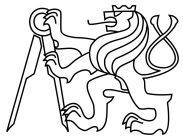
ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Výkonnostní analýza algoritmů neuronových sítí na různých počítačových systémech
Student:	Alan Dragomirecký
Vedoucí:	Ing. Miroslav Skrbek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Teoretická informatika
Katedra:	Katedra teoretické informatiky
Platnost zadání:	do konce letního semestru 2015/16

Pokyny pro vypracování

Seznamte se s neuronovými sítěmi s neurony typu Perceptron, Radial Basis Function a pulzními neurony Izhikevichovými. Pro algoritmy vybavovací fáze těchto sítí, případně jejich učení, navrhněte a implementujte výkonnostní testy a portujte je na různé počítačové platformy. Uvažujte statické implementace, jejichž zdrojové texty budou pro různé parametry, jako je počet neuronů, počet vrstev, typ neuronů a datové typy, automaticky generovány. Výsledky testů zpracujte z pohledu časových i paměťových nároků a určete limity pro velikost neuronových sítí zpracovatelných pro stanovenou dobu odezvy. Zaměřte se zejména na počítačové platformy s omezenou velikostí paměti, výkonem nebo instrukčním repertoárem. Rozsah implementační a experimentální práce stanovte po dohodě s vedoucím práce.

Seznam odborné literatury

Dodá vedoucí práce.

L.S.

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

prof.Ing. Pavel Tvrdík, CSc.
děkan

V Praze dne 26. února 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra teoretické informatiky

Bakalářská práce

Výkonnostní analýza algoritmů neuronových sítí na různých počítačových systémech

Alan Dragomirecký

Vedoucí práce: Ing. Miroslav Skrbek, Ph.D.

17. května 2016

Poděkování

Rád bych zde poděkoval vedoucímu práce Ing. Miroslavu Skrbkovi, Ph.D. za vedení této práce, věnovaný čas a cenné rady. Dále bych rád poděkoval své rodině a přátelům za jejich trpělivost a podporu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo"), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 17. května 2016 ……

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Dragomirecký, Alan. *Výkonnostní analýza algoritmů neuronových sítí na různých počítačových systémech.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato práce pojednává o návrhu a implementaci systému pro automatizované testování výkonnosti algoritmů umělých neuronových sítí. Testování je zaměřeno hlavně na vestavěné systémy. K dosažení tohoto cíle byla vyvinuta aplikace pro generování implementací umělých neuronových sítí, systémy pro jejich automatické testování a měření výkonnosti.

Klíčová slova umělé neuronové sítě, vestavěné systémy, testování, výkonnost, generování, ARM, AVR

Abstract

This thesis focuses on design and implementation of system for automated performance testing of artificial neural network algorithms. Testing is mainly focused on embedded systems. To achieve this, we have developed application which generates implementations of artificial neural networks and systems for it's automated testing.

Keywords artificial neural networks, embedded systems, testing, performance, generation, ARM, AVR

Obsah

Seznam obrázků

Seznam tabulek

Úvod

V dnešní době čím dál tím víc roste popularita oboru umělé inteligence. Jeden z důležitých výpočetních modelů v tomto oboru jsou neuronové sítě. Jejich využití můžeme sledovat u problémů, kde přesně nedokážeme popsat souvislosti mezi vstupem a očekávaným výstupem algoritmu. Příkladem může být například predikce vývoje finančních časových řad, v lékařství k diagnostice, nebo například ke zpracování vstupních signálů v robotice.

Poslední zmíněný případ už naznačuje směr této práce. Užití algoritmů umělých neuronových sítí ve vestavěných systémech začíná být s čím dál větší popularitou vestavěných systémů zajímavější. Jelikož se ale jedná o zařízení s převážně omezenými zdroji, vyvstává otázka, jaké jsou reálné limity užití těchto algoritmů. Řešení právě této otázky si klade za cíl tato práce.

Práce je rozdělena do čtyř kapitol. První z nich má za účel uvedení do teorie týkající se algoritmů neuronových sítí. Druhá kapitola se věnuje analýze možných řešení a představení praktické části této práce. Předposlední kapitola je zaměřena na implementační detaily s popisem možností rozšíření vytvořených systémů. V poslední části je pozornost věnována analýze naměřených výsledků.

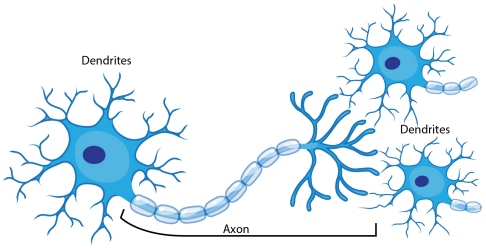
Kapitola 1

Teorie

1.1 Algoritmy umělých neuronových sítí

Hlavní inspirací umělých neuronových sítí je centrální nervová soustava živočichů, především mozek. Napřed tedy uvedeme základní principy fungování zmíněného biologického vzoru. Mozková kůra člověka se skládá odhadem z 10 miliard nervových buněk (neuronů), z nichž každý může být spojen se stovkami až desítky tisíc jiných neuronů. Strukturu neuronu máme znázorněnou na obrázku 1.1. Dendrity jsou vstupní přenosové kanály neuronu, výstupním je axon. Impulsy z ostatních neuronů jsou dendrity přeneseny do těla neuronu, čímž se změní jeho vnitřní stav. Pokud tento stav přesáhne určitou mez, neuron sám vyprodukuje impuls který putuje pomocí axonu do dalších neuronů. Spojení mezi konci axonu (terminály) a dendrity nazýváme synapse, a jsou hlavním úložištěm znalostí v síti. Procesem učení pak označujeme neustále změny synaptických propustností, jejich zanikání či vytváření. [11]

Jednou z důležitých vlastností umělých neuronových sítí (anglicky Artificial Neural Network – ANN) je jejich možnost použití u problému, kde přesně neznáme matematické vztahy mezi vstupními daty a požadovaným výstupem.



Obrázek 1.1: Ilustrace propojení neuronových buňek.
Zdroj: `https://askabiologist.asu.edu/plosable/speed-human-brain`.

K tomu abychom ale takový problém byli schopný řešit je potřeba nejdříve ANN vhodně postavit a nakonfigurovat. Toto obsahuje zvolení vhodného typu neuronů, vybrání topologie sítě (propojení neuronů), a na závěr nakonfigurování parametrů sítě.

1.2 Perceptron

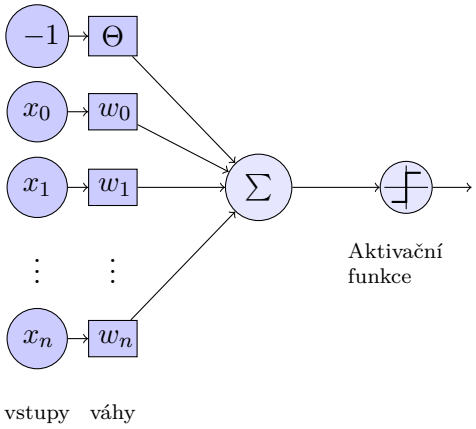
Jedním z nejstarších umělých modelů neuronu je Perceptron. Byl navržen v 50. letech 20. století Frankem Rosenblattem, a stále se jedná o jeden z nejpoužívanějších modelů. Podobně jako skutečný neuron, perceptron má n vstupů a právě jeden výstup. Vnitřní potenciál ξ vypočítáme jako skalární součin vektoru vstupů a vektoru vah, od kterého odečteme práh Θ neuronu. Výstupem neuronu je vnitřní potenciál na který aplikujeme tzv. aktivační funkci σ. [11]

$$\xi = \vec{x} \cdot \vec{w} - \Theta \tag{1.1}$$

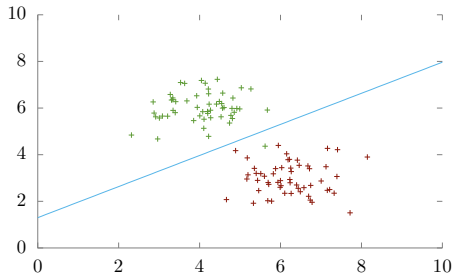
$$y = \sigma(\xi) \tag{1.2}$$

Aktivační funkce může být různého druhu. Nejjednodušší variantou je Heavisideova funkce nebo funkce signum. Do skupiny nejpoužívanějších spojitých funkcí pak patří sigmoidální funkce. Volba aktivační funkce závisí na povaze vstupních (a výstupních) dat.

Funkce perceptronu se dá popsat poměrně jednoduše. Pokud máme nějaký prostor vstupních vektorů o dimenzi n, pak vektor vah s prahem neuronu nám určuje jednoznačnou nadrovinu v tomto vstupním prostoru. Signum vnitřního potenciálu pak určuje do jakého poloprostoru daný vstup spadá (pro hodnoty $sgn(\xi) \neq 0$).



Obrázek 1.2: Struktura perceptronu. Pro zjednodušení výpočtu můžeme práh reprezentovat jako váhu imaginárního vstupu s hodnotou −1.



Obrázek 1.3: Ilustrace funkce perceptronu. Vstupní data představují dva shluky, které chceme separovat.

1.3 RBF neuron

Radiální bázické funkce (Radial Bases Function) jsou funkce jejichž fukční hodnota závisí pouze na vzdálenosti argumentu od nějakého určitého středu. Často používanou metrikou je euklidova vzdálenost. Pro představu, pokud v dvourozměrném vstupním prostoru definujeme bod s, pak všechny body ležící na kružnici o poloměru r se středem v s budou mít stejnou funkční hodnotu (při použití euklidovy vzdálenosti).

RBF neuron je strukturou velmi podobný perceptronu, také definuje pro každý vstup určitou váhu a jeden koeficient navíc, který v tomto případě nebudeme nazývat *prahem*, ale *šířkou*. Vektor vah ale narozdíl od perceptronu bude určovat střed pro již zmíněnou radiální bázickou funkci. Argumentem této funkce pak bude vektor vstupů neuronu.[7, 9]

Pro neuron s vahami (středem) $\vec{w}$, bázickou funkcí d a šířkou b můžeme pro vstup $\vec{x}$ definovat vnitřní potenciál jako:

$$\xi = \frac{d(\vec{x}, \vec{w})}{b}. \tag{1.3}$$

Výstup neuronu je pak definován stejně jako u perceptronu v závislosti na vnitřním potenciálu ξ a aktivační funkci σ:

$$y = \sigma(\xi). \tag{1.4}$$

Základní aktivační funkcí, kterou v této práci budeme používat, je Gaussova:

$$y = e^{-\frac{\xi}{\beta}^2}. \tag{1.5}$$

Narozdíl tedy od perceptronu, který vstupní prostor rozděluje na dva poloprostory, RBF neuron má určený nějaký vektor v prostoru, který můžeme nazvat třeba vzorem. Výstup neuronu pak reprezentuje vzdálenost mezi tímto vzorem a vstupním vektorem.

1.4 Izhikevich

Výše probrané modely neuronů jsou inspirovány biologickým neuronem, jedná se ale o velmi zjednodušené modely které mají k chovaní skutečných neuronů velmi daleko. Mladší kategorií neuronů, které jsou svým chováním o něco blíže, jsou pulsní neurony. Jejich hlavní odlišností je zavedení času do výpočtu. Pulsní neurony zároveň (jak už název napovídá) produkují pulsy, a to v závislosti na čase, jejich vnitřním stavu a vstupu. Tímto se mění potřeba jakým způsobem kódovat vstupní a výstupní informace. Narozdíl od perceptronu, kde můžeme např. vstupní data jednoduše zakódovat do binární podoby a „vyslat do sítě", v pulsní síti používáme posloupnost pulsů v čase jako nosič informace. Používané kódování jsou například Time-To-First-Spike, Phase, Spike Rate a další.

Existuje více typů pulsních neuronů. Například biologicky poměrně věrohodný model *Hodgkin-Huxly*, který je ale relativně výpočetně náročný. Tento nedostatek nemá *Integrate-And-Fire* model, u kterého ale na druhou stranu postrádáme komplexnější simulační schopnosti. Zajímavou možností je pak model *Izhikevich*, který kombinuje přednosti modelů zmíněných výše. Je relativně výpočetně nenáročný a zároveň poskytuje dobré výsledky. Funkci tohoto modelu si můžeme definovat jako obyčejné diferencionální rovnice

$$\frac{\mathrm{d}v}{\mathrm{d}t} = 0.04v^2 + 5v + 140 - u + I \tag{1.6}$$

$$\frac{\mathrm{d}u}{\mathrm{d}t} = a(bv - u) \tag{1.7}$$

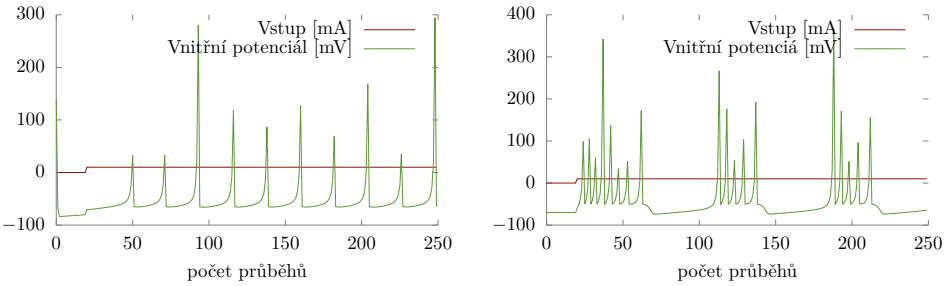
kde v reprezentuje vnitřní potenciál neuronu, u obnovovací fázi, t čas a a, b, c, d jsou parametry definující chování neuronu. Různé chování neuronu můžeme vidět na grafech 1.4. Vnitřní potenciál v je v tomto případě výstupem neuronu. Navíc je ale potřeba po každém vygenerovaném pulsu „vyresetovat" neuron, což provedeme podle 1.8.

$$\text{if } v \geq 30 \text{ then } \begin{cases} v \leftarrow c \\ u \leftarrow u + d \end{cases} \tag{1.8}$$

1.5 Topologie sítě

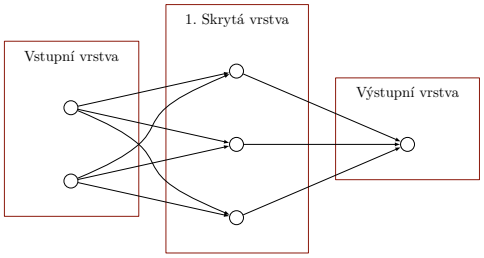
Pokud se bavíme o topologii neuronové sítě, máme na mysli různé propojení neuronů mezi sebou, označení určitých neuronů jako *vstupních*, případně *výstupních*. Zvolená topologie sítě se z velké části odvíjí od použitého typu neuronu.

Pro perceptrony a RBF se typicky používá topologie Feed-Forward. V té jsou neurony organizovány do série vrstev a informace vždy putuje z jedné vrstvy do druhé. První vrstvu nazýváme vstupní, a slouží pouze k vložení



(a) $a = 0.02; b = 0.2; c = -65; d = 2.0$ (b) $a = 0.02; b = 0.2; c = -50; d = 2.0$

Obrázek 1.4: Chování neuronu Izhikevich s různým nastavením.



Obrázek 1.5: Ilustrace topologie feed-forward sítě s jednou skrytou vrstvou.

vstupních dat do neuronové sítě. Po ní může následovat n vrstev skrytých, které s poslední výstupní vrstvou představují samotnou výpočetní sílu neuronové sítě. Vektor výstupů poslední vrstvy pak považujem za výstup sítě.

Je potřeba zmínit, že pod označením „perceptron“ se většinou nerozumí pouze model neuronu, ale používá se i pro popis feed-forward sítě, která má jednu vstupní a jednu výstupní vrstvu (a zároveň používá pouze perceptron neurony). Pokud taková síť obsahuje nějakou skrytou vrstvu, pak mluvíme o vícevrstvém perceptronu.

Stejně tak pokud mluvíme o síti RBF, většinou tím rozumíme síť, která má jednu skrytou vrstvu obsahující RBF neurony a ve výstupní vrstvě neurony provádějící vážený součet (takové neurony můžou být i perceptrony s váhou rovnou nule a identickou funkcí použitou jako aktivační). Každý neuron v takové síti pak představuje nějaký střed ve vstupním prostoru a výstupní vrstva přiřazuje těmto středům různou váhu.

1.6 Rekurentní sítě

Za rekurentní považujeme takovou síť, která obsahuje alespoň jednu zpětnou vazbu. Výpočetní síla tohoto modelu je vyšší než u feed-forward topologie, jelikož se dá říci že zpětnou vazbou přidáváme do sítě *paměť*. Výstup rekurentní sítě tedy nemusí být závislý pouze na aktuálním vstupu, ale i na posloupnosti

vstupů předchozích. Ačkoliv jsou rekuretní sítě sílnějším výpočetním modelem, jejich učení je obtížnější než u feed-forward sítí. [9]

1.7 Učení sítě

Ačkoliv algoritmy učení neuronové sítě přesahují praktickou část této práce, uvedeme o co se jedná pro obsáhnutí základních pojmů týkajících se neuronových sítí. Cílem učení neuronové sítě je nastavit parametry neuronů tak, aby síť podávala chtěné výsledky. Učící algoritmy dělíme na učení s učitelem a bez učitele.

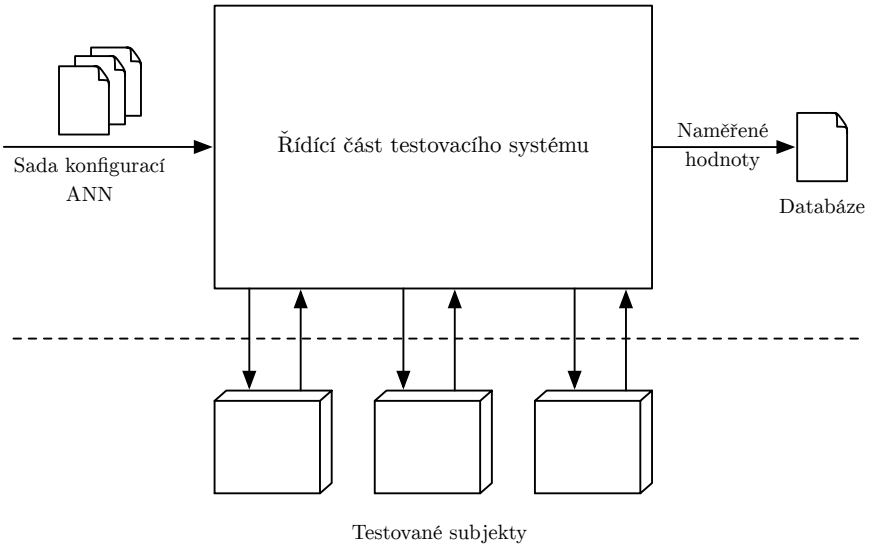
Při učení s učitelem vycházíme z toho, že máme k dispozici vzorová data (respektive vzorové vstupy a k ním očekávané výstupy). Jedním z nejpoužívanějších algoritmů učení s učitelem [11] je algoritmus zpětného šíření chyby (backpropagation). Tento algoritmus postupně předkládá vzorová vstupní data, a následně podle odchylky od vzorových výstupních dat upravuje váhy neuronů.

U algoritmů učení bez učitele nemáme k dispozici žádné informace, podle kterých bychom byli schopni ohodnotit „správnost“ výstupu sítě. Tyto algoritmy se tedy zaměřují pouze na hledání shluků ve vstupních datech. Do této skupiny algoritmů patří například Kohenovy mapy. [10]

Kapitola 2

Analýza a návrh

V této kapitole se zaměříme na vysokoúrovňový pohled na celý systém. Z jakých komponent je složen, jak fungují a jaké a jaké důvody vedly k rozhodnutím učiněných v této práci.



Obrázek 2.1: Základní pohled na celý systém. Vstupem je sada konfigurací neuronových sítí. Ty jsou automatizovaně otestovány na všech subjektech a výsledky jsou následně uloženy do lokální databáze.

2.1 Výběr testovaných systémů

Pro vysvětlení volby mikrokontrolerů které jsou používány k testování v této práci se napřed zaměříme na to, jakou informaci očekáváme že nám podají samotné výsledky. V dnešní době získává čím dál tím větší pozornost poměrně

nové odvětví vestavných systému nazývané Internet věcí (anglicky *Internet Of Things*, zkratka IoT). Do této kategorie můžeme zařadit například žárovky ovladané přes internet, osobní váhy které zaznamenávají naměřené informace do cloudu (internetového uložiště) apod. Další zajímavou kategorií zařízení jsou různé roboty, koptery a další. Všechny tyto zařízení mají společnou věc, a to že se jedná o vestavěné systémy s omezenou výpočetní silou a důrazem kladeným na spotřebu elektrické energie. Zároveň ale jsou to zařízení, u kterých se dá očekávat zajímavé využití umělých neuronových sítí (a to například pro zpracování dat z různých sensorů). Rozhodli jsme se tedy zaměřit na tuto kategorii zařízení.

Nejpoužívanější architekturou ve vestavných systémech je aktuálně architektura ARM. Jak už z jejího původního názvu *Advanced RISC Machine* vyplývá, jedná se o architekturu s omezenou sadou instrukcí (respektive s malým počtem univerzálních a velice rychle proveditelných instrukcí). Podkategorií ARM na kterou jsme se rozhlodli zaměřit je ARM Cortex-M. Tato rodina 32 bitových mikrokontrolerů je speciálně zaměřená na nízkou spotřebu energie a využití v IoT. Z této skupiny jsme pro testování vybrali následující mikrokontrolery:

ARM Cortex-M0 Nejslabší a zároveň nejúspornější 32 bitový mikrokontroler z rodiny ARM. Pro testování jsme použili vývojovou desku *NRF51 Dongle* od společnosti Nordic osazenou tímto mikrokontrolerem.

ARM Cortex-M3 Cortex-M3 by se dal nazvat střední cestou této rodiny mikrokontrolerů. Při stále velmi úsporné spotřebě energie již nabízí kompletní instrukční sadu Thumb-2 (Cortex-M0 implementuje pouze podmnožinu této sady), více paměti apod. K testování toho mikrokontroleru jsme použili vývojovou desku Nucleo STM32F103RB.

ARM Cortex-M4 Tento mikrokontroler je speciálně navržen aby byl vhodný pro zpracování signálů a podobných matematických operací. Narozdíl od mikrokontrolerů zmíněných výše již tedy obsahuje matematický koprocesor (anglicky Floating Point Unit, zkráceně FPU) a dále třeba instrukční sadu DSP. Pro testování byla použita deska STM32F429I – DISCO.

Dále jsme se rozhodli otestovat určité mikrokontrolery z rodiny AVR. AVR je označení pro osmi-bitové mikrokontroléry společnosti Atmel, které jsou velice populární pro jejich jednoduchost. Jejich použití pak můžeme nejvíce vidět u nenáročných vestavěných systémů, a také jsou velice známé díky platformě Arduino, která je využívá. Jako zástupce pro testování jsme vybrali:

ATMega328P ATMega328p reprezentuje papírově nejslabší mikrokontrolér který budeme testovat. Důvodem k jeho výběru je jeho využití na desce Arduino Uno, kterou můžeme považovat za základní model této platformy.

ATMega2560 Jedná se o nejsilnější mikrokontrolér z rodiny osmi-bitů od Atmelu. Jeho výsledky tak budou představovat maximum co můžeme od této skupiny očekávat. Pro testování jsme použili desku vyrobenou na ČVUT.

Konkrétní konfigurace a parametry testovaných vývojových desek jsou uvedeny v kapitole 4.5.

2.2 Volba programovacího jazyka C

Je zřejmé, že je nutné zvolit jazyk ve kterém bude implementovaná jak samotná neuronová síť, tak nutné prostředí ve kterém bude testována. Jelikož testování bude probíhat na vestavných systémech s velice omezenými zdroji, a zároveň je našim cílem testování limitů těchto zařízení, je nutné zvolit nějaký nízkoúrovňový jazyk s kterým je možné využít celý potenciál daných zařízení. Teoreticky bychom byli schopni dosáhnout nejlepších výsledků při použití assembleru. Zápory by ale v takovém případě převažovaly klady. Kód v assembleru je například vázaný na specifika mikrokontroleru (jeho instrukční sadu, počet registrů…), což by vyžadovalo rozdílnou implementaci pro každou platformu. Dalším neméně podstatným argumentem může být vysoká implementační náročnost takového řešení.

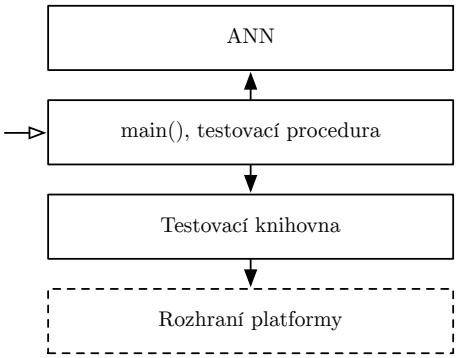
Jasným adeptem pro implementaci je tedy buď jazyk C nebo jazyk C++, které jsou v odvětví vestavěných systému nepsaným standardem. Oba jazyky nabýzejí vytvoření multiplatformního zdrojového kódu a zároveň je možné v nich vytvořit vysoce optimalizovaný kód pro danou platformu. V této práci je použit převážně jazyk C z důvodu jeho jednoduchosti, díky čemuž je snazší předpovídat zkompilovaný kód, jeho paměťovou náročnost apod.

2.3 Architektura testovací aplikace

V této části si ukážeme základní strukturu aplikace určené k měření časové náročnosti určité neuronové sítě (tedy aplikace běžící na cílovém zařízení které chceme testovat). Následně projdeme podrobněji komponenty z kterých se skládá a jaké jsou jejich zodpovědnosti.

Jak můžeme vidět na obrázku 2.2, aplikace je rozdělená na 4 komponenty.

1. První komponenou je samotná implementace konkrétní neuronové sítě, jejíž rozhraní tvoří tři funkce. První z nich vykonává jeden průběh sítě (vstupem funkce jsou tedy vstupy ANN, a výstupem je výstup ANN). Další dvě funkce slouží pro testování. Jedna z nich kontroluje zda se implementace sítě chová tak jak je od ní očekáváno (anglicky se tento typ testu označuje jako unit-test). Druhá z nich je určená k výkonostním testům – obsahuje proceduru na které má být provedeno testování dané



Obrázek 2.2: Struktura aplikace pro testování výkonnosti neuronové sítě.

implementace ANN. Například se může jednat o jedno zavolání funkce ANN pro změření její odezvy (jak dlouho trvá vypočítat výstup pro nějaké vstupy sítě).

2. Druhá komponenta představuje samotný vstupní bod aplikace. Pří jejím spuštění tato komponenta napřed spustí unit-testy dané ANN implementace. Pokud tyto testy nevrátí chybu, tak se pokračuje spuštěním výkonostních testů a měření jejich času. Poslední zodpovědností této komponenty je všechny výsledky a úspěchy či neúspěchy určitým způsobem ohlašovat.

3. U popisu předchozí „hlavní“ komponenty jsme neuvedli, jakým způsobem dochází k měření času trvání výkonnostního testu a také jakým způsobem dochází k ohlášení výsledků. K těmto účelům slouží tato komponenta, kterou můžeme nazva „Testovací knihovna“. Jejím učelem je tedy zapouzdření určitých platformě závislých operací vyžadovaných pro testování. Těmito operacemi myslíme:

   Textový výstup aplikace Při testování v prostředí operačního systému může být implementováno jako standardní výstup aplikace. U vestavěných systémů se jedná typicky sériovou linku.

   Jednoduché hlášení úspěchu / neúspěchu Analogicky k předchozímu, může se jednat o návratový status aplikace (v případě běhu v rámci operačního systému), nebo u vestavěného systému rozsvícení LED a vytisknutí chyby na sériovou linku.

   Měření času Poslední důležitou zodpovědností této komponenty je samotné měření času. Každá platforma musí tuto funkci implementovat způsobem nabízející dostatečné rozlišení pro měření.

Tato komponenta tedy představuje univerzální rozhraní zapouzdřující operace, které testovací aplikace vyžaduje. Její implementace je zároveň jedinou částí námi psaného kódu, která je platformě závislá.

4. Poslední komponenta „Rozhraní platformy“ je zde uvedena pro znázornění závislostí v návrhu. Na této části je závislá pouze „Testovací knihovna“, která ji využívá pro implementaci svých funkcí.

2.4 Generování statických implementací ANN

V této kapitole se seznámíme s požadavky, které vedly k volbě vytvoření vlastního systému generování statických implementací umělých neuronových sítí, a jeho základními vlastnostmi.

2.4.1 Motivace

Je zřejmé, že pro testování algoritmů ANN musíme zvolit nějakou implementaci na které budeme testy provádět. Zde máme několik možností volby. Jedna z nich může být použití C knihovny pro práci s neuronovými sítěmi, a testy provádět na ní. Naprostá většina takových knihoven ale vytváří strukturu sítě až za běhu programu, což může být velice omezujícím faktem při použití na zařízení jako je například Arduino Uno, které má 2 kB paměti RAM. V takovém případě je lepší varianta použít takovou implementaci ANN, kde je maximum věcí definováno v době kompilace, a v době spuštění programu je již vše připravené a není potřeba například alokovat další paměť. Takového výsledku by se teoreticky dalo dosáhnout pouze pomocí jazyka C a jeho maker, práce s takovými zdrojovými kódy by ale nebyla vůbec uživatelsky přívětivá. Proto jsme se rozhodli pro vytvoření systému, který pro zadané parametry neuronové sítě vygeneruje zdrojové kódy, které ji implementují. Tento systém nám dovoluje vygenerovat takové zdrojové kódy, které mají minimální možné nároky na paměť.

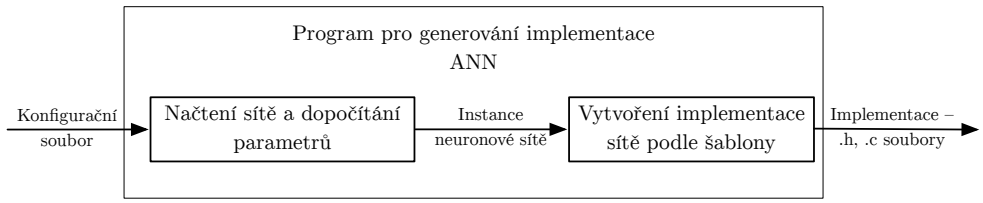
2.4.2 Použité nástroje

Pro vytvoření toho generátoru jsme se rozhodli použít skriptovací jazyk *Python*. Python je interpretovaný[1] jazyk používaný v mnoha odvětví IT. Důvodem pro jeho volbu byla především jeho jednoduchost a efektivita vývoje.

K samotnému vytvoření zdrojového kódu jsme použili šablonovací systém *Jinja*.

[1]Python se obecně označuje za interpretovaný jazyk, ve skutečnosti ale většina jeho implementací kompiluje kód do bytekódu a ten následně spouští ve virtuálním stroji. Obdobným způsobem jsou implementovány například jazyky Java a C#, které bychom za interpretované určitě neoznačili.

2.4.3 Proces generování sítě



Obrázek 2.3: Znázornění procesu generování sítě

Jak už bylo zmíněno, vstupem pro generátor je nějaká konfigurace ANN a výstupem je implementace dané sítě v jazyce C (jak můžeme vidět na obrázku 2.3). Generátor nejprve přečte zadaný konfigurační soubor, a následně dopočítá potřebné informace pro vytvoření požadované implementace ANN (může se jednat například o generování náhodných vah, dosazení výchozích hodnot pokud nejsou uvedeny v konfiguraci apod.).

Následně jsou všechny tyto informace vloženy do kontextu šablonovacího systému a v tomto kontextu se vytvoří implementace ANN pomocí příslušných šablon. Podrobněji je tento proces probrán v kapitole 3.1.

2.4.4 Konfigurace ANN

Konfigurace pro vygenerování určité ANN se skládá z několika částí. První a nejpodstatnější z nich je samotná konfigurace struktury síťě, typů neuronů, nastavení vah apod. Pro tuto práci jsme zvolili implementovat podporu pro feed-forward topologii, díky které jsme následně schopni implementovat jak (vícevrstvý) perceptron, tak například běžně používané RBF sítě. V rámci konfigurace tedy specifikujeme síť jako **pole vrstev**. Pro každou vrstvu pak zvlášť specifikujeme **typ neuronů** v ní použitých, jejich **aktivační funkci**, **pole vah** a další potřebné koeficienty v závislosti na typu použitých neuronů. Další části konfiguračního souboru pak umožňují specifikovat různé implementační detaily (například použité datové typy) a dodatečné testy pro síť. Příklad konfigurace sítě implementující funkci XOR je vidět na ukázce níže.

Každá definice vrstvy (kromě vstupní) musí splňovat následující náležitosti:

- Definovat `name` – jméno vrstvy. Slouží jak k zpřehlednění konfigurace, tak pro pojmenování symbolů v implementaci souvisejících s touto vrstvou.
- Definovat `neurons_count` – počet neuronů ve vrstvě.
- Definovat `type` - typ použitých neuronů. Podporované jsou tři typy: *perceptron*, *rbf* a *izhikevich*.

```yaml
name: ann
layers:
    - name: input_layer
      neurons_count: 2

    - name: hidden_layer
      type: perceptron
      neurons_count: 2
      thresholds: [1, 1]
      weights: [[0.6, 0.6], [1.1, 1.1]]
      activation_function: perceptron_neuron_activate

    - name: output_layer
      type: perceptron
      neurons_count: 1
      thresholds: [1]
      weights: [[-2, 1.1]]
      activation_function: perceptron_neuron_activate
code:
    filename: ann_imp
    number_type: float
    size_type: int
    printf: wr_uart_printf
    include:
        - wr_uart.h
tests:
    - { input: [0,0], output: [0] }
    - { input: [1,0], output: [1] }
    - { input: [0,1], output: [1] }
    - { input: [1,1], output: [0] }
performance-tests:
    - { input: [0,0] }
```

Ukázka kódu 1: Konfigurace pro síť typu perceptron implementující funkci XOR.

- Definovat `weights` - váhy vstupů neuronů. Tato hodnota musí být dvourozměrné pole o velikost $N \times M$, kde N je počet neuronů v právě definované vrstvě a M počet neuronů ve vrstvě předchozí.

- Pro vrstvy typu *perceptron* a *rbf* definovat `activation_function` – jméno aktivační funkce. Je možné použít již připravenou funkci *perceptron_neuron_activate* chovající se jako Heavisideova funkce nebo `rbf_neuron_activate` implementující Gaussovu funkci. Jak použít jinou aktivační funkci je popsáno v kapitole 3.1.

- Pro vrstvy typu *perceptron* je nutné definovat `thresholds` – prahy neuronů. Hodnotou musí být jednorozměrné pole o velikosti počtu neuronů ve vrstvě.

- Pro vrstvy typu *rbf* je nutné definovat `betas`. Podobně jako u perceptronu, hodnotou musí být jednorozměrné pole o velikosti počtu neu-

ronů ve vrstvě. Tyto hodnoty odpovídají proměnné b ve vzorci 1.3 na straně 5.

- Pro vrstvy typu *izhikevich* je nutné definovat konstanty `a`, `b`, `c` a `d` (jejich použití je blíže popsáno v kapitole 1.4). Hodnoty těchto proměnných jsou sdílené mezi všemi neurony v dané vrstvě.

2.4.4.1 Rekurentní vrstvy

Další funkcí generátoru, kterou jsme ještě nezmínili, je podpora pro generování implementace ANN s nějakými rekurentními vrstvami. Vrstvu můžeme udělat rekurentní pomocí přidaní `recurrent: true` do konfigurace vrstvy. Vstupem pro takovouto vrstvu pak není pouze výstup vrstvy předchozí, ale jeho spojení s výstupem právě definované vrstvy z minulého průběhu sítě. Tento fakt je nutný vzít v potaz například při definování vah vstupů neuronů. Rozměry pole `weights` tedy v tomto případě budou $N \times M$ kde N je počet neuronů v této vrstvě a M je součet N a počtu neuronů v předchozí vrstvě. Příklad konfigurace sítě s rekurentní vrstvou je na stránce 16.

```
2  layers:
3      - name: input_layer
4        neurons_count: 1
5
6      - name: rec_layer
7        type: perceptron
8        recurrent: true
9        neurons_count: 2
10       thresholds: [0.5, 0.5]
11       weights: [[1, 0, 0], [0, 1, 0]]
12       activation_function: perceptron_neuron_activate
13
14     - name: output_layer
15       type: perceptron
16       neurons_count: 1
17       thresholds: [0.5]
18       weights: [[0, 1]]
19       activation_function: perceptron_neuron_activate
```

Ukázka kódu 2: Konfigurace rekurentního vícevrstvého perceptronu. Výstup této sítě je *předchozí* vstup sítě (pro vstupy 0 a 1).

2.4.4.2 Možnosti vygenerovaného zdrojového kódu

Konfigurační soubor mimo samotné struktury umělé neuronové sítě obsahuje i další definice. Jednou z nich je sekce `code` začínající na řádce 19 v ukázke kódu 1 na straně 15. V této části konfiguračního kódu je možnost definovat různé implementační detaily týkající se vygenerované implementace ANN.

- Jedinou povinnou položkou v této části je `filename` – hodnotou musí být řetězec, kterým budou začínat jména vygenerovaných souborů implementace. Pokud specifikujeme například „network“, pak budou vytvořeny soubory `network.h`, `network.c` a `network_tests.c`. Soubor `network.h` obsahuje rozhraní pro práci se sítí. Samotná implementace sítě se nachází v souboru `network.c`. Implementace testů je oddělena od implementace sítě v souboru `network_tests.c`.

- Další možností je specifikovat `number_type` a `size_type`. Tyto položky ovlivňují, jaké datové typy budou použity ve vygenerované implementaci ANN. `number_type` je použit pro vstupy sítě, výstupy, koeficienty, konstanty apod. `size_type` je pak použit pro indexování, definici velikostí vektorů atd.

  Výchozí hodnotou pro `number_type` je `float`, a pro `size_type` `int`.

- Vygenerované implementace unit-testů poskytují textový výstup informující o úspěchu/neúspěchu. Ve výchozím stavu je pro tento výstup použita standardní funkce `printf`, je ale možné specifikovat vlastní funkci. Předpokládejme například, že chceme použít vlastní funkci `uart_printf`, která vytiskne data přes sériovou linku. Tohoto můžeme docílit pomocí specifikováním `printf: uart_printf`.

- Poslední (nepovinnou) možností je položka `include`. Její hodnotou musí být pole jmen souborů. Vygenerovaná implementace pak bude obsahovat direktivu preprocesoru `#include` pro specifikované soubory. Toto se hodí například při použití vlastní aktivační funkce.

2.4.4.3 Specifikace testů

Jak už jsme zmínili dříve, jednou z možností konfiguračního souboru je specifikace jednoduchých unit-testů a „výkonnostních testů“.

Definice jednoduchých unit-testů umožňuje ověřit zda se implementace sítě chová jak od ní očekáváme. Příklad využití je vidět na ukázkové konfiguraci na stránce 15 (řádka 26). Testy jsou definovány jako pole vstupů a k ním přiřazeným výstupům. Generátor pak vytvoří jednu funkci (v tomto případě bude deklarována jako `int ann_test()`. Její návratovou hodnotou je počet testů kde se skutečný výstup sítě nerovnal výstupu deklarovanému v konfiguračním souboru.

Druhý typ testu, v konfiguračním souboru označený jako „performance-tests“, sám o sobě nic netestuje. Jedná se spíše o nástroj, který nám později pomůže v procesu testování výkonnosti sítě. V konfiguraci `performance-tests` pouze specifikujeme pole vstupů pro danou neuronovou síť. Generátor následně vygeneruje jednoduchou funkci, která postupně spouští tuto neuronovou síť s danými vstupy. Díky tomuto systému pak kód, který například měří jak dlouho trvá průběh různých implementací neuronových sítí nemusí mít

žádnou znalost o tom, jaký vstup je potřeba na kterou síť při testování aplikovat – pouze změří průběh této vygenerované funkce.

2.4.4.4 Dynamické hodnoty v konfiguraci

Způsob konfigurace, který jsme doteď popsali, je vhodný pro umělé neuronové sítě malého rozsahu. Jelikož se ale v této práci chceme věnovat testování limitů různých zařízení, je zcela zřejmé že takových limitů u většiny zařízení nedosáhneme za pomoci konfigurace perceptronu s pěti neurony (jako například v ukázce kódu 1). Pro tyto potřeby bude nutné vytvořit konfigurace s řádově minimálně stovky neuronů a v takovém případě by bylo krajně nepraktické obsáhnout v konfiguračním souboru váhy pro všechny neurony, jejich další koeficienty apod.

Z tohoto důvodu je možné v konfiguraci specifikovat různé hodnoty „dynamicky“ za pomocí python výrazů. Zaměřme se například na specifikaci vah pro vstupy neuronů. Zde je vždy vyžadováno dvourozměrné pole o konkrétní velikosti. Základní možností je takové pole přímo vypsat do konfiguračního souboru, jak jsme ale zmínili, u větších sítí by toto pole mohlo nabývat enormních rozměrů a spravovat takovou konfiguraci by bylo velmi obtížné.

Vhodnějším řešením je v takovém případě specifikovat takové hodnoty jako python výraz, který je následně při interpretaci konfiguračního souboru použit pro vytvoření požadovaného parametru sítě. Jako python výraz je interpretovaná každá posloupnost znaků (string) na místě, kde je v konfiguraci očekáváno pole hodnot.

Například jednoduchá varianta definice vah může vypadat takto:

```
weights: [[0, 1], [1, 2], [0, 0]]
```

Ekvivalentě k tomu, s použitím python výrazu:

```
weights: "[[0, 1], [1, 2], [0, 0]]"
```

Příklad výše je pouze demonstrace nedávající praktický smysl. Zajímavější využití může být například načtení vah ze souboru:

```
weights: json.load(open('weights.json'))
```

Nebo automatické vygenerování náhodných vah:

```
# kontext ve kterém jsou výrazy vyhodnocovány
#   má importované všechny objekty z modulu random
weights: "[[random(), random()], [random(), random()], [random(), random()]]"
```

Tento příklad s generováním náhodných vah je pro potřeby této práce velmi užitečný. Na malých sítích ručně nakonfigurovaných napřed ověříme (pomocí testů) zda se implementace chová jak očekáváme a poté na řádově větších sítích s náhodnými váhami budeme testovat jejich rychlost. Předchozí příklad

s definicí náhodných vah je ale stále zbytečně „upovídaný“. Pro zjednodušení konfigurace je proto možné váhy specifikovat následujícím způsobem:

```
weights: random()
```

V tomto případě generátor rozpozná že hodnota tohoto výrazu nemá potřebnou dimenzi, a použije tento výraz pro vygenerování příslušného dvourozměrného pole. Konfigurace nějaké vrstvy pak může vypadat například takto:

```
1     - name: rbf_layer
2       type: rbf
3       neurons_count: 3
4       betas: json.load(open('betas.json'))
5       weights: uniform(-1, 1)
6       activation_function: rbf_neuron_activate
```

2.5 Build systém

Důležitou součástí celého testovacího prostředí je build systém – tedy systém, který má na starosti koordinaci a kompilovaní zdrojových kódu do spustitelné podoby pro daná zařízení.

Pro tuto práci jsme zvolili kolekci kompilátorů GCC (GNU Compiler Collection), spolu s dalšími nástroji spadající do GNU Toolchainu[2]. Jedním z důvodů je velká rozšířenost GCC, takže obecně není problém sehnat verzi pro jakoukoliv platformu. Použité distribuce v této práci jsou hlavně *GCC ARM Embedded* pro mikrokontrolery s ARM architekturou a *AVR GCC* pro osmibitové mikrokontroléry společnosti Atmel.

Pro samostatnou koordinaci procesu kompilace jsme použili program *Make* spolu s vlastním nástrojem pro řešení závislostí mezi zdrojovými kódy. Ačkoliv by se náš problém kompilace zdrojových kódů dal řešit jedním velkým zdrojovým souborem programu Make, velice rychle bychom narazili na problémy se správou takového souboru, a rozhodně by se nejednalo o pěkné řešení. Zároveň je možné použít nástroje které tento problém přímo řeší (například CMake, GNU Autotools apod). V našem případě jsme se ale rozhodli jít cestou pouze nástroje Make s malým skriptem k němu. Výhodou tohoto řešení je naprostá „průhlednost“ nastavení celého procesu kompilace, což je důležité pro validitu výsledků měření. Jak je systém navržený si podrobněji představíme v následujících odstavcích.

2.5.1 Požadavky

Pro začátek je dobré si rozlišit problémy, které v našem případě chceme řešit. Pokud bychom například náš projekt testovací aplikace chtěli zkompilovat a následně spustit na nějaké vývojové desce, jedny z věcí na které narazíme jsou:

[2]Označení pro nástroje k programování vytvořené v rámci projektu GNU.

- jaké **zdrojové soubory** jsou součástí projektu,
- v jakých složkách hledat **hlavičkové soubory** (anglicky header files),
- s jakými **přepínači** projekt zkompilovat,
- jak provést samotnou **kompilaci** se znalostmi z bodů výše,
- a následně jak produkt kompilace **nahrát a spustit** na dané vývojové desce.

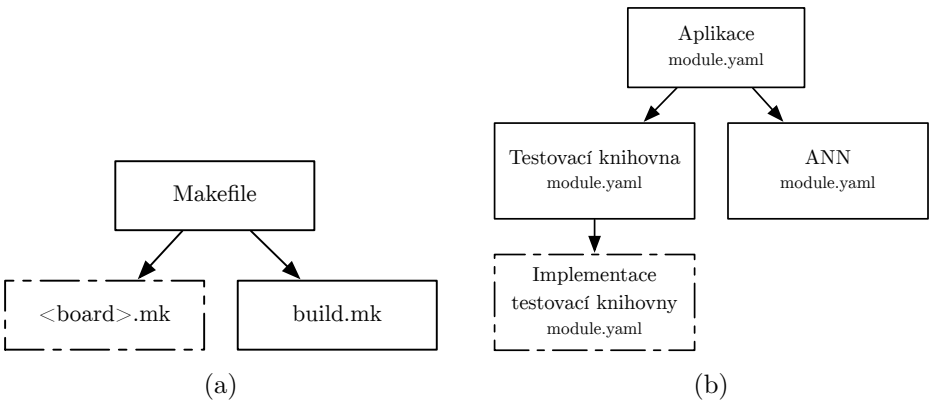
Pro utvoření představy o řešení je navíc dobré připomenout, že se aplikace sestává z několika modulů (moduly myslíme například implementaci ANN, knihovnu zapouzdřující funkce nutné pro testování, samotnou testovací aplikaci atd.).

Můžeme tedy předpokládat, že zdrojové soubory a cesty pro hledání hlavičkových souborů budou definovány v rámci modulů. V rámci modulů může být také vhodné specifikovat definice různých maker ovlivňující následnou kompilaci.

Dalším cílem je oddělit z procesu kompilace problémy, které jsou závislé na cílovém zařízení, pro které projekt kompilujeme. Tím je především celá sada přepínačů pro GCC a také do této skupiny můžeme zařadit proces nahrání zkompilovaného programu na cílové zařízení a jeho spuštění.

Samotný proces kompilace se znalostmi z bodů výše je již poměrně rutinní záležitost, která není na ničem závislá. Proto jí můžeme definovat jednou a společně pro různé platformy a projekty.

2.5.2 Definice modulů



Obrázek 2.4: Ilustrace build systému. Uzly čerchovaně jsou implementované pro každou platformu zvlášť, zbytek je multiplatformní.

Na obrázku 2.4b je znázorněna struktura testovací aplikace (respektive z jakých modulů se skládá). Modul je definován složkou obsahující soubor pojmenovaný *module.yaml*. V tomto souboru jsou pak uvedeny všechny potřebné náležitosti. Například seznam zdrojových souborů, v jakých adresářích se mají hledat hlavičkové soubory, definice maker atd. Zároveň je zde definováno na jakých dalších modulech je daný modul závislý.

Definice modulů je stejně jako konfigurace ANN ve formátu YAML. Jeho konkrétní strukturou se v tuto chvíli nebudeme zabývat, a podrobněji jí ukážeme až v kapitole 3.

Pro získání informací definovaných pro určitý modul jsme vytvořili jednoduchý skript napsaný v pythonu pojmenovaný nápaditě *module*. Jeho použití je velmi jednoduché. Například pro získání všech zdrojových souborů určitého modulu stačí použít skript následovně:

```
$ module --sources
/…/ann_imp.c
/…/ann_imp_tests.c
```

V tomto případě skript našel soubor *module.yaml* pro adresář ve kterém se právě nacházíme a vytisknul cesty k jeho zdrojovým souborům. Obdobným způsobem můžeme vytisknout všechny makra (`module --defines`), nebo například adresáře s hlavičkovými soubory (`module --includes`). Skript rekurzivně projde všechny závislosti daného modulu a obsáhne jejich definice ve výstupu.

2.5.3 Proces kompilace a spuštění

Definicí modulů jsme pokryli velkou část potřebných informací pro proces kompilace. Ten je definován zdrojovými soubory programu Make. Logika tohoto procesu je rozdělena na dvě části. První z nich definuje věci platformě závislé (přepínače GCC, proces nahrání a spuštění aplikace). Tato část je pro každou platformu implementovaná v odděleném souboru a při běhu dynamicky zvolena v závislosti na cílovém zařízení. Druhou částí je definice samotného procesu kompilace, který využívá informací z první části a zároveň informací z definic modulů.

2.6 Měření výkonnosti

Měření výkonnosti implementací ANN v této práci můžeme rozdělit na dvě částí: paměťovou a časovou.

2.6.1 Paměťová náročnost

Měření paměťové náročnosti je v našem případě poměrně jednoduché. Jelikož je implementace ANN navržená tak, aby nepoužívala dynamicky alokovanou

paměť, můžeme požadované informace získat analýzou zkompilovaných produktů. K této analýze použijeme nástroj `size` z GNU Toolchainu. Pomocí tohoto programu jsme schopni získat přesný počet bajtů které daná implementace bude okupovat ve FLASH paměti, a zároveň kolik paměti bude vymezeno po startu aplikace v RAM. Tyto informace jsou určitě přínosné, neposkytují ale ucelený náhled na limity určitých zařízení. Jako nástin skutečných limitů pak můžeme považovat následné rychlostní testy ANN, jelikož při nich je implementace ANN spouštěna v programu, který může reprezentovat minimální program na dané platformě (respektive pouze inicializuje základní věci nutné pro testování).

2.6.2 Časová náročnost

Další testovanou metrikou je odezva sítě. Tedy jak dlouho trvá určité síti na konkrétním zařízení spočítat výstup pro daný vstup. Toto měření je náročnější a je nutné si předem určit nějaká kritéria abychom dosáhli relevatních výsledků.

- Zaprvé by testované zařízení mělo být nakonfigurováno, aby dosáhlo „maximálního možného standardního“ výkonu. Například při testování zařízení s určitou uvedenou frekvencí výrobcem je většinou možné mikrokontrolér jak podtaktovat, tak přetaktovat nad tuto hodnotu. V této práci tedy všechny zařízení budeme testovat nakonfigurované na horní mez škály dané výrobcem.

- Pokud je to možné pomocí standardních nástrojů poskytovaných výrobcem, v době běhu sítě za účelem měření rychlosti nebude testovaný algoritmus přerušovaný a bude mu poskytnuto 100 % výpočetního výkonu CPU.

Samotné měření času je platformě závislou záležitostí, z toho důvodu je v „Testovací knihovně“ definováno pouze rozhraní. Implementace je pak rozdílná pro každou platformu. V testovaných vestavných systémech jsme bez výjimky použili pro měření vhodně nakonfigurované časovače[3], nebo jejich kombinaci. Jejich konkrétní nastavení jsme pak upravovali při testování, pro dosáhnutí co nejvyšší přesnosti měření.

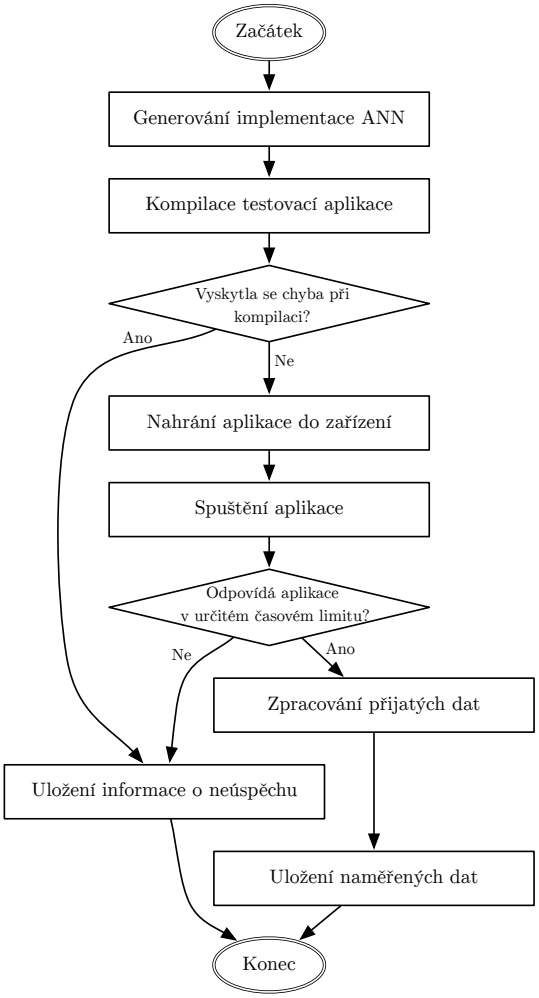
2.6.3 Sběr výsledků měření

Jelikož si tato práce dáva za cíl udělat měření kompletně automatizované, je nutné nějakým způsobem vyřešit i shromažďování naměřených dat. Poté co aplikace naměří odezvu sítě, vytiskne čas spolu s dalšími metadaty přes

[3] Anglicky Timer nebo Counter. Jedná se o hardwarovou komponentu, která je mimo jiné schopná počítat změny vstupního signálu – například hodin.

sériovou linku. Tyto informace jsou počítačem přečteny a následně uloženy do lokální databáze (celý proces je znázorněn na obrázku 2.5).

Součástí práce je také mnoho skriptů, s jejichž pomocí je možné automaticky testovat celé sady neuronových sítí na více zařízeních současně. Podrobněji jsou tyto nástroje a jejich použití popsány v kapitole 3.



Obrázek 2.5: Proces testování ANN na nějakém zařízení.

Kapitola 3

Implementace

V této kapitole se blíže seznámíme s různými implementačními detaily práce, strukturou celého projektu, základem jeho použití a možnostmi jeho rozšíření.

Předtím než začneme popisovat konkrétněji implementační detaily je nutné zmínit, že co v této práci označujeme za „platformu“ či „zařízení” je v implementaci vždy označováno jako „board“ (jelikož se vždy jedná o konkrétní vývojovou desku). Zároveň je každá podporovaná platforma identifikována jednoslovným názvem, například `nrf51`, `mega2560` nebo `stm32f4`. Pokud v této kapitole někde uvedeme „`<board>`“, myslíme tím možnost dosazení některého z identifikátoru podporovaných platforem na toto místo.

3.1 NetGen – Generátor implementací ANN

Generátor sítí je implementován jako *Python package*, nacházející se v adresáři `/netgen`. Tento package je navržen pro použití z příkazové řádky následovným způsobem:

```
$ python netgen -d <destination_dir> <config>
```

V tomto případě tedy generátor uloží implementaci ANN podle konfiguračního souboru `<config>` do adresáře `<destination_dir>`. Jak už bylo uvedeno v předchozí kapitole, implementace je vygenerována do tří souborů. Prvním z nich je hlavičkový soubor, další dva implementují testy a samotnou síť.

Implementace je generována pomocí šablonovacího systému Jinja. Zmíněné soubory implementace jsou pak generovány podle jinja šablon umístěných v adresáři `/netgen/src/templates`. Kód těchto šablon samozřejmě není sám o sobě validní C kód, jelikož obsahuje velké množství dynamických jinja výrazů. Z tohoto důvodu je jeho vývoj poměrně náročný (v porovnání s psaním prostého C kódu), proto tyto šablony obsahují pouze nejdynamičtější část celé implementace.

Větší část implementace je psána jako hybridní zdrojový kód mezi Jinja šablonou a C zdrojovým kódem (tím rozumíme fakt, že jeden soubor je zároveň validní C kód, a zároveň obsahuje nutný jinja kód, aby mohl být následně použitý při generování implementace). Tento druh „šablon" je oddělen v adresáři `/netgen/src/shared`. Fakt, že velká část implementace, která nemusí být dynamicky generována, je oddělena v těchto „hybridních souborech", dělá celou implementaci čitelnější a usnadňuje to její vývoj. Tyto hybridní soubory totiž mohou být sami o sobě použity například při prototypování nové funkce, která má být následně zobecněna šablonou.[4]

3.1.1 Optimalizace pro různé platformy

Ačkoliv generovaný kód si klade za cíl být dostatečně optimalizovaný, stále se jedná o obecnou multiplatformní implementaci v jazyce C. Je tedy na kompilátoru, aby tento kód co nejlépe přeložil pro cílovou platformu. Různé platformy ale můžou obsahovat rozšíření instrukční sady, s kterým je možné dosáhnout lepších výsledků, ale GCC tyto instrukce nedokáže plně využít.

Z tohoto důvodu jsou nejkritičtější funkce definovány jako `weak` symboly. Při jejich optimalizaci pro určitou platformu tedy stačí tyto funkce přepsat vlastní optimalizovanou variantou a není nutné zasahovat do vygenerované implementace ANN.

Z našich testovaných mikrokontrolerů je jedinou použitelnou rozšiřující instrukční sadou sada DSP, která je implementována v některých čipech společnosti ARM (v našem případě je implentována na desce STM32F4). Tuto instrukční sadu je možné pohodlně využít pomocí knihovny `arm_math`, která zapouzdřuje časté matematické operace pro procesory ARM. Z důvodu popularity procesorů ARM jsme se rozhodli podporu pro tuto knihovnu vestavět přímo do kódu generovaného programem netgen (při kompilaci je tedy automaticky použita knihovna `arm_math`, pokud je dostupná).

Pokud se například podíváme na implementaci vrstvy perceptronu, můžeme vidět, že její průchod je implementovaný jako násobení matice výstupu předchozí vrstvy a matice vah. Operace násobení matic je v tomto případě zapouzdřená v souborech `matrix.h` a `matrix.c`. `matrix.c` pak v případě, že je dostupná knihovna `arm_math` použije násobení matic z této knihovny. Tato knihovna pak na určitých procesorech využije SIMD[5] instrukcí.

[4] V adresáři `/netgen/src/prototypes` jsou k vidění prototypy implementací různých druhů neuronových sítí, podle kterých byly tvořeny šablony v `/netgen/src/templates`. Jak šablony, tak tyto prototypy, mají maximum statické implementace společné v hybridních souborech v `/netgen/src/shared`.

[5] **S**ingle **I**nstruction **M**ultiple **D**ata – označení pro instrukce, které operují na více datech najednou.

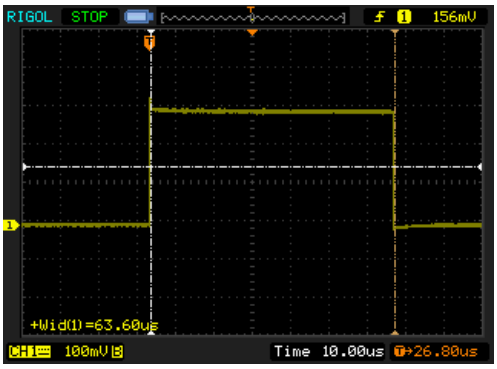
3.2 Ověření správnosti měření

Implementace podpory určité platformy se z části skládá z její počáteční konfigurace. Součástí této konfigurace je vždy inicializace časovače, který je klíčovým prvkem pro měření času. Problém u této konfigurace ale je, že se obtížně pozná zda je daný časovač nakonfigurovaný správně.

Cílem tedy bylo otestovat, že každá implementace naší „Testovací knihovny“ implementuje správně fungující měření času. Pro ověření správné funkčnosti jsme se rozhodli porovnat různé naměřené hodnoty touto knihovnou s hodnotami z externího zařízení a to konkrétně osciloskopu. Vytvořili jsme tedy aplikaci která v cyklu prováděla následující:

1. Začala měřit čas.
2. Rozsvítila LED.
3. Počkala dobu X.
4. Zhasla LED.
5. Skončila měření času.
6. Vytiskla naměřenou hodnotu do textového výstupu.

Připojením výstupu LED k osciloskopu jsme tak mohli živě pozorovat hodnoty měřené aplikací v porovnání s hodnotami z osciloskopu. U každé platformy jsme zároveň provedli test s více různými časy ve 4. bodě. Jednak s relativně krátkým časem pro ověření přesnosti (řádově desítky mikrosekund), také ale s časem v řádu několika vteřin pro ověření rozsahu časovače.



Obrázek 3.1: Kontrola měření času osciloskopem.

3.3 Automatizace testování

Pro automatizaci celého procesu testování jsme vytvořili několik skriptů napsaných v jazyce Bash a Python.

Základní z nich je `run_nn.sh`, který slouží pro otestování jedné konfigurace neuronové sítě na určitém zařízení. Výsledek testování je pak uložen do

předem specifikované databáze. Pokud při testování nastane nějaká známá chyba, je tato událost také zaznamenána v databázi. Při výskytu neznámé chyby je do standardního výstupu vypsaný celý log operace a následně je uživatel dotázán o komentář který bude uložen v databázi. Proces implementovaný v tomto skriptu je popsán obrázkem 2.5 na stránce 23. Tento skript také provede analýzu na zkompilovaných souborech a informace o paměťové náročnosti uloží.

Dalším skriptem je `run_suite.sh`. Jeho cílem je testování více konfigurací neuronových sítí na více zařízeních (sériově) za použití skriptu `run_nn.sh` uvedeného výše. Sada konfigurací, které chceme testovat, je možné specifikovat pomocí textového souboru, kde na každém řádku je uveden jeden konfigurační soubor neuronové sítě.

Předchozí skript neprovádí testy na různých zařízeních paralelně. Sice by nebyl problém takový skript vytvořit (a tím výrazně snížit dobu testování), bylo by ale zbytečně komplikované řešit například situace, když nastane neznámá chyba a je uživatel dotázán o komentář. Pro paralelní testování více zařízení je tak mnohem vhodnější použít nějaký terminal multiplexer[6], s kterým můžeme velice jednoduše spustit paralelně několik instancí `run_suits.sh` (vždy jednu instanci pro jedno zařízení).

```
# Následující příkaz otestuje všechny konfigurace
#   uvedené v souboru config/suite_basic.txt.
# Druhý parametr je přidán jako komentář k výsledkům
#   v databázi, pro jednoduchou identifikaci.
# Jelikož jsme neuvedli konkrétní zařízení na kterých testy
#   provést, bude test proveden na všech dostupných zařízeních.
./run_suite.sh config/suite_basic.txt "test only"
# Spuštění sady testů pro konkrétní zařízení
BOARDS="nrf51 mega2560" ./run_suite.sh config/suite_basic.txt "2. test"
# Spuštění konkrétního testu na jednom zařízení
BOARD=nrf51 DATABASE=results.db TEST_NAME="3. test" \
    CONFIG_PATH=./config/xor.yaml ./run_nn.sh
```

Ukázka kódu 3: Ukázka použití skriptů pro testování výkonnosti.

Další skripty, které jsou použity v různých částech této práce, jsou:

`./check_connection.sh` Skript, který vypíše které z podporovaných zařízení jsou připojené, a které odpojené.

`./bin/module` Python skript implementující práci s moduly popsanými v sekci 2.5.2 na straně 20.

[6]Terminal Multiplexer je program, který umožnuje rozdělit okno terminálu na více virtuálních terminálů a nad nima operovat (například synchronizovaně vykonnávat příkazy).

`./bin/serial-collect` Python aplikace, která pro daný sériový port monitoruje příchozí data a z jejich obsahu vybírá reporty, které mají být následně uloženy v databázi.

`./bin/list-ports` Vypíše všechny aktuálně dostupné sériové porty.

`./bin/save-report` Jednoduchá utilita pro vložení reportu s naměřenými výsledky do nějaké databáze.

3.4 Rozšíření o nové platformy

V této a následující sekci probereme možnosti rozšíření práce. Zaměříme se především na dvě věci: rozšíření programu NetGet o nový druh neuronů a přidání podpory pro nové platformy.

Přidání podpory pro novou platformu (respektive zařízení) se skládá ze dvou hlavních částí. První je rozšíření build systému o tuto platformu. Poté je potřeba vytvořit modul implementující platformě závislé záležitosti z „Testovací knihovny“.

Všechny věci týkající se této nové platformy jsou pak uloženy v adresáři `/boards/<board>/`, kde `<board>` je jednoslovný název pro novou platformu.

3.4.1 Přidání podpory do build systému

Pro rozšíření build systému je nutné vytvořit makefile `/boards/<board>/make/<board>.mk`. Tento soubor musí definovat proměnné popsané v záhlaví souboru `/boards/shared/make/build.mk`. Jedná se převážně o vytvoření seznamu souborů pro kompilaci a sady přepínačů. Velká část těchto informací se dá získat přímo z definice modulů (například všechny zdrojové kódy v makefilu získáme pomocí výrazu `$(shell module --sources)`.

Další součástí tohoto souboru je definice operace pro spuštění aplikace na dané platformě, spolu s informací přes jaký sériový port aplikace bude komunikovat. Blížší popis je v ukázkovém souboru `/boards/template/make/template.mk`.

3.4.2 Implementace platformě závislých funkcí

Druhou podstatnou částí pro přidání nové platformy je vytvoření modulu, který bude implementovat funkce „Testovací knihovny“ (která se nachází v `/boards/shared/lib`). Tato implementace pro danou platformu musí být v adresáři `/boards/<board>/lib`. V tomto adresáři je tedy nutné vytvořit soubor `module.yaml`, který bude definovat které zdrojové soubory mu náleží, jaké jsou cesty k hlavičkovým souborům atd.

3.5 Rozšíření podporovaných druhů neuronů

Přidání nového druhu neuronu realizujeme vytvořením nového typu vrstvy (jelikož neurony jsou vždy stejné v rámci jedné vrstvy). Prvním krokem je přidání podpory do deserializace konfiguračního souboru. Ta se provádí v souboru `/netgen/model.py`. Cílem zde je tedy transformovat konfigurační soubor na python instanci třídy `Network`, která je následně používaná v šablonách pro gerování zdrojových kódu.

Implementaci této nové vrstvy je pak vhodné oddělit do samostatných souborů uložených v `/netgen/src/shared/`. Tyto soubory by měly obsahovat maximum kódu, který není potřeba generovat dynamicky. Prakticky by tak rozhraním těchto souborů měla být funkce, která pro vstupní vektor a strukturu uchovávající informace vrstvy vrátí výstup vrstvy. Přestože jsou tyto soubory používané jako ninja šablony (a je tedy do nich možné vložit výrazy pro šablonovací systém), mělo by se jednat o validní C kód, aby bylo možné například ručně vytvořit prototyp sítě s použitím těchto souborů.

Posledním krokem je úprava souboru `/netgen/src/templates/network.c.template`. V něm je zapotřebí udělat následující změny:

- Pokud síť obsahuje naší novou vrstvu, zajistit vložení souborů implementující tuto vrstvu do výsledného souboru.
- Definovat formou statických (ideálně konstatních) proměnných všechny potřebné struktury pro danou instanci této vrstvy.
- Zajistit vygenerování kódu pro instance této vrstvy v funkci implementující samotný průchod sítí.

Při dodržení následujících několika zavedených konvencí v implementaci bude výrazně jednodušší integrovat nový druh vrstvy do projektu (nějaké věci budou fungovat i automaticky).

- Soubory implementující danou vrstvu jsou pojmenovány stejně jako typ, který ji identifikuje.
- Informace k dané vrstvě jsou dostupné ze struktury pojmenované `<layer_type>_layer_t`.
- Funkce, která implementuje daný průběh vrstvou, má signaturu odpovídající `void <layer_type>_layer_activate(<layer_type>_layer_t const * layer, ann_number_t const * input, ann_number_t * output)`.

Kapitola 4

Experimentální výsledky

Před tím, než se zaměříme na konkrétní naměřené hodnoty, je dobré připomenout které naměřené veličiny máme k dispozici. První veličinou je **čas** jednoho průběhu implementace neuronové sítě. Dále náš systém automaticky měří potřebnou **paměť** implementace podle zkompilovaných produktů. Naměřené hodnoty paměti pak rozdělujeme na dvě hlavní části. Konstantní data, která jsou uložena v paměti FLASH a potřebnou velikost bloku dat, která bude pro ANN vymezena v paměti RAM.

První fází testování, které jsme uskutečnili, bylo vytvoření relativně velkého počtu konfigurací neuronových sítí pro získání základních znalostí o limitech daných zařízení. Testovali jsme řádově jak velké sítě je možné spustit na daném zařízení, jaké jsou jejich odezvy apod. Výsledky těchto měření jsou zajímavé sami o sobě, jsou ale příliš obsáhlé na to abychom je zde uváděli v úplné podobě. Proto se v této kapitole zaměříme více na analýzu naměřených dat.

Můžeme si všimnout, že vrstvy neuronové sítě jsou oddělené funkční celky. Pokud se tedy budeme snažit zobecnit výkonnost implementace, dává smysl analyzovat odděleně chování jednotlivých druhů sítí. Součet jejich výkonnostích parametrů pak relativně dobře představuje parametry celé sítě.

4.1 Časová náročnost vrstvy typu Perceptron

Při analýze výpočtu průchodu jedné vrstvy perceptronu jsme zjistili, že jeho asymptotická složitost se dá vyjádřit jako $\mathcal{O}(n \cdot m)$, kde n je počet neuronů předchozí vrstvy a m počet neuronů v dané vrstvě perceptronu.

Pro testované sítě (skládající se pouze z vrstev typu perceptron) jsme tedy vypočítali teoretickou náročnost, kterou nám poskytuje vzorec výše. Následné porovnání těchto hodnot a časů naměřených pro dané sítě ukázalo vysokou korelaci těchto veličin (přes 0.9, průměrně 0.97). Hodnoty pro různá zařízení jsou uvedeny v tabulce 4.1. Díky tomuto jsme poměrně přesně schopni definovat časovou náročnost jakékoliv perceptronové sítě pro určité zařízení. Například

pro perceptron topologie 5–10–2 je teoretická náročnost 70 ($5 \cdot 10 + 10 \cdot 2$). Jednoduchým vynásobením koeficientem b z tabulky 4.1 tak například zjistíme, že průběh této sítě bude na *stm32f1* trvat $70 \cdot 0.00356$, tedy zhruba 0.25 ms.

označení zařízení	korelační koeficient	b – koeficient
arduino	0.913	0.0280
mega2560	0.973	0.0280
nrf51	0.999	0.0222
stm32f1	0.999	0.00356
stm32f4	0.998	0.000670

Tabulka 4.1: Srovnání vypočítané náročnosti perceptronů s měřením. Koeficient b je výsledkem lineární regrese ($t = bx$, kde x je teoretická náročnost, t odhadovaný čas)

4.2 Časová náročnost vrstvy typu RBF

Testované konfigurace sítí typu RBF jsou složeny z jedné RBF vrstvy a výstupní vrstvy perceptron (toto odpovídá nejpoužívanější topologii u neuronů RBF, viz sekce 1.3). Výsledky těchto měření jsou uvedeny v tabulce 4.4. Pro zobecnění se ale pokusíme zaměřit speciálně na časovou náročnost samotné RBF vrstvy. K tomuto účelu jsme provedli dva druhy testů. Jelikož dokážeme poměrně přesně vypočítat časovou náročnost vrstvy perceptronu, odečetli jsme z výsledků testů RBF sítí náročnost vrstev perceptronu v nich obsažených. Pro ověření jsme dále vytvořili sadu konfigurací RBF sítí neobsahující žádné vrstvy typu perceptron.

Průběh vrstvy RBF můžeme jednoduše popsat takto: pro každý neuron ve vrstvě je nutné spočítat euklidovu vzdálenost jeho vah a vstupního vektoru a na ní aplikovat aktivační funkci (viz rovnice 1.5 na 5. straně). Velice jednoduše bychom tedy mohli říct, že asymptotická složitost tohoto algoritmu je stejně jako u perceptronu $\mathcal{O}(n \cdot m)$. Následně jsme tedy provedli lineární regresi pro nezávislé proměnné n a m. Výsledek ukázal, že samotný výpočet euklidovy vzdálenosti můžeme při odhadech zcela zanedbat (důvodem je nejspíš aktivační funkce, která je v porovnání s euklidovou vzdáleností podstatně náročnější). Při odhadu náročnosti tedy můžeme vycházet hlavně z počtu neuronu ve vrstvě samotné. Stejným procesem jako u vrstvy perceptronu pak můžeme vyjádřit náročnost vrstvy na různých zařízeních tabulkou 4.2.

4.3 Časová náročnost vrstvy typu Izhikevich

Zobecnit časovou náročnost vrstvy izhikevich je o něco jednodušší než dva předchozí typy vrstev. Neurony v této vrstvě jsou samostatně funkční celky nezávislé na velikosti vstupu. Udělali jsme proto několik měření sítí, které

označení zařízení	korelační koeficient	b – koeficient
arduino	0.989	0.2568
mega2560	0.999	0.2480
nrf51	0.764	0.8288
stm32f1	0.733	0.1091
stm32f4	0.999	0.0553

Tabulka 4.2: Srovnání vypočítané náročnosti RBF vrstvy s měřením. *Koeficient náročnosti* je čas běhu implementace vydělený její teoretickou náročností.

obsahovaly různé velikosti vrstev izhikevich a žádných jiných. Prostým vydělením času počtem neuronů pak získáváme časovou náročnost jednoho neuronu na určitém zařízení (výsledky jsou v tabulce 4.3).

označení zařízení	prům. čas výpočtu jednoho izh. neuronu
arduino	0.0786 ms
mega2560	0.0824 ms
nrf51	0.0862 ms
stm32f1	0.0114 ms
stm32f4	0.000979 ms

Tabulka 4.3: Průměrný čas výpočtu neuronu izhikevich na různých zařízeních.

4.4 Paměťová náročnost sítí

U paměťové náročnosti se zaměříme na paměť RAM, jelikož nároky na paměť FLASH byly málo kdy omezující pro běh sítě. Generované implementace ANN používají paměť RAM pro dva hlavní učely. Zaprvé jako vyrovnávací paměť (ang. buffer) pro přenos dat mezi vrstvami. Vygenerovaný kód vytváří maximálně dva buffery, jejichž velikost můžeme odvodit podle největšího počtu neuronů v lichých a sudých vrstvách. Podle zvolených datových typů při konfiguraci pak můžeme vypočítat přesný alokovaný počet bajtů.

Druhý podstatný alokovaný blok v RAM je nutný v případě použití vrstvy typu izhikevich. Pro každý neuron v této vrstvě je nutné uložit proměnné u a v, které jsou popsány v kapitole 1.4. Tudíž izhikevich vrstva vyžaduje $2 \cdot sizeof(number\_type) \cdot neurons\_count$ bajtů RAM paměti.

4.5 Dodatečné informace

Všechny měření byly prováděny za použití datového typu `float` pro `number_type` a `int` pro `size_type` (viz kapitola 2.4.4.2). Použitá zařízení s jejich základními parametry jsou uvedeny v tabulkách níže.

Arduino Uno

Označení v implementaci	arduino
Mikrokontrolér	ATmega328P
Frekvence	16 Mhz
Paměť RAM	2 kB
Paměť FLASH	32 kB
FPU	Ne

ATmega2560 – deska vyrobená na ČVUT

Označení v implementaci	atmega2560
Mikrokontrolér	ATmega2560
Frekvence	16 Mhz
Paměť RAM	8 kB
Paměť FLASH	256 kB
FPU	Ne

Nordic nRF51 Dongle

Označení v implementaci	nrf51
Mikrokontrolér	ARM Cortex-M0
Frekvence	16 Mhz
Paměť RAM	32 kB
Paměť FLASH	256 kB
FPU	Ne

Nucleo – STM32F103RB

Označení v implementaci	stm32f1
Mikrokontrolér	ARM Cortex-M3
Frekvence	64 Mhz
Paměť RAM	20 kB
Paměť FLASH	128 kB
FPU	Ne

STM32F429I – DISCO

Označení v implementaci	stm32f4
Mikrokontrolér	ARM Cortex-M4
Frekvence	180 Mhz
Paměť RAM	260 kB
Paměť FLASH	2048 kB
FPU	Ano

	arduino	atmega2560	nrf51	stm32f1	stm32f4
Perceptron					
2-10-2	1.168	1.188	0.918	0.145	0.0366
2-100-2	–	10.64	8.568	1.367	0.2466
2-1000-2	–	–	85.685	13.36	2.35
Rekurentní per.					
2-10-2	2.652	2.748	2.203	0.29	0.0624
2-100-2	–	–	145.76	16.618	2.025
2-1000-2	–	–	–	–	–
RBF					
2-10-2	2.568	2.684	5.814	0.68	0.575
2-100-2	–	26.372	57.29	6.95	5.003
2-1000-2	–	–	586.7	71.9	51.5
Izhikevich					
10	0.792	0.832	0.857	0.115	0.011
100	7.8	8.192	8.63	1.132	0.094
1000	–	–	122.34	16.48	1.28

Tabulka 4.4: Velmi malá podmnožina naměřených dat. Hodnoty jsou časy v milisekundách.

Závěr

Výsledkem této práce je sada několika nástrojů, jejichž spojení tvoří platformu pro rychlé a jednoduché testování výkonnosti algoritmů umělých neuronových sítí.

Vytvořili jsme program, který na základě konfiguračního souboru generuje implementaci určité neuronové sítě v jazyce C. Vygenerované zdrojové kódy jsou sami o sobě optimalizované pro vybrané platformy a je možné je dále optimalizovat bez přímého zásahu do vygenerovaných souborů. Podporovanými sítěmi jsou Perceptron, RBF a izhikevich s možností konfigurace různých výsledných topologií (podporované jsou i rekurentní sítě). Konfigurační soubor popisující požadavky na výslednou neuronovou síť zároveň umožňuje použití python výrazů, s nimiž lze načíst části konfigurace dynamicky z různých zdrojů.

Další samostatně funkční celek je vytvořený vlastní build systém. Je navržený tak, aby bylo možné jasně oddělit specifikaci různých modulů, build proces, a nebo podporu pro novou platformu.

Vytvořili jsme multiplatformní knihovnu, která zapouzdřuje operace nad různými platformě závislými operacemi. Platformě závislé implementace této knihovny jsou odděleny v samostných modulech pro jednoduché rozšíření o nové platformy. Pro testování v rámci této práce jsme implementovali podporu pro pět různých vestavěných zařízeních. Tato knihovna je pak využita pro implementaci aplikace běžící na testovaných zařízeních, nebo pro aplikaci, s jejíž pomocí byla ověřena přesnost měření.

Pro zautomatizování celého procesu testování jsme vytvořili sadu skriptů. Díky nim je nutné pouze zapojit některá z podporovaných zařízení k počítači, a pomocí jednoho příkazu spoustit celé sady testů. Výsledky testování jsou následně automaticky ukládány do lokální SQLite databáze. Pro základní analýzu výsledku je tedy možné použít SQL dotazy nad výslednou databází (pro komplexnější analýzu jsme použili program RapidMiner).

V neposlední řadě jsou výsledkem této práce naměřená data a jejich analýza. V analýze tak například uvádíme postup, pomocí kterého je možné pro

obecnou neuronovou síť relativně přesně vypočítat její časovou i paměťovou náročnost na daném zařízení. Tyto informace můžeme považovat za velice uspokojivé, co se splnění cílů této práce týká.

Literatura

[1] *Atmel ATmega640/V-1280/V-1281/V-2560/V-2561/V*. 2014. Dostupné z WWW: <www.atmel.com/images/atmel-2549-8-bit-avr-microcontroller-atmega640-1280-1281-2560-2561_datasheet.pdf>

[2] *ATmega48A/PA/88A/PA/168A/PA/328/P*. 2015. Dostupné z WWW: <http://www.atmel.com/images/atmel-8271-8-bit-avr-microcontroller-atmega48a-48pa-88a-88pa-168a-168pa-328-328p_datasheet_complete.pdf>

[3] *STM32F103xB*. 2015. Dostupné z WWW: <www.st.com/content/ccc/resource/technical/document/datasheet/33/d4/6f/1d/df/0b/4

[4] *Discovery kit with STM32F429ZI MCU*. 2016. Dostupné z WWW: <www.st.com/content/ccc/resource/technical/document/user_manual/6b/25/05/23/a9/45

[5] Arduino: Arduino Uno. 2016. Dostupné z WWW: <www.arduino.cc/en/main/arduinoBoardUno>

[6] Barr, M.: Introduction to Counter/Timers. 2002. Dostupné z WWW: <www.embedded.com/electronics-blogs/beginner-s-corner/4024440/Introduction-to-Cou

[7] Haykin, S. S.: *Neural networks and learning machines*. New York: Prentice Hall, 2009.

[8] Krákora, M.; Sojka, M.: Vícevrstvý perceptron a RBF sítě. 2001. Dostupné z WWW: <gerstner.felk.cvut.cz/biolab/X33BMI/slides/NN.pdf>

[9] Theodoridis, S.; Koutroumbas, K.: *Pattern Recognition*. Mass.: Academic Press, 2009.

[10] Vojáček, A.: Samoučící se neuronová síť - SOM, Kohonenovy mapy. 2006.

[11] Šíma, J.; Neruna, J.: *Teoretické otázky neuronových sítí*. Praha: Matfyzpress, 1996.

Příloha A

Seznam použitých zkratek

ANN Artificial Neural Network

RBF Radial Basis Function

IoT Internet of Things

ČVUT České Vysoké Učení Technické

FIT Fakulta Informačních Technologií