
Total [ Subtraction fxpSubInLine ] time 4 uS
Average time for subtraction [fxpSubInLine]: 4


*** Time test subtraction [fxpSub] math
**********************************************************
A: 30.00000, B: 0.00098
Iteration count j: 1000
Last value: 29.02344
Total [ Subtraction fxpSub ] time 255 uS
Average time for subtraction [fxpSub]: 255


*** Time test subtraction [Float Point]
```

```
**********************************************************
A: 30.233999, B: 0.001000
Iteration count j: 1000
Last value: 29.234549
Total [ Subtraction Float Point ] time 757 uS
Average time for subtraction [Float Point]: 757


**********************************************************
*** Multiply - addition
**********************************************************

*** Time test addition [fxpMACInLine] math
**********************************************************
A: 0.90039, B: 0.00293
Iteration count j: 1000
Last value: 1.95313
Total [ MAC fxpMACInLine ] time 4 uS
Average time for addition [fxpAddInLine]: 4


*** Time test MAC [Float Point]
**********************************************************
A: 0.900000, B: 0.003000
Iteration count j: 1000
Last value: 2.700008
Total [ Addition Float Point ] time 735 uS
Average time for MAC [Float Point]: 735


**********************************************************
*** TEST math sign variations
**********************************************************

TEST TYPE ( + | + )
n.A: 1.00000
n.B: 2.00000
------------
MUL INLINE: 2.00000
SUM: 3.00000
SUB: -1.00000
DIV: 0.50000

TEST TYPE ( + | - )
n.A: 1.00000
n.B: -2.00000
```

```
------------
MUL INLINE: -2.00000
SUM: -1.00000
SUB: 3.00000
DIV: -0.49903

TEST TYPE ( - | + )
n.A: -1.00000
n.B: 2.00000
------------
MUL INLINE: -2.00000
SUM: 1.00000
SUB: -3.00000
DIV: -0.49903

TEST TYPE ( - | - )
n.A: -1.00000
n.B: -2.00000
------------
MUL INLINE: 2.00000
SUM: -3.00000
SUB: 1.00000
DIV: 0.50000

Fixed Point math end ...
```