Aplikace neuronových sítí pro predikci využití systémových zdrojů

Bc. Martin Basel

Diplomová práce
2017

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky
akademický rok: 2016/2017

ZADÁNÍ DIPLOMOVÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Bc. Martin Basel**
Osobní číslo:	**A15358**
Studijní program:	**N3902 Inženýrská informatika**
Studijní obor:	**Informační technologie**
Forma studia:	**kombinovaná**
Téma práce:	**Aplikace neuronových sítí pro predikci využití systémových zdrojů**
Téma anglicky:	**The Application of Neural Networks for the Prediction of System Resource Usage**

Zásady pro vypracování:

1. **Proveďte analýzu současného stavu a možností v předpovídání využití systémových zdrojů.**
2. **Seznamte se s umělými neuronovými sítěmi.**
3. **Navrhněte vhodná data a extrahujte vhodným způsobem vstupní atributy.**
4. **Implementujte navržený model pro predikci využití systémových zdrojů.**
5. **Srovnejte navržený model s již existujícím produktem.**

Rozsah diplomové práce:

Rozsah příloh:

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

Seznam odborné literatury:

1. **ŠÍMA J., NERUDA R.: Teoretické otázky neuronových sítí. Vyd. 1. Praha, 1996: Matfyzpress, 390 s., ISBN 80-85863-18-9**
2. **KŘIVAN, Miloš. Úvod do umělých neuronových sítí. Vyd. 3., přeprac. Praha: Oeconomica, 2014, 44 s. ISBN 978-80-245-2024-7.**
3. **ZELINKA, Ivan. Umělá inteligence I: Neuronové sítě a genetické algoritmy. 1. vyd. Brno: VUT v Brně, 1998, 126 s. ISBN 80-214-1163-5.**
4. **FRANK, R.J, N. DAVEY a S.P. HUNT. Time Series Prediction and Neural Networks. Journal of Intelligent and Robotic Systems [online]. 2001, (31), 91-103.**
5. **HEATON, Jeff. Programming Neural Networks with Encog 3 in C#. Vyd. 1. St. Louis: Heaton Research, 2011, 232s. ISBN 978-1-60439-026-1.**

Vedoucí diplomové práce: **doc. Ing. Zuzana Komínková Oplatková, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **3. února 2017**

Termín odevzdání diplomové práce: **16. května 2017**

Ve Zlíně dne 3. února 2017

doc. Mgr. Milan Adámek, Ph.D.
děkan

prof. Mgr. Roman Jašek, Ph.D.
ředitel ústavu

Jméno, příjmení: Martin Basel

Název bakalářské/diplomové práce: Aplikace neuronových sítí pro predikci využití systémových zdrojů

Prohlašuji, že

- beru na vědomí, že odevzdáním diplomové/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 diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/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 UTB 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 – diplomovou/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 diplomové/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 diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/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 diplomové/bakalářské práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně, dne 8.5.2017

.....
podpis diplomanta

ABSTRAKT

Práce se zabývá problematikou předpovědi systémových zdrojů za použití umělé inteligence, konkrétně neuronových sítí. V teoretické části jsou popsány neuronové sítě a jejich využití pro předpověď hodnot. V praktické části je implementována konzolová aplikace pro předpověď využití systémových zdrojů. Této aplikaci jsou poté předkládány na vstup naměřené hodnoty a aplikace se snaží předpovědět vývoj využití těchto hodnot.

Klíčová slova: Předpověď využití CPU a RAM, neuron, neuronová síť, Resilient propagation, časové řady.

ABSTRACT

Thesis is focused on system resources prediction, using artificial intelligence, specifically neural networks. The teoretical part describes neural networks and their use for prediction of values. In the practical part is implemented console application, which predicts the usage of system resources. This application is given measured values of system resources usage and the application tries to predict the future values.

Keywords: Prediction of CPU and RAM usage, neuron, neural network, Resilient propagation, time series.

Rád bych poděkoval své vedoucí doc. Ing. Zuzaně Komínkové Oplatkové, Ph.D. za odbornou pomoc při psaní této práce. Dále chci poděkovat své přítelkyni a rodině za podporu a toleranci při psaní této diplomové práce.

OBSAH

ÚVOD

V dnešní době se snažíme předpovídat hodnoty v mnoha různých oborech. Například se snažíme předpovědět vývoj počasí, vývoj burzovního trhu, ceny zboží apod. Nahrává tomu i fakt, že se stále zdokonalují zařízení pro získání a ukládání dat. K předpovědi mohou být využity jednak statistické metody, nebo tzv. *Machine Learning* (strojové učení). Strojové učení je oblast umělé inteligence, zabývající se metodami, které dávají počítačům schopnost se učit.

Jednou z podoblastí strojového učení jsou umělé neuronové sítě. Tyto neuronové sítě jsou schopny s určitou mírou abstrakce simulovat činnost lidského mozku. Jednou z důležitých funkcí, kterou jsou schopny sítě simulovat, je schopnost učení. Pokud tedy budeme předkládat síti na vstup dostatečný vzorek dat, můžeme ji naučit určitým vztahům mezi vstupními daty. Neuronové sítě jsou dnes implementovány v některých statistických programech, např. Matlab[1)] nebo Statistica[2)].

V této práci budou předpovídány hodnoty využití dvou významných počítačových systémových zdrojů: CPU a RAM. Tyto zdroje jsou klíčové pro běh jakýchkoliv moderních zařízení. Proto je užitečné znát nejen aktuální hodnoty využití těchto zdrojů, ale také budoucí vývoj těchto hodnot. To je jeden z hlavních cílů této práce. Na základě těchto předpovědí mohou správci těchto zařízení vykonávat určité akce. Pokud tedy spolehlivá předpověď ukazuje, že v následujícím časovém intervalu přesáhne využití zdroje určitou hodnotu, dává to možnost správci zařízení předejít například přetížení, krátkodobému výpadku apod.

Práce je členěna na dvě části: teoretickou a praktickou. V úvodu teoretické části jsou charakterizovány systémové zdroje, jejichž hodnoty budou předpovídány, tudíž jsou důležité pro tuto práci. Dále bude popsána metoda, která k problému přistupuje ze statistického úhlu pohledu. V další části je popsán úvod do neuronových sítí, kde poznatky uvedené v této kapitole budou využity v praktické části. Na závěr teoretické části je vysvětleno, jak za pomoci neuronových sítí lze předpovídat budoucí hodnoty časových řad. V praxi totiž předpovídáme budoucí hodnoty využití na základě naměřených hodnot a na tyto naměřené hodnoty se lze dívat jako na časové řady.

Výstupem praktické části bude aplikace, která aplikuje nabyté poznatky z teoretické části. Aplikace bude umět využívat neuronové sítě k předpovědi budoucích hodnot využití CPU a RAM. Programovacím jazykem pro implementaci byl zvolen objektově orientovaný jazyk C#. Nejdříve bude objasněn postup při implementaci, i s ukázkami zdrojového kódu. Poté budou prováděny různé experimenty s nastavením sítě za účelem dosažení co nejpřesnější předpovědi. Budou ukázány situace, kdy je předpověď přesná,

[1)] `https://www.mathworks.com/products/matlab.html`
[2)] `http://www.statsoft.cz/`

ale naopak i datové vzorky, na kterých bude předvedeno, kdy předpověď nedává přesné hodnoty. Na závěr této části bude srovnán přístup pro předpověď na základě neuronových sítí a za použití statistické metody, nazývané lineární regrese.

Základními cíli této práce tedy jsou:

- uvedení do problematiky neuronových sítí z pohledu předpovědi budoucích hodnot časové řady,
- implementace neuronové sítě,
- srovnání výstupu z implementované aplikace s jiným přístupem pro předpověď.

I. TEORETICKÁ ČÁST

1 Úvod do problematiky predikce systémových zdrojů

V úvodní kapitole budou popsána predikovaná data a také bude vysvětlen pojem *systémový zdroj* v kontextu této práce. Poté budou popsány již existující řešení pro předpověď systémových zdrojů. Bude předvedena jedna z používaných metod, lineární regrese [1].

1.1 Vysvětlení pojmu systémový zdroj

Systémové zdroje jsou prostředky, které na základě potřeby operační systém přiděluje běžícím programům. Tyto zdroje jsou tedy jednou ze základních věcí pro běh, resp. spuštění programů. Mezi nejzákladnější zdroje patří:

- CPU,
- RAM,
- HDD (pevný disk).

Jednou z mnoha důležitých funkcí operačního systému je algoritmus přidělování těchto zdrojů. Ideálně tedy chceme, aby programy měly dostatek zdrojů, na druhou stranu ale nechceme těmito prostředky plýtvat. Proto je vhodné vytvořit jistý model, který by nám byl schopen předpovědět budoucí hodnoty.

Tato práce bude zaměřena především na predikci využití CPU a RAM, nicméně jak bude dále popsáno, předpověď by se dala vztáhnout i na využití HDD nebo síťových rozhraní počítače.

1.1.1 CPU

Central processing unit (CPU) je základní elektronická součástka v počítači, která je schopna vykonávat strojové instrukce. Někdy také může být nazývána jako *mozek počítače*. Každý procesor se skládá z několika stovek milonů až jednotek miliard tranzistorů. Tyto tranzistory jsou velice jednoduché součástky, které dovedou pouze rozlišovat mezi dvěma stavy: 0 (vypnuto) a 1 (zapnuto). V roce 1965 Gordon Moore vyslovil předpoklad, že každý rok se počet tranzistorů na čipu zhruba zdvojnásobí. Toto pravidlo platilo až donedávna, nyní se zdá že se výrobci blíží fyzikálním limitům a tento zákon přestává platit [2].

V dnešní době jsou procesory zastoupeny již prakticky ve všech elektronických zařízeních. Jako příklad si vezměme technologii IoT (*Internet of Things*). IoT představuje propojení jednotlivých elektronických zařízení, budov a vozidel s Internetem, za účelem vzájemné komunikace a sdílení informací mezi těmito zařízeními. Všechna tato zařízení

musí obsahovat jistý druh procesoru. A ať už jsou tyto prvky řízeny centrálně nebo se řídí sami, bylo by pro ně vhodné s jistou mírou předpovědět využití zatížení jejich procesorů. To by se mohlo využít např. pro efektivní plánování úspory elektřiny.

1.1.2 RAM

Paměti typu RAM (Random Access Memory) slouží jako operační paměť počítačů. V této paměti bývají uloženy běžící procesy a data pro tyto běžící procesy. Paměť se vyznačuje rychlejším čtením a zápisem než klasické pevné disky, nicméně paměti typu RAM mají vyšší cenu, a proto nemají takovou kapacitu. Dnes se v klasických stolních počítačích setkáváme hlavně s kapacitami 4-8 GB. Nicméně těchto pamětí může být v počítači vloženo více, záleží na počtu tzv. *slotů*, tedy přípojných míst na základní desce [3].

Paměti typu RAM jsou tzv. *volatilní*, tedy napěťově závislé. Pokud tedy počítač vypneme, obsah této paměti je vymazán. Při předpovídání využití paměti RAM tedy musíme vzít v potaz i časovou složku, abychom pracovali pouze s hodnotami, kdy byla paměť v provozu.

1.2 Analýza současných možností pro předpověď systémových zdrojů

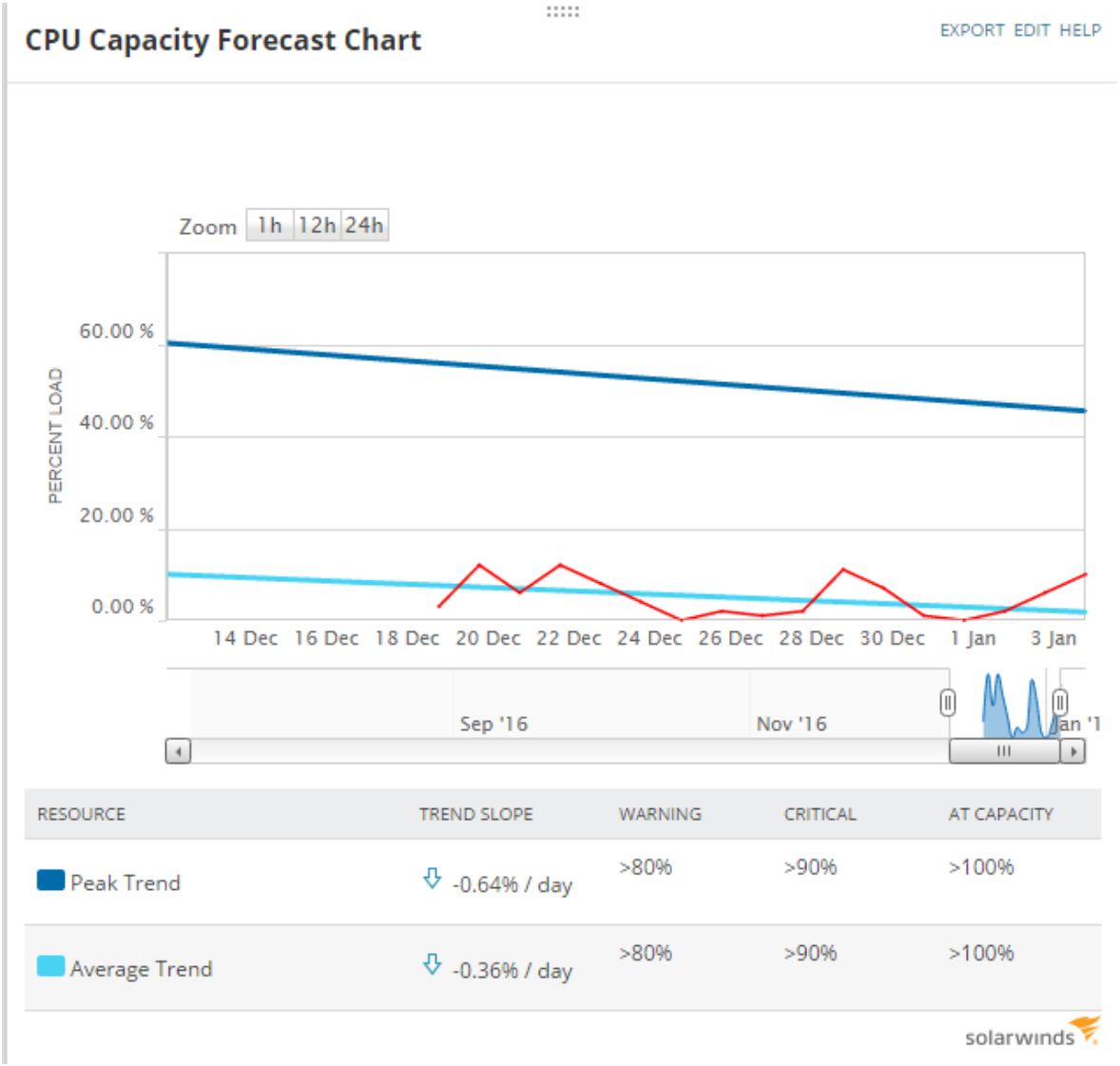
V současné době existují určité metody pro předpověď CPU a RAM. Jednou z těchto metod je algoritmus LiRCUP (Linear Regression Based CPU Usage Prediction Algorithm for Live Migration of Virtual Machines in Data Centers)[16]. Tento algoritmus byl představen v roce 2013 na konferenci *Euromicro Conference on Software Engineering and Advanced Applications*. Algoritmus se používá v cloudových technologiích, které beží na virtuálních strojích. Proto je důležité zajistit, aby tyto stroje nebyly přetížené, ale zároveň je třeba zajistit efektivní využití elektrické energie. LiRCUP slouží pro předcházení situacím, kdy jsou virtuální stroje přetížené, tím že se snaží předpovědět CPU a na základě výsledku spustit migraci na jiný stroj. Tím může například zabránit porušení SLA[1)], což je smlouva mezi zákazníkem a poskytovatelem služeb, kdy se poskytovatel zavazuje k určité dostupnosti služeb. Pokud by totiž služba nebyla dostupná, může dojít k porušení podmínek ve smlouvě. Je založen na lineární regresi, což je metoda používaná pro proložení bodů přímkou.

Jedním z dalších nástrojů jsou produkty od firmy SolarWinds[2)]. Také tyto produkty používají proložení naměřených hodnot přímkou a jsou schopny předpovědět, kdy by mohlo dojít k přetížení (viz. Obr. 1.1). Červenou křivkou jsou zobrazeny naměřené hodnoty, světle modrá přímka představuje predikci průměrných denních hodnot a tmavě

[1)] Service Level Agreement

[2)] `http://www.solarwinds.com/`

modrá přímka znázorňuje předpověď nejhorších možných hodnot.



Obr. 1.1 Předpověď využití CPU od firmy SolarWinds. Na podobném principu funguje také předpověď pro RAM nebo využití místa na disku.

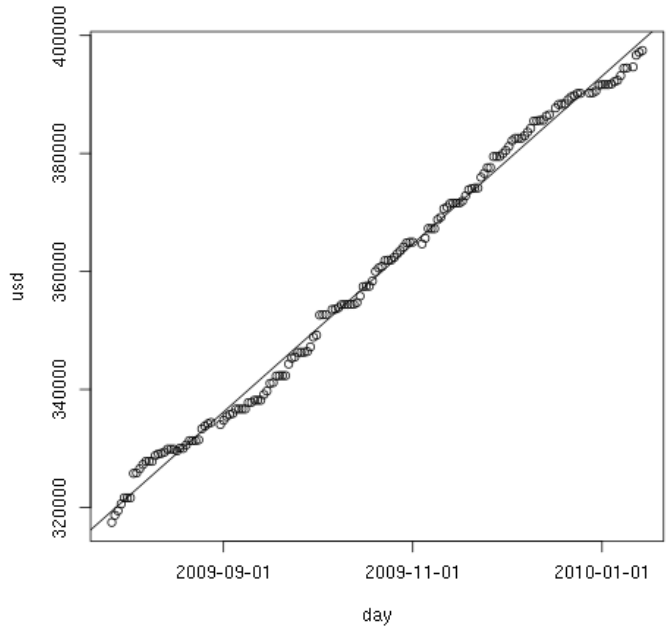
1.3 Lineární regrese

Pro předpověď využití systémových zdrojů v této práci bude využito neuronových sítí, nicméně zde bude stručně představena další metoda, která se používá v praxi (např. v algoritmu LiRCUP), lineární regrese. Lineární regrese je statistická metoda používaná pro modelování závislosti mezi proměnnými. Pomocí lineární regrese se snažíme proložit body v grafu přímkou. V kontextu této práce si tedy můžeme představit, že osa x bude představovat čas a na ose y budou vyneseny naměřené hodnoty CPU nebo RAM v průběhu času. Pokud tedy naměřené hodnoty proložíme přímkou, můžeme dosáhnout jistého způsobu předpovědi budoucích hodnot.

Obecný tvar přímky je popsán rovnicí 1.1. Lineární regrese se snaží pomocí metody nejmenších čtverců najít koeficienty a a b.

$$f(x) = ax + b \tag{1.1}$$

Dalším z důležitých systémových zdrojů, které počítač využívá, je pevný disk. Na stránce *Hard drive occupation prediction with R* [5] je popsán postup pro předpověď volného místa na disku. Používá programovací jazyk a zároveň statistický software R[3)]. Lineární regrese funguje dobře pro lineární data, viz Obr. 1.2, kde na ose x je zobrazena časová proměnná a na ose y je vyneseno aktuální využití HDD. Jak lze vidět, velkou nevýhodou této metody je náchylnost k odchýleným hodnotám. Pokud tedy budeme provádět na počítači pravidelnou údržbu a promazávat soubory, stane se tato metoda nepřesnou. Je tedy zapotřebí tuto metodu nějak optimalizovat. Autoři článku [5] navrhují například počítat pouze s diferencemi mezi jednotlivými dny. Tím pádem je algoritmus schopen extrémní hodnoty (před a po údržbě disku) odfiltrovat.



Obr. 1.2 Lineární model pro předpověď využití HDD za pomoci lineární regrese [5].

[3)] `https://cran.r-project.org/`

1.4 Shrnutí

V úvodní kapitole bylo vysvětleno, že v této práci pod pojmem *systémový zdroj* budeme myslet CPU a RAM. CPU slouží jako mozek počítače, který vykonává jednotlivé instrukce, ze kterých se pak skládají složitější programy. V paměti typu RAM jsou uloženy jednotlivé programy a jejich využívaná data. Toto jsou jen dva ze systémových zdrojů, nicméně postupy aplikované v této práci by se daly aplikovat i na ostatní systémové zdroje.

Dále byly objasněny současné možnosti pro předpověď využití systémových zdrojů v počítačových zařízeních. Většina z nich je založena na metodě lineární regrese. Na lineární regresi je založen např. algoritmus LiRCUP, nebo ve spolupráci se statistickým softwarem *R* ji lze použít pro odhad zbývajícího místa na pevném disku. Nevýhodou této metody je, že pokud nedostane lineární model, vrací nepřesné výsledky. Nicméně tato metoda je v některých algoritmech dostačující a její výhodou je snadná implementace, nedokáže ovšem vystihnout závislosti v nelineárním modelu. Ve zbytku této práce tedy bude objasněna metoda, která by tyto závislosti měla umět vystihnout a měla by dávat lepší výsledky pro nelineární model.

2 Neuronové sítě

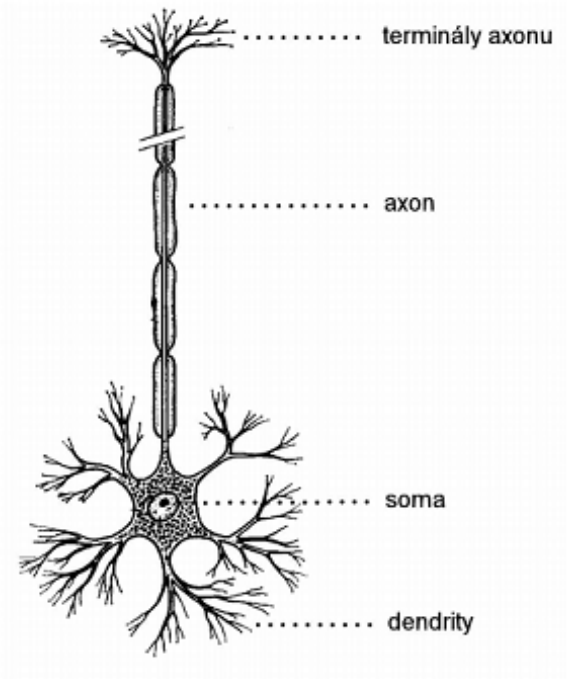
V této kapitole budou popsány základní principy neuronových sítí. Bude vysvětlen pojem *umělý neuron* a jeho podobnost s biologickým neuronem. Dále bude popsána architektura neuronových sítí, princip učení a nejpoužívanější učící algoritmus.

Neuronová síť je výpočetní model, spadající do oboru umělé inteligence, který je inspirován činností lidského mozku. Tento výpočetní model je založen na principech živých organismů.

2.1 Biologický neuron

Biologický neuron představuje jednu nervovou buňku a je základní jednotkou nervové soustavy. Člověk má 15–25 miliard neuronů a 300 miliard kontaktních ploch (synapsí). Biologický neuron je zobrazen na Obr. 2.1. Každý neuron se skládá z několika částí:

- soma - tělo neuronu,
- dendrity - vstupy neuronu,
- axon - výstup neuronu.



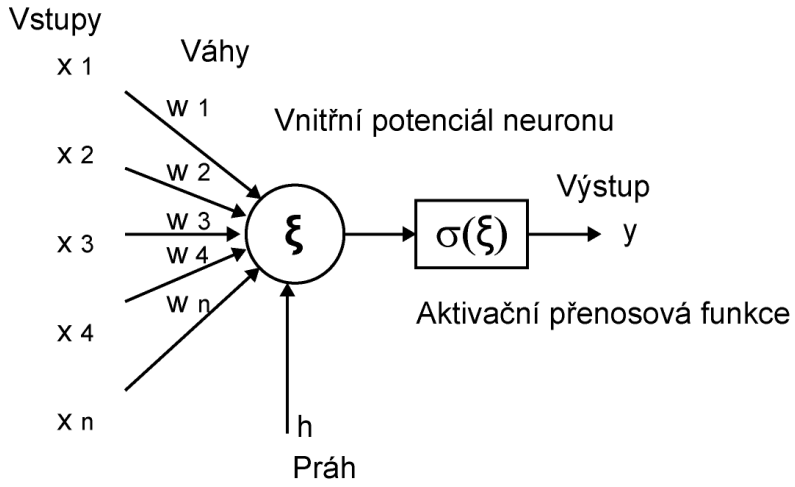
Obr. 2.1 Schéma biologického neuronu [6].

Neuron má několik dendritů a jeden axon. Mezi neurony se šíří signály (vzruchy). Pro přenos těchto singálů slouží synapse, což je v podstatě spojení dvou neuronů.

Při učení člověka dochází k vytváření synapsí mezi jednotlivými neurony. Pro naučení člověka tedy nestačí pouze neurony, ale je potřeba mezi nimi vytvořit správné vazby, což je důležité také pro učení neuronových sítí.

2.2 Matematický (formální) neuron

Model matematického neuronu vychází z biologického neuronu. Matematický neuron je zobrazen na Obr. 2.2. Má obecně n vstupů $x_1, ..., x_n$, jeden výstup y a ke každému vstupu náleží váha w_n. Hodnota h označuje práh neuronu.



Obr. 2.2 Schéma matematického neuronu [6].

Vnitřní potenciál neuronu se značí jako ξ a vyjadřuje součet součinů vstupů a jejich vah (rovnice 2.1).

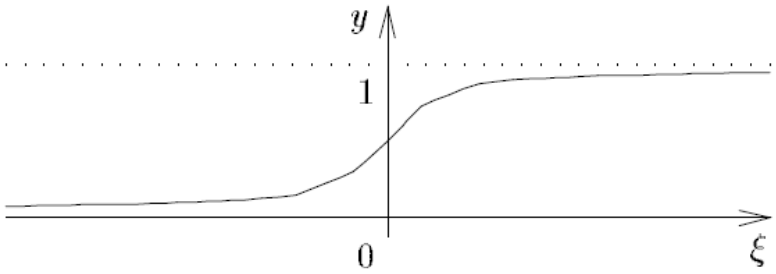
$$\xi = \sum_{i=1}^{n} w_i x_i + w_0 x_0 = \sum_{i=1}^{n} w_i x_i - h \tag{2.1}$$

Neuron přenáší hodnotu na výstup pouze pokud je excitován, tedy pokud vnitřní potenciál přesáhne prahovou hodnotu. Jakmile vnitřní potenciál přesáhne prahovou hodnotu, je na výstup neuronu y přiveden výstup z přenosové funkce (rovnice 2.2).

$$y = \sigma(\xi) \tag{2.2}$$

Přenosová funkce tedy určitým způsobem transformuje vnitřní potenciál na výstup. Může mít několik tvarů: *lineární*, *Perceptron*, *logistická sigmoida*, *hyperbolický tangens*. V této práci se bude pracovat s funkcí *logistická sigmoida* (viz podkapitola 2.5). Její průběh je uveden na Obr. 2.3. Tvar funkce je dán rovnicí 2.3.

$$\sigma(\xi) = \frac{1}{1 + e^{-\xi}} \tag{2.3}$$

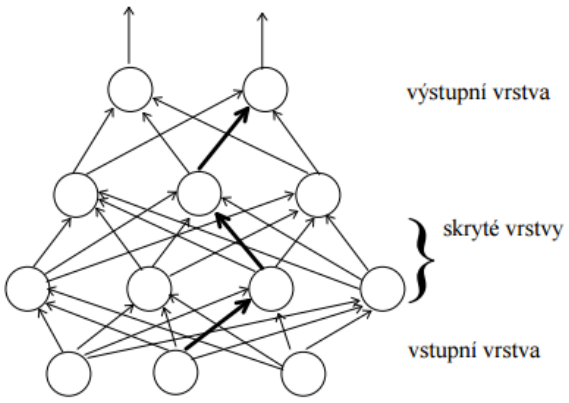


Obr. 2.3 Logistická sigmoida [6].

2.3 Neuronová síť

Každá neuronová síť je složena z formálních neuronů, které jsou vzájemně propojeny tak, že výstup jednoho neuronu je vstupem do (obecně i více) neuronů [7]. Obecně existují dva druhy neuronových síti: *cyklické* a *acyklické*. Cyklické sítě obsahují obsahují neurony, kde je aspoň jeden cyklus. Za cyklus se považuje, pokud se výstup z neuronu dokáže přes n ($n > 0$) neuronů dostat zpět do původního neuronu. Acyklické sítě neobsahují žádný cyklus.

Dále v této práci budeme uvažovat pouze acyklické sítě. Někdy se těmto sítím také říká vrstevnaté, jelikož samotné neurony jsou uspořádány do vrstev: *vstupní*, *skrytá*, *výstupní*. Pokud má síť alespoň jednu skrytou vrstvu, jedná se o *vícevrstvou síť*. Pokud má pouze vstupní a výstupní vrstvu, nazýváme tuto síť *jednovrstvou*. Výstup každého neuronu z nižší vrsty je vstupem do všech neuronů vyšší vrstvy. Tvar sítě je dán jeho topologií, která určuje, kolik vrstev a kolik neuronů v každé vrstvě bude daná síť mít. Například na Obr. 2.4 je zobrazena síť *3-4-3-2* se dvěma skrytými vrstvami.



Obr. 2.4 Neuronová síť *3-4-3-2* [7].

2.4 Učení neuronových sítí

Vstupní data se dělí na trénovací a validační množinu. Vstupními daty je množina dvojic, kde každá dvojice je složena z vektroru vstupů $\mathbf{X_n}$ (n značí počet vstupních neuronů) a vektoru jejich očekávaných výstupů $\mathbf{T_m}$ (m značí počet výstupních neuronů). Trénovací množina slouží k naučení neuronové sítě, validační množina pro kontrolu učení. Tyto dvě množiny by měly být navzájem disjunktní.

Data v trénovací množině se musí volit tak, aby pokryla co největší prostor všech možných kombinací. Zároveň je ale třeba dávat pozor, aby nedocházelo k jevu zvanému přeučení. Tento jev nastává tehdy, jakmile síť naučíme rozpoznávat příliš přesně pouze detaily a daná síť poté není schopna generalizovat. *Generalizace* neuronové sítě je schopnost odvodit souvislosti z naučených dat.

Pro trénování neuronových sítí lze využít dva typy učení:

- s učitelem,
- bez učitele.

Při učení s učitelem jsou předkládány kromě vstupních dat také očekávané výsledky. Na základě rozdílu očekávaných a vypočítaných výsledků poté dochází ke korekci vah. Naproti tomu, při učení bez učitele nejsou předkládány žádné očekávané výsledky. Většinou pracují na principu shlukování, kdy jsou vstupní data tříděna do skupin. Tento způsob učení je vhodný např. v dataminingu.

V této práci bylo využito pouze učení s učitelem, kdy na vstup byly předkládány naměřené historické hodnoty a za očekávaný výsledek byla vybrána aktuální hodnota.

2.5 Algoritmus backpropagation

Nejrozšířenějším algoritmem pro učení neuronových sítí je algoritmus *backpropagation* (zpětné šíření chyby), který spadá do kategorie učení s učitelem. Tento algoritmus pracuje na principu hledání nejmenší celkové chyby. Chybová funkce je dána součtem druhých mocnin rozdílu očekávaného výstupu t_i a skutečného výstupu y_i (rovnice 2.4). Hodnota p označuje počet prvků v trénovací množině. Při učení hledáme kombinaci vah, pro kterou je chyba E nejmenší [9].

$$E = \frac{1}{2}\sum_{i=1}^{p}(t_i - y_i)^2 \tag{2.4}$$

Algritmus se skládá z několika fází:

1. Inicializace vah náhodnými malými čísly.

2. Výběr vstupního vektoru z trénovací množiny.

3. Dopředné šíření vstupního signálu (feedforward). Během této fáze dochází k vyhodnocování hodnot, popř. chyby na výstupních neuronech.

4. Zpětné šíření chyby (backpropagation), vyhodnocování chyby na nižších vrstvách.

5. Aktualizace vah.

6. Pokud je celková chyba menší než povolená odchylka nebo pokud počet iterací přesáhl danou mez, algoritmus končí, jinak pokračuje na bod 2.

Algortimus backpropagation vyžaduje nastavení dvou parametrů: *krok učení* (learning rate) a *momentum*. Parametr *learning rate* ovlivňuje rychlost učení tím, že určuje velikost změny vah při zpětném šíření chyby. Pokud je nastavena příliš nízká hodnota, učí se síť příliš pomalu. Pokud je ale nastavena vysoká hodnota, síť není schopna se naučit podstatným detailům. Pomocí parametru *momentum* lze přičíst k aktuální změně váhy také část její předchozí změny. Oba parametry mohou nabývat hodnoty $< 0, 1 >$.

Aktivační funkce pro tento algoritmus musí splňovat dvě základní podmínky:

- Musí být spojitá.
- Musí být diferencovatelná, jelikož pracuje s gradientem. Gradient je vektor určující směr největšího spádu chybové funkce.

Tyto podmínky splňuje sigmoidální aktivační funkce, která se v algoritmu backpropagation používá nejčastěji [9]. Nevýhodou této funkce je, že hodnota chyby může spadnout do lokálního minima (Obr. 2.5) a nemusí být nalezeno globální minimum. Naopak výhodou této funkce je její jednoduchost.

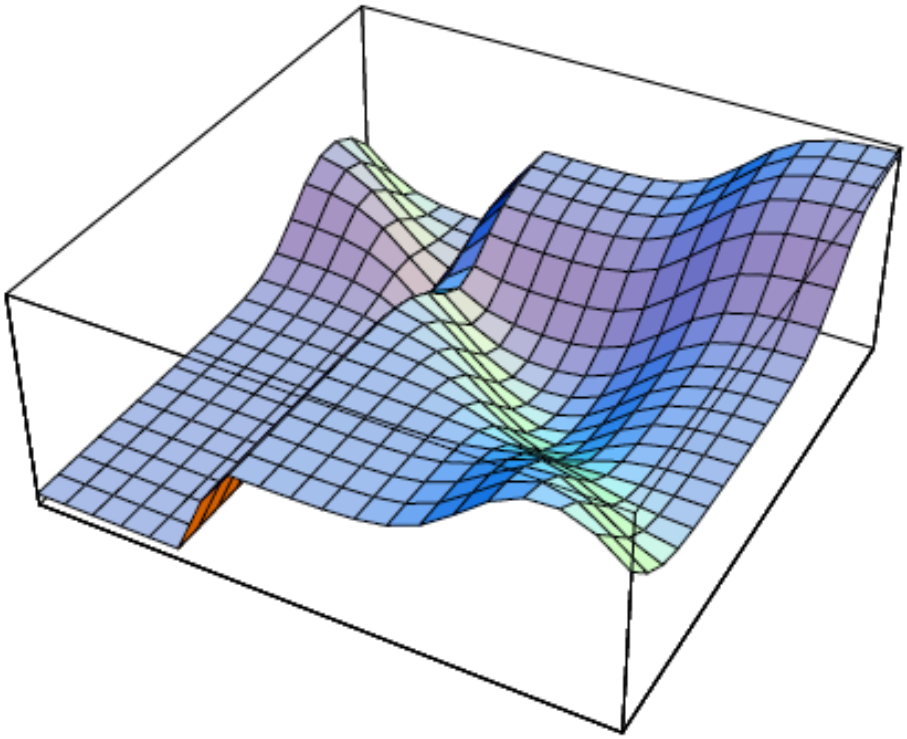
2.6 Algoritmus resilient propagation

Algoritmus *resilient propagation* (Rprop) představuje jednu z dalších možností pro učení sítě. Funguje na podobném principu jako výše zmíněný algoritmus *backpropagation*, nicméně přichází s některými vylepšeními [13]:

1. Je rychlejší než backpropagation (viz kapitola 5.2.1).

2. Nevyžaduje nastavení žádných počátečních parametrů, čímž odpadá starost s tím, jak určení parametrů ovlivní celkový průběh učení.

Algoritmus backpropagation používá velikost parciální derivace pro výpočet nové váhy. Na Obr. 2.6 si ukážeme, kde může nastat problém [13]. Modrou křivkou je zde



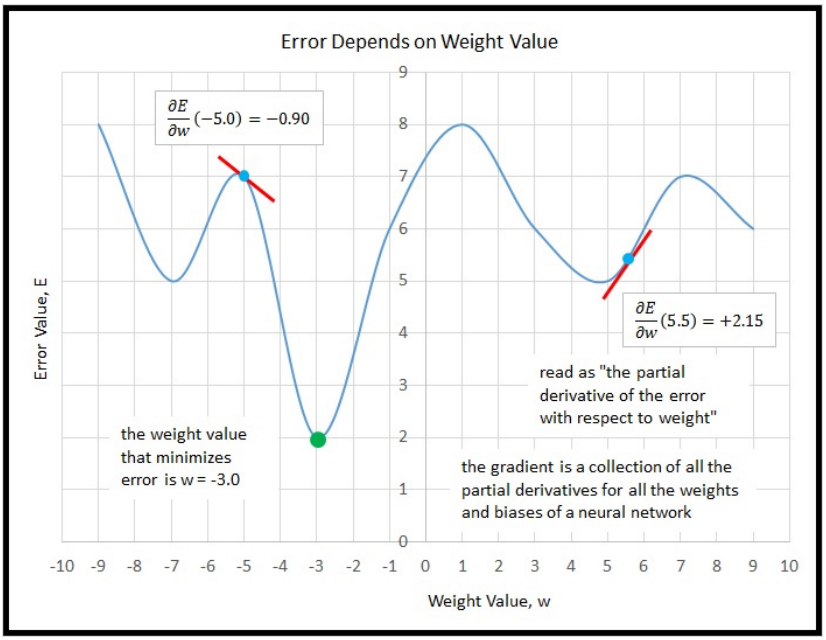
Obr. 2.5 Lokální minimum chybové funkce [9].

znázorněna velikost chyby v závislosti na nastavení váhy neuronu. Předpokládejme, že aktuální váha má hodnotu -5, algoritmus vyhodnotí poměrně strmý gradient a spočítá přírustek váhy jako +7. Tím pádem hodnota váhy se přesune do bodu +2. Tím se ale přeskočí globální minimum v bodě -3. V následující iteraci by mohlo dojít k návratu na hodnotu -5 a algoritmus by takto mohl kmitat mezi dvěma hodnotami. V praxi to ale funguje tak, že je zvolena nízká hodnota parametru *learning rate*, což má ale za následek velmi pomalý průběh učení.

Algoritmus *Rprop* tedy přichází jako vylepšení algoritmu *backpropagation* a přináší dvě změny:

- Pro výpočet přírustku vah používá pouze znaménko gradientu.
- Jak již bylo zmíněno algoritmus nevyžaduje oproti *backpropagation* nastavení počátečních parametrů. A to právě z toho důvodu, že nepoužívá *learning rate* pro všechny váhy společně, ale pamatuje si jednotlivé váhové přírustky pro každou váhu zvlášť.

Na pseduokódu v ukázce 2.7 je uveden princip algoritmu. Lze na něm vyčíst, že pokud předchozí a aktuální parciální derivace mají stejné znaménko, znamená to, že



Obr. 2.6 Průběh chybové funkce metody backpropagation [13].

váha nepřeskočila hodnotu s minimální chybou (narozdíl od *backpropagation*, jak bylo popsáno výše). Pokud se ale znaménka liší, znamená to, že hodnota minima již byla překročena, proto je váha nastavena na předchozí hodnotu a je snížen přírustek, což způsobí, že v další iteraci nebude přírustek tak velký a zvyšuje se šance, že funkce klesne do minima. Tento mechanismus dovoluje dynamicky zvětšovat přírustky, a tím pádem zvyšuje celkovou rychlost algoritmu. Tento fakt bude potvrzen také v praktické části, kde se bude měřit doba učení pro oba algoritmy.

```
while epocha < max počet epoch do
    spočítej gradient pro všechny vstupní prvky z trénovací množiny
    pro každou váhu
      if (předchozí a aktuální parciální derivace mají stejné znaménko)
         zvyš naposledy použitý přírustek
         akutalizuj váhy podle nového přírustku
      else
         sniž naposledy použitý přírustek
         vrať váhu na předchozí hodnotu
      předchozí_přírustek = aktuální_přírustek
      předchozí_gradient = aktuální_gradient
   epocha++
end while
```

Pseudokód 2.7 Popis činnosti algoritmu *resilient propagation*.

2.7 Využití neuronových sítí

Neuronové sítě mohou být využity na problémy, které nelze zapsat algoritmem, nebo opravdu jen velmi těžce. Obecně platí, že je třeba mít dostatečný počet vstupních dat k naučení sítě a odpovídající výpočetní výkon pro učení. Jen pro představu zde bude uvedeno několik možných využití, nicméně neuronové sítě v sobě skrývají velký potenciál a jejich použití je velmi široké. Náplní této práce ovšem není zaměřit se na všechny obory využití, ale tyto neuronové sítě využít pro predikci systémových zdrojů v počítači.

Neuronové sítě se dají využít například pro:

- rozpoznávání vzorů (pattern recognition),
- diagnóza onemocnění na základě příznaků,
- komprese počítačových obrázků,
- předpověď vývoje finančního trhu,
- předpověď počasí,
- umělá inteligence v počítačových hrách.

2.8 Shrnutí

V této kapitole byly popsány základní poznatky pro práci s neuronovými sítěmi. Základní jednotkou neuronových sítí je matematický neuron, jehož model je inspirován biologickým neuronem. V této práci budeme pracovat s dopřednými neuoronovými sítěmi, které obsahují vstupní, skryté a výstupní vrstvu. Stejně jako se lidský mozek potřebuje naučit určité závislosti, tak neuronová síť se musí pro správný výstup výsledků nejdříve naučit. Nejrozšířenějším algoritmem je algoritmus *backpropagation*, ovšem byla představena i vylepšená verze tohoto algoritmu: *resilient propagation*. Tato vylepšená verze je oproti původnímu *backpropagation* rychlejší.

3 Predikce s využitím neuronových sítí

Následující kapitola popisuje navržení neuronové sítě pro predikci časových řad. Konkrétně tedy využití systémových zdrojů na základě předchozích hodnot.

Časová řada je posloupnost skalárních hodnot, které se mění s časem. Pro potřeby této práce budeme tuto časovou řadu vzorkovat s určitou periodou. Tím nám vznikne diskrétní posloupnost hodnot. Pokud budeme předpokládat, že počítač je dynamický systém a naměřené hodnoty využití jeho zdrojů jsou proměnné, může docházet k tomu, že naměřená data mohou mezi sebou mít určité závislosti.

Pro předpověd systémových zdrojů budeme využívat N předchozích hodnot časové řady (viz rovnice 3.1). Budeme se tedy snažit o nalezení funkce, která bude schopna odhadnout hodnotu x v čase $t + 1$ na základě N předchozích hodnot [15].

$$x(t+1) = f(x(t), x(t-1), ..., x(t-N)) \tag{3.1}$$

Tyto řady mohou reprezenovat různé časové proměnné, například [10]:

- vývoj teploty v budovách,
- cenu zboží na trhu,
- počet narozených dětí ve městě,
- množství stočené vody v domácnosti.

Jak lze vidět, předpověď časových řad pomocí neuronových sítí může být užitečná v mnoha oborech. Podmínkou je dostatečný počet vstupních dat. Pro tuto práci byla jako vstupní časová řada zvolena množina naměřených hodnot využití CPU a RAM.

3.1 Typy zpracování číslených řad

Na analýzu časových se lze dívat několika způsoby [10]:

- předpověď budoucího vývoje řady,
- klasifikace řad do kategorií,
- popis řady pomocí modelu,
- mapování časové řady na jinou.

Nejrozšířenějším typem zpracování je předpověď budoucího vývoje [10]. Pokud se na časovou řadu budeme dívat jako na funkci času, lze pro předpověď následujících

hodnot využít rovnici 3.1. Předpověď tedy spočívá v nalezení jisté funkce, která je schopna z N předcházejících hodnot odhadnout jednu budoucí hodnotu.

V mnoha případech se zkoumá, jestli budoucí vývoj řady bude mít klesající nebo stoupající tendenci. V tomto případě tedy mluvíme o klasifikaci řady do některé z kategorií. Obecně platí, že při klasifikaci řady do některé z kategorií hledáme funkci, která přiřazuje jednu z kategorií časové řadě.

Popis řady pomocí modelu je již obsažen v obou typech zpracování popsaných výše: předpovědi a klasifikaci. Model je vytvářen pomocí předpovědi, kdy na vstup přivádíme již předpovězené hodnoty. Tím pádem jsme schopni vytořit model, který bude předpovídat hodnoty.

Vytvoření modelu je v podstatě mapování časové řady sama na sebe. Existuje ovšem možnost mapování časové řady na jinou. V rámci této práce můžeme uvažovat příklad, kdy budeme zkoumat využití CPU na základě využití RAM. Lze toho dosáhnout dvěma způsoby. V jednodušší formě bude trénovací množina obsahovat jako vstup dvojici hodnot (CPU a RAM) a na výstupu bude požadovaná hodnota. Dalším (a také složitějším) způsobem je modelování obou řad a nalezení mapovací funkce mezi řadami.

V praktické části této práce bude hlavně představena předpověď časových řad. Také bude vytvořen model, který by měl předpovídat budoucí hodnoty i z již předpovězených hodnot. V podkapitolách 5.4.3 a 5.4.4 bude ukázán i pokus o předpověď za využití více řad. Tedy předpověď CPU za využití naměřených hodnot CPU a RAM a předpověď RAM za využití hodnot CPU a RAM.

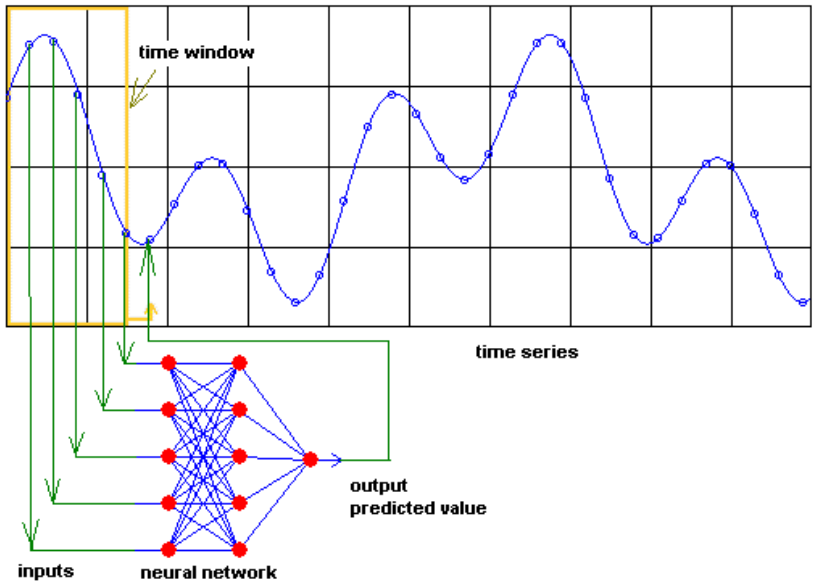
3.2 Metoda posunujícího se okna

Pro předpovídání hodnot časové řady použijeme techniku posunujícího se okna (anglicky *sliding window*). Tato metoda predikuje na základě N předchozích hodnot číselné řady. Jakmile je predikce hodnoty dokončena, okno se posune dopředu. Tato metoda je graficky znázorněna na Obr. 3.1.

Vstupy do sítě jsou tedy předchozí naměřené hodnoty, výstupem je predikovaná hodnota. Struktura sítě tedy bude mít tvar odpovídající Obr. 3.2.

3.3 Odhad velikosti okna

Pro metodu *sliding window* je potřeba zvolit velikost okna, jinými slovy určit počet vstupních neuronů. Pokud je okno příliš malé, nebudeme mít tolik vstupních dat a odhad nemusí být přesný. Na druhou stranu, pokud bude okno příliš velké, může se stát, že hodnoty, které jsou v okně nejstarší, mohou představovat určitý druh šumu a budou zkreslovat celkovou predikci. Jednou z heuristik pro určení velikosti okna je metoda *false nearest neighbor*.



Obr. 3.1 Princip metody sliding window [14].

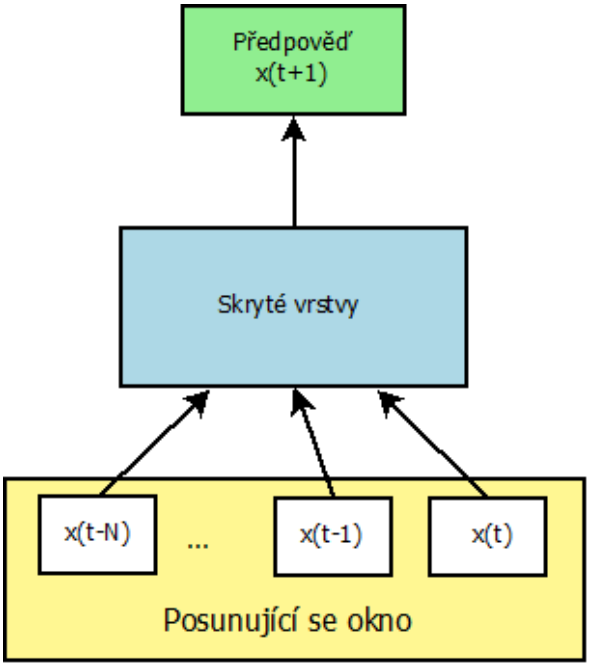
Metoda *false nearest neighbour* slouží pro určení minimální dostatečné velikosti dimenze N. Alogritmus tedy hledá minimální dimenzi, která má minimální počet falešných sousedů. Nejdříve jsou vytvořeny časově posunuté vektory. Dále je pro každý bod v daném vektoru nalezen nejbližší soused. Přidáním další dimenze by měly být zachovány jednotlivé vztahy, pokud se ve vyšší dimenzi najde bližší soused, je původní označen jako falešný soused.

Princip algoritmu je následující [17]: pro každý bod R_i časové řady algoritmus hledá nejbližšího souseda R_j v m-rozměrném prostoru. Spočítá jejich vzdálenost $|R_i - R_j|$ a opakovanými iteracemi se počítá prahová hodnota R_d, viz rovnice 3.2. Pokud prahová hodnota R_d překročí jistou prahovou hodnotu, je bod považován za falešného souseda.

$$R_d = \frac{|R_{i+1} - R_{j+1}|}{|R_i - R_j|} \tag{3.2}$$

3.4 Time delay neural networks

Další možností pro předpověď časových řad jsou tzv. *Time delay neural networks* (TDNN). Tyto neuronové sítě modelují určitý druh paměti. Princip těchto sítí je založen na tom, že vstupní hodnoty jsou do skrytých vrstev předávány s určitým zpožděním. V klasických *feed forward* sítích vstupuje do sítě celý kontext (vstupní okno). V TDNN nejprve síť zpracuje část vstupu a s vyššími vrstvami se učí na dalších částech vstupního datasetu. Proto mají vyšší vrstvy schopnost rozpoznat lépe vazby mezi vstupními daty. Každá vrstva v TDNN pracuje nad jiným časovým detailem dat, který se zvyšuje s počtem vrstev sítě. Jinými slovy, vstupy přichází do skrytých vrstev v různých časech,



Obr. 3.2 Struktura sítě u sliding window.

a proto mohou být uloženy tak dlouho, aby dokázaly ovlivnit následné vstupy.

3.5 Shrnutí

V předchozí kapitole bylo ukázáno, jak mohou být pomocí neuronových sítí zpracovávány časové řady. Jednou z možností je předpověď budoucích hodnot. Dále můžeme pouze klasifikovat řady do některé z kategorií, snažit se vytvořit model nebo pokusit se o namapování časové řady na jinou. Tyto metody budou v praxi předvedeny na implementované aplikaci v praktické části.

Pro předpověď budoucích hodnot bude využito tzv. *sliding window*, tedy posunující se okno, představující vzorek dat, který bude právě vstupovat do neuronové sítě. Další možností je použít tzv. *Time dealy neural networks*, které jsou schopny modelovat určitý druh paměti.

II. PROJEKTOVÁ ČÁST

4 Implementace

Součástí této práce je kromě teoretické části také praktická ukázka využití poznatků nabytých v předchozích kapitolách. Výstupem této projektové části je konzolová aplikace, která umožňuje předpovídat využití CPU a RAM. Dalším úkolem praktické části bude srovnání výstupu z aplikace s již existujícím komerčním produktem. Budou také prováděny experimenty s neuronovou sítí za účelem dosažení nejoptimálnější předpovědi.

4.1 Seznámení s nástroji

Aplikace je implementována v programovacím jayzce C# za použití knihovny Encog[1)]. C# je objektově orientovaný programovací jazyk běžící na platformě .NET, tudíž vývoj i testování aplikace probíhalo na operačním systému Microsoft Windows. Pro vývoj aplikace bylo použito vývojové prostředí Microsoft Visual Studio.

Jazyk C# byl zvolen z několika důvodů:

- objektově orientovaný přístup,
- podpora pro knihovnu Encog (vysvětleno dále),
- jednoduchá práce s databází Microsoft SQL Server,
- jednoduchá práce při manipulaci s textovými soubory.

4.2 Knihovna Encog

Encog je knihovna umožňující efektivní práci se strojovým učením. Popdporuje algoritmy pro práci s nástroji jako *Support Vector Machines* (metoda podpůrných vektorů), *neuronové sítě*, *bayesovské sítě*, *genetické programování* nebo *skrytý Markovův model* [8]. Pro nás je ale důležitá hlavně podpora pro práci s neuronovými sítěmi. Encog obsahuje implementaci jak pro C# tak i pro jazyk Java.

Framework Encog obsahuje celou řadu metod pro práci s neuronovými sítěmi, dále se ale v této práci budeme zajímat pouze o metody, které jsou využitelné v této práci:

- normalizace dat,
- vytvoření neuronové sítě,
- trénování neuronové sítě,
- předpověď pomocí neuronové sítě.

[1)] `http://www.heatonresearch.com/encog/`

4.3 Aplikace

Výsledná aplikace byla naimplementována v konzolové podobě. Nápověda pro práci s konzolovou aplikací je uvedena v příloze.

Řídící část aplikace je ve třídě `Runner`. Tato třída na základě poskytnutých parametrů určuje běh celé aplikace. Pro přehlednost jsou zde jako první všechny příkazy pohromadě, z nich by měl vyplynout význam jednotlivých částí, a poté budou jednotlivé části popsány detailněji:

- `data = DataProvider.GetData()`. Získání dat, která budou využita pro trénování a posléze pro předpověď.
- `predictor.CreateNetwork()`. Na základě parametrů poskytnutých aplikaci se vytvoří neuronová síť s danou strukturou.
- `predictor.GenerateTraining()`. Vytvoření trénovací množiny z dat.
- `predictor.Train(network, training)`. Učení sítě na trénovací množině.
- `predictor.DoPredict(network)`. Předpovídání hodnot v testovací množině.
- `resultProcessor.SaveResults()`. Uložení výstupu.

4.3.1 Získání dat

Pro získání dat bylo využito základních metod platformy .NET. Aplikace umí data získat dvěma způsoby: z databáze nebo ze souboru ve formátu CSV[2)]. Pro tyto účely bylo vytvřeno rozhraní `IDataProvider`, které obsahuje metodu `GetData()`. Tato metoda vrací seznam objektů typu `SystemResourcesUsage`, které v sobě uchovávají následující hodnoty:

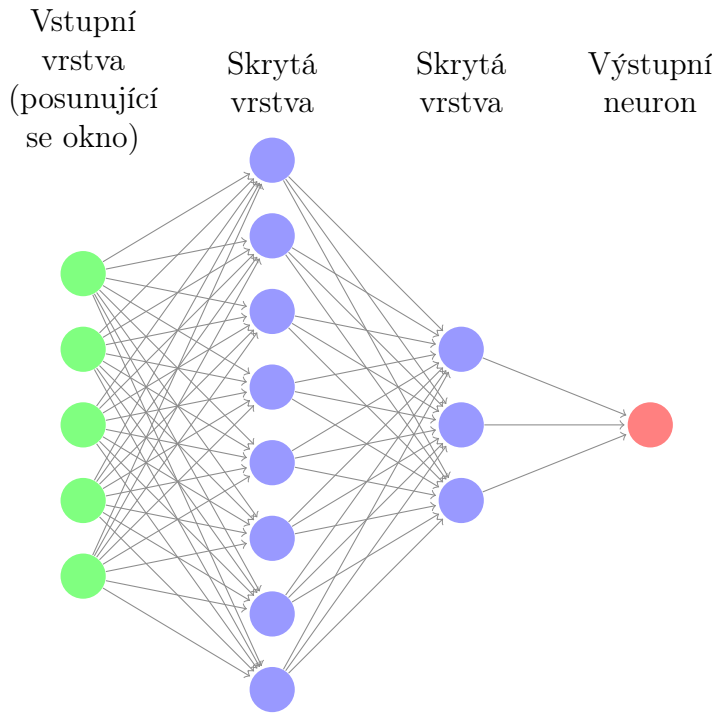
- `DateTime Timestamp` udávající čas naměření dané hodnoty,
- `double CpuAvgLoad` udávající využití CPU v procentech,
- `double RamAvgLoad` udávající využití RAM v procentech.

Byly tedy vytvořeny dvě třídy `CsvDataProvider` a `DatabaseDataProvider`, které implementují rozhraní `IDataProvider`. To, která z těchto tříd bude použita, určují vstupní parametry. Pokud je vstupní metodou zvolen textový soubor, je potřeba zadat pouze cestu k tomuto souboru. V případě výběru dat z databáze je třeba zadat databázový server, jméno databáze a přístupové údaje účtu, který má právo čtení z této databáze.

[2)] Formát souboru, kde jsou jednotlivé hodnoty na řádku odděleny čárkou nebo středníkem.

4.3.2 Vytvoření neuronové sítě

Pro vytvoření sítě (stejně jako pro veškerou práci se sítí) byla použita knihovna Encog. Encog obsahuje třídu `BasicNetwork`, která implementuje neuronovou síť. Tato třída obsahuje metodu `AddLayer(n)`, kde parametr n určuje počet neuronů ve vrstvě. Konfiguraci sítě je možno nastavit pomocí parametru parametru `NetworkConfig`. Parametr obsahuje celočíselné hodnoty oddělené pomlčkou, kde jednotlivá čísla určují počet neuronů ve vrstvě. Tento parametr ovšem určuje pouze vstupní a vnitřní vrstvy sítě. Výstupní neuron bude vždy jeden. Pokud tedy parametr `NetworkConfig` bude obsahovat hodnotu `5-8-3`, bude síť vypadat jako na Obr. 4.1.



Obr. 4.1 Ukázka konfigurace sítě, pokud bude parametr `SiteConfig` roven hodnotě `5-8-3`.

Kód pro vytvoření sítě na základě poskytnutého parametru je popsán na následující ukázce.

```
BasicNetwork network = new BasicNetwork();
networkConfig.Split('-').ToList().ForEach(layer =>
{
    network.AddLayer(new BasicLayer(int.Parse(layer)));
});
network.AddLayer(new BasicLayer(1));
```

4.3.3 Vytvoření trénovací množiny z dat

Knihovna Encog obsahuje podporu pro vytváření trénovací, potažmo validační množiny (v podkapitole 4.3.4 bude vysvětleno jak jsou tyto dvě množiny ve frameworku Encog provázány). Jako první krok před vytvořením testovací množiny je třeba data normalizovat. Pro normalizaci je třeba zvolit interval, do kterého budou data převedena. Pro účely této práce byla data normalizována do intervalu $< 0, 1 >$.

Pro vytvoření trénovací množiny jsou důležité následující 3 třídy:

- `TemporalMLDataSet`
- `TemporalDataDescription`
- `TemporalPoint`

`TemporalMLDataSet` reprezentuje třídu, která implementuje tzv. *temporal data set*. Encog pro předpověď budoucích hodnot používá tzv. *temporal data*, což může být chápáno jako data, která se mění s časem. Objekt typu `TemporalMLDataSet` obsahuje jeden nebo více objektů `TemporalDataDescription`, které reprezentují jednu konkrétní vstupní jednotku do neuronové sítě. V konstruktoru této třídy se určuje, zdali je tato jednotka určena pouze pro předpověď nebo se budou předpovídat její hodnoty. Každá tato jednotka obsahuje datové pole objektů typu `TemporalPoint`, které drží naměřené hodnoty.

4.3.4 Trénování sítě

Pro trénování sítě je nejdřív potřeba zvolit trénovací metodu: *backpropagation* nebo *resilient propagation*, obě metody jsou popsány v kapitole 2.5, resp. 2.6. Učící algoritmus se volí pomocí vstupního parametru `TrainingMethod`. Obě trénovací množiny mají zabudovanou podporu ve knihovně Encog. Encog obsahuje rozhraní `IMLTrain`, ze kterého dědí všechny trénovací metody podporované ve frameworku Encog, např. *ManhattanPropagation*, *QuickPropagation*, apod.

Trénovaní v této aplikaci vypadá velice jednoduše, jelikož rozhraní `IMLTrain`, obsahuje metodu `Iteration()`, která vykoná jednu iteraci učení.

Pro srovnávací účely jednotlivých metod bylo implementováno měření času stráveného nad učením sítě. Dále také můžeme na výstupu vidět, kolik epoch bylo třeba k naučení sítě se zadanou chybou. Níže je popsána implementace učení neuronové sítě.

```
Stopwatch sw = new Stopwatch();
int iterationCnt = 0;
sw.Start();
```

```
do
{
    //train je objekt implementujici IMLTrain
    train.Iteration();
    iterationCnt++;
} while (train.Error > specifiedError);

sw.Stop();
Console.WriteLine($"Cas trenovani: {sw.Elapsed}");
Console.WriteLine($"Pocet iteraci: {iterationCnt}");
```

Jak lze vidět z předchozí ukázky kódu, učení končí pokud trénovací chyba klesne pod určitou mez. S tím se pojí i termín validační množina. Jak již bylo vysvětleno, validační množina se používá pro testování chyby v průběhu trénování. Validační množinu si tedy Encog vytvoří sám z poskytnutých dat, takže ji z pohledu vytváření aplikace nemusíme řešit a data musíme rozdělit na trénovací množinu a testovací množinu.

4.3.5 Předpovídání hodnot pomocí neuronové sítě

Pro předpovídání budoucích hodnot slouží testovací množina. Budeme používat techniku posunujícího se okna, popsanou v kapitole 3.2. Předpokládejme tedy, že máme vytvořenou testovací množinu dat, kde tato data budeme předkládat na vstup sítě. Počet vstupních neuronů je dán velikostí okna, popř. počtem vstupních kritériií. Pokud tedy budeme předpovídat využití RAM pomocí posledních deseti hodnot, bude počet vstupních neuronů (velikost okna) roven 10. Pokud budeme chtít experimentovat se sítí a předpověď postavit i na posledních deseti naměřených hodnotách využití CPU, bude velikost okna rovna dvaceti neuronům (10 pro CPU a 10 pro RAM).

Dále se data dají předpovídat v tzv. *uzavřené smyčce*. Postup pro předpovídání v uzavřené smyčce je téměř totožný jako regulérní predikce dat, ovšem probíhá nad jinými daty. Pokaždé když se spočítá predikce na určitém okně, uloží se spočítaná hodnota do pole `closedLoop`, což je pole číselných hodnot. Predikce v uzavřené smyčce tedy potom probíhá nad tímto polem. V principu tedy získáváme dvě predikce: regulérní predikci z naměřených hodnot a predikci v uzavřené smyčce, která se provádí na již předpovězených hodnotách.

V kódu je algoritmus zapsán na následující ukázce zdrojového kódu:

```
//evaluate_start zacatek testovaci mnoziny
//evaluate_end konec testovaci mnoziny

for (int i=evaluate_start+window_size; i < evaluate_end; i++)
```

```
{
 //vytvoreni okna
 input = BasicMLData(velikost_okna);

//naplneni okna
prediction = GetPredictionValue(y, network, false);

//naplneni okna pro uzavrenou smycku
closedLoopPrediction = GetPredictionValue(y, network, true);

//nastaveni hodnoty pro dalsi predpovedi uzavrene smycky
closedLoop[i] = prediction;

//denormalizace dat
denormalizationCoeficient = GetDenormalizationCoeficient(y);

//vynasobeni hodnot prediction a closedLoopPrediction
//denormalizacnim koeficientem

//ulozeni vysledku do vysledne struktury
}
```

Jak lze vidět v předcházející ukázce kódu, posunující okno je implementováno třídou `BasicMLData`. Samotná predikce probíhá v metodě `GetPredictionValue`, která jednak naplní posunující se okno, a zároveň zavolá metodu `network.Compute(vstupni_okno)`, která z poskytnutého okna vypočítá předpovězený výstup. Jako třetí parametr přijímá metoda `GetPredictionValue` přepínač, který určuje, zdali má předpovídat z naměřených hodnot, nebo z uzavřené smyčky. Pro regulérní predikci tedy bude okno vytvořeno z naměřených dat, a pro predikci v uzavřené smyčce je okno vytvořeno z předpovězených hodnot. Do výsledné struktury se ukládají denormalizovaná data. Data se jednoduše denormalizují podle rovnice 4.1, jelikož známe normalizovanou i naměřenou hodnotu, ze které tato normalizovaná hodnota vznikla. Díky tomu jsme schopni získat denormalizační koeficient, který předpovězenou hodnotu vrátí z normalizovaného oboru hodnot zpět do oboru hodnot, ve kterém jsou naměřená čísla.

$$denormalizacni\_koeficient = \frac{namerena\_hodnota}{normalizovana\_hodnota} \tag{4.1}$$

Po získání předpovězených hodnot a vynásobení těchto hodnot denormalizačním

koeficientem přichází na řadu jednoduchý postprocessing dat. Tento postprocessing spočívá v ořezání hodnot, které nepasují do intervalu hodnot. Pokud tedy předpovídáme využití CPU v procentech, je zřejmé, že předpovězená hodnota může nabývat hodnoty v intervalu <0, 100>. Pokud tedy některá z předpovězených hodnot nepasuje do zadaného intervalu je jí automaticky přiřazena mezní hodnota, tedy 0, resp. 100. Pro zachování maximální názornosti a jednoduchosti ukázky je zde uvedena pouze část kódu metody, kde se pracuje s předpovídáním hodnot, nicméně k práci je přiložen zdrojový kód celé aplikace, kde lze nalézt také ukládání výsledků do výsledné struktury apod.

Metoda pro předpověď vrací seznam objektů type `Result`, které v sobě drží následující hodnoty:

- `DateTime Timestamp` značí čas, kdy byla hodnota naměřena.
- `double ExpectedCpu` značí hodnotu CPU, která byla naměřena.
- `double ExpectedRam` značí hodnotu RAM, která byla naměřena.
- `double Computed` značí předpovězenou hodnotu.
- `double ClosedLoop` značí předpovězenou hodnotu z uzavřené smyčky.

4.3.6 Zobrazení výsledků

Jak již bylo zmíněno, aplikace je implementována konzolovou formou, proto její výstup je v textové podobě. Aplikace je schopna na základě poskytnutého parametru `resultType` zobrazit výstup ve dvou podobách:

1. Výstup ve formátu CSV. Tento formát je již použitý pro načítání dat ze souboru. Jednotlivé hodnoty v řádku jsou odděleny středníkem. Tento soubor pak lze otevřít v v některém z tabulkových procesorů, jako např. Micorosoft Excel[3)], nebo LibreOffice Calc[4)]. V těchto tabulkových procesorech lze z tabulky vyrobit graf. Pro názorné zobrazení výstupu z této práce je nejvhodnější spojnicový graf, kde na ose x je znázorněn čas a na ose y jsou vyneseny jednotlivé hodnoty (ať už naměřené, nebo předpovězené).
2. Výstup ve formátu grafu a zobrazený pomocí knihovny CanvasJS[5)]. Tato knihovna slouží pro zobrazení grafů ve webových prohlížečích. CanvasJS je knihovna napsaná v jazyce Javascript a HTML5. Pro zobrazení grafu jsou potřeba dva soubory: javascriptový soubor, obsahující knihovní funkce pro práci s grafem, a také

[3)] https://products.office.com/cs-cz/excel
[4)] https://www.libreoffice.org/discover/calc/
[5)] http://canvasjs.com/

`.html` soubor obsahující samotnou stránku a kód pro vytvoření grafu a naplnění grafu hodnotami. Hodnoty jsou předány ve formátu JSON [6]. Aplikace tedy umí vytvořit z výsledků objekty ve formátu JSON a tyto objekty uložit v textové podobě do grafu CanvasJS.

Pro detailnější práci s výsledky je vhodnější použít první variantu, s použitím formátu CSV a pozdějším zobrazením těchto dat ve formě spojnicového grafu. Nicméně pro okamžité zkontrolování výsledku lze použít i druhou metodu zobrazení, tedy grafy pomocí knihovny CanvasJS. V aplikaci byla převážně použita knihovna CanvasJS.

Výstupní metody jsou reprezentovány třídami `CsvResultsProcessor` a `ChartResultsProcessor`, které dědí z rozhraní `IResultsProcessor`.

Dalším výstupem aplikace jsou data, která nemá smysl zobrazovat v tabulkách. Tato data jsou pouze vypsána na výstup konzole pro informativní a experimentární účely uživatele. Jedná se o celkový čas trénování a o počet iterací při trénování.

4.4 Shrnutí

Jedním z výstupů z praktické části je konzolová aplikace implementující nabyté poznatky v předchozích kapitolách teoretické části. Tato aplikace je naimplementována v jazyce C#. Pro práci s neuronovými sítěmi byla použita knihovna Encog, která v sobě zapouzdřuje nízkovrstevní funkce pro práci s neuronovými sítěmi. Aplikace má celou řadu nastavení, která jsou uvedena v příloze. Pomocí těchto parametrů má uživatel možnost přizpůsobit chování aplikace pro své účely. Aplikace umí ze získaných dat předpovědět budoucí hodnoty a tyto hodnoty posléze zobrazit v dané formě. Dále byly v této kapitole představeny ukázky kódu z aplikace. Ukázky byly vybrány tak, aby v praxi ukázaly použití neuronových sítí za pomoci knihovny Encog.

[6] JavaScript Object Notation - formát pro výměnu textových dat

5 Experimenty se sítí a s daty

V následující kapitole bude ukázán výstup z naimplementované konzolové aplikace. Pomocí těchto výstupů bude poté provedeno několik experimentů se sítí, s daty, s metodou učení apod. Pro zobrazení výstupu bude použita již zmíněná knihovna CanvasJS.

5.1 Vstupní data

Jak bylo zmíněno v teoretické části, pro předpověď budou použita data z časové řady. Tato časová řada se skládá z naměřených hodnot. Historická data byla naměřena v reálné síti na několika zařízeních pomocí programu *Network Performance Monitor*[1)] od firmy SolarWinds. Tento program se mimo jiné zabývá měřením statistik na jednotlivých zařízeních v síti. Hodnoty jsou měřeny po 10 minutách a následně uloženy do SQL databáze. Pro potřeby této práce tedy vstupní data musí být získána z databáze pomocí SQL dotazů nebo ze souboru ve formátu CSV, kam byla databáze vyexportována. Po určité době jsou tato data v databázi agregována, a díky tomu je možné data získat s určitým detailem:

- vzorkování po 10 minutách,
- vzrokování po 1 hodině,
- vzorkování po 1 dni.

Platí, že čím jsou použita data s menšími intervaly, tím krátkodobější bude predikce. V této práci byla použita neagregovaná data, jelikož dataset obsahuje pro tato data nejvíce hodnot. Pokud by ovšem program umožnoval uchovávat i větší množinu agregovaných hodnot, dala by se předpověď provádět také na této množině dat.

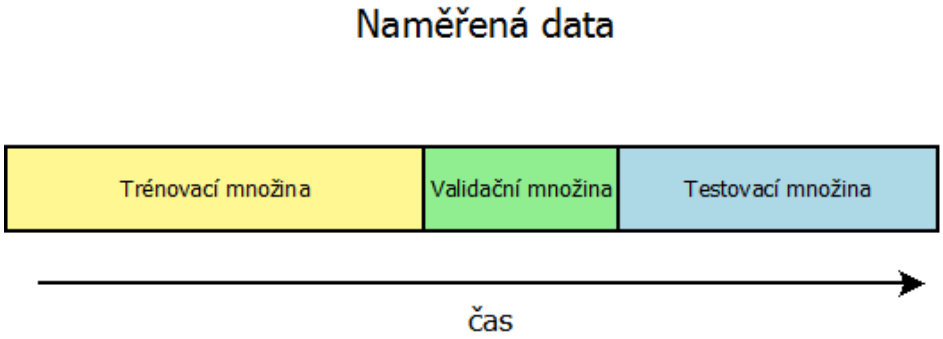
5.1.1 Rozdělení vstupních dat

Celý datový soubor získaný z databáze pro konkrétní zařízení a konkrétní časový rozptyl je rozdělen do 3 skupin:

- Trénovací množina - množina dat, která slouží pro učení sítě. Na základě těchto dat se nastavují váhy v síti a probíhá proces učení.
- Validační množina - množina dat, na které je vyhodnocována aktuální chyba sítě. Jakmile je tato chyba dostatečně malá, je proces trénování ukončen.
- Testovací množina - množina dat, na které je síť testována.

[1)] `https://www.solarwinds.com/network-performance-monitor`

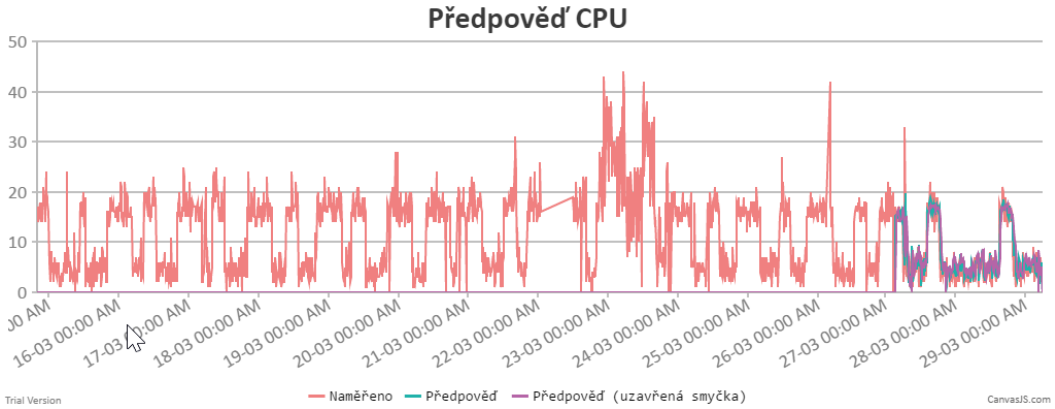
Jednotlivé množiny by měly být navzájem disjunktní. Grafické znázornění naleznete na Obr. 5.1. Ne vždy je nutné, aby množiny byly tvořeny souvislým blokem časové řady.



Obr. 5.1 Rozdělení dat na trénovací, validační a testovací množinu.

5.2 Periodicky se měnící data

V následující podkapitole bude ukázán výstup pro předpověď CPU na počítači, kde se data periodicky opakují. Vidíme, že počítač neustále přechází z pasivního režimu do aktivního a naopak. V tomto případě je pro neuronovou síť velice jednoduché se naučit závislosti mezi daty. Výstup je zobrazen na Obr. 5.2. Červenou křivkou jsou znázorněna naměřená data, zelenou předpovězená data a fialovou je vykreslena křivka předpovědi pomocí uzavřené smyčky.



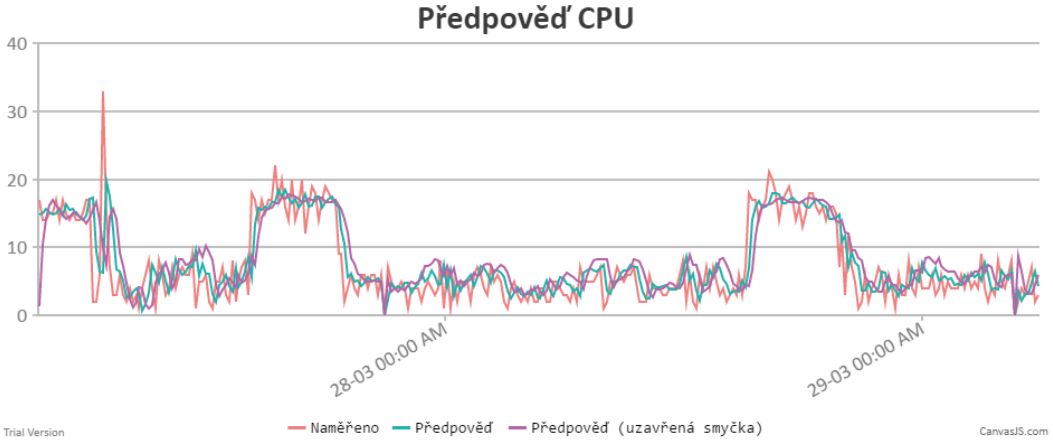
Obr. 5.2 Ukázka výstupu aplikace, pokud jsou vstupní data periodická.

Dalšími kritérii, která se budou posuzovat jsou trénovací čas a počet epoch nutných k naučení sítě. Tyto hodnoty jsou vypsány na konzolový výstup aplikace. Pro tento konkrétní případ jsou tyto hodnoty uvedeny v Tab. 5.1.

Aplikace umožňuje zobrazit pouze předpovězená data a tím zvýšit i detailnější informace o případných rozdílech mezi naměřenými a předpovězenými daty. Tento detailnější záběr dat je zobrazen na Obr. 5.3.

Tab. 5.1 Ukázka výstupních parametrů při učení sítě s konfigurací `30-10-1`.

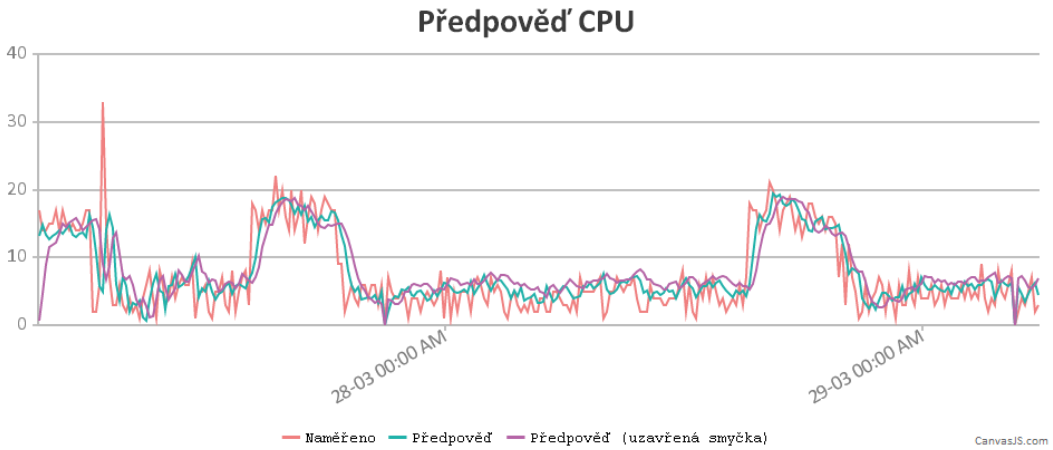
Velikost trénovací množiny	1665
Celkový čas trénování	0,182 s
Počet epoch pro naučení sítě	77



Obr. 5.3 Záběr na detail předpovědi periodicky se opakujících CPU dat při použití konfigurace sítě `30-10-1`.

5.2.1 Rozdíly mezi učícími algoritmy

Jak lze vidět na obrázcích 5.2 a 5.3, periodicky opakující se data lze snadno předpovědět. Pro učení byl použit algoritmus *Resilient Propagation* a doba učení byla zhruba 0,18 sekund. Chyba při trénování byla nastavena na hodnotu 0,01. Předpovězené hodnoty s určitou mírou kopírují data z testovací množiny, i když lze vidět, že neuronová síť předpovídá s určitou mírou zpoždění a neumí zachytit předpověď pro extrémní hodnoty.



Obr. 5.4 Výstup aplikace při učení pomocí metody backpropagation.

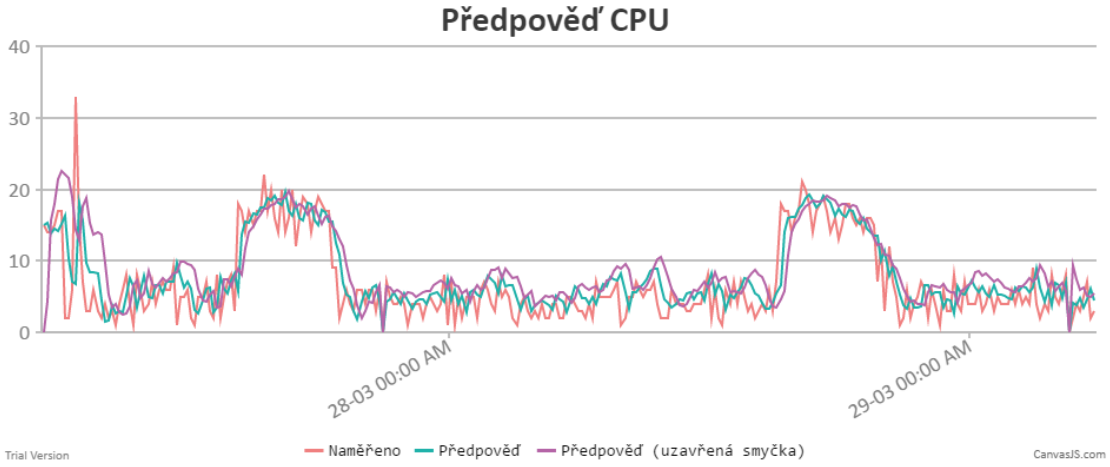
Na Obr. 5.4 je zobrazen výstup po naučení sítě pomocí algoritmu *Backpropagation*. Na první pohled můžeme lze vidět, že obě metody dávají zhruba stejné předpovědi. Hlavní rozdíl je zde v době učení a počtu epoch použitých pro naučení sítě, viz podkapitola 2.6. Jak lze vyčíst z Tab. 5.2, učení zabralo zhruba šestkrát delší dobu a počet epoch byl zhruba sedminásobný. Dále v této práci bude použit efektivnější algoritmus *Rprop*.

Tab. 5.2 Výstupní parametry při různém nastavení trénovacího algoritmu.

	Backpropagation	Resilient Propagation
Celkový čas trénování	1,245 s	0,182 s
Počet epoch pro naučení sítě	549	77

5.2.2 Vliv konfigurace sítě

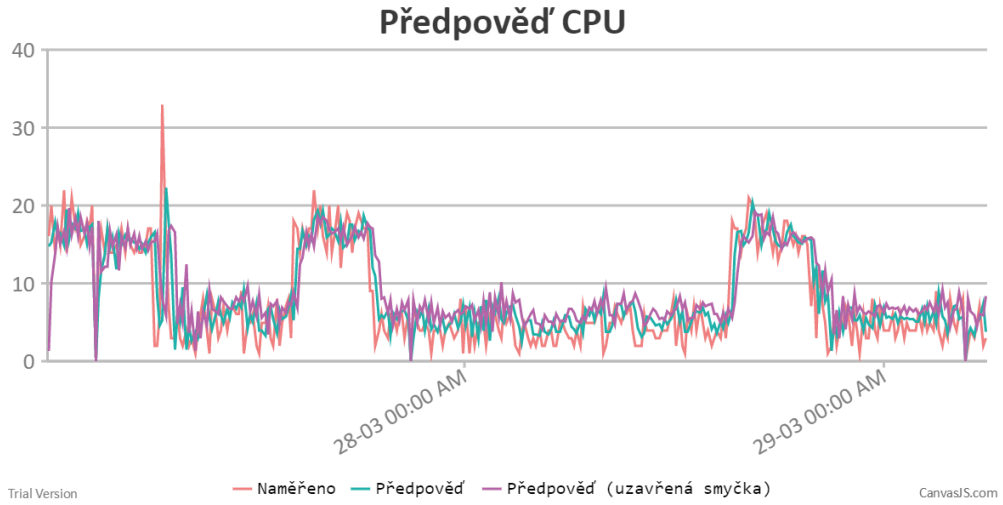
Dále bude znázorněn vliv nastavení konfigurace sítě. Na Obr. 5.5 je zobrazen průběh predikce pro konfiguraci sítě `40-50-30-1`. Lze pozorovat, že předpovězené hodnoty (i z uzavřené smyčky) opět kopírují tvar křivky testovací množiny. Stejně jako v předchozím případě, hlavní rozdíl je zde v době učení a počtu epoch nutných k naučení sítě.



Obr. 5.5 Výstup aplikace při konfiguraci sítě `40-50-30-1`.

Na druhou stranu, pokud použijeme síť pouze s 10 vstupními neurony (viz Obr. 5.6), tak lze pozorovat, že síť produkuje předpověď, která již není tak vyhlazená a má mezi jednotlivými hodnotami větší rozdíl.

Na závěr této podkapitoly je uvedena srovnávací tabulka 5.3, která ukazuje rozdíl mezi počtem epoch a trénovacím časem pro různé konfigurace sítě. Lze v ní také vypozorovat, že síť s menším počtem neuronů potřebovala více epoch pro naučení, protože k



Obr. 5.6 Výstup aplikace při konfiguraci sítě `10-1`.

zachycení závislostí měla méně prostředků. Nicméně jedna epocha díky menšímu počtu neuronů v síti zabrala méně času, a proto výsledný čas pro učení sítě byl menší.

Tab. 5.3 Výstupní parametry při různém nastavení konfigurace sítě.

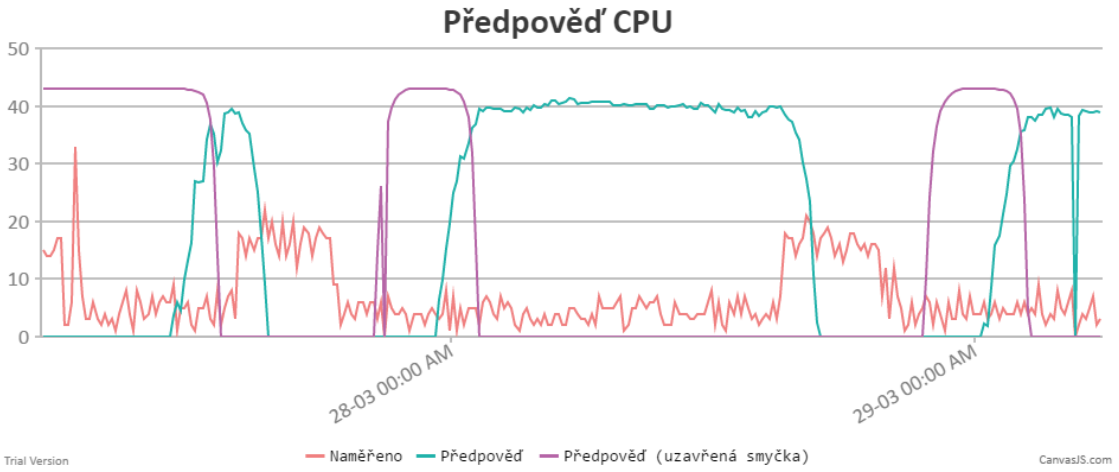
Konfigurace sítě	40-50-30-1	30-10-1	10-1
Celkový čas trénování	1,18 s	0,182 s	0,13 s
Počet epoch pro naučení sítě	59	77	147

5.2.3 Vliv nastavení chyby při učení

V podkapitole 2.5 bylo vysvětleno, jaký vliv bude mít nastavení chyby při učení sítě. Tato chyba se vyhodnocuje na validační množině a slouží pro ukončení trénování. Pro experiment použijeme předchozí datový soubor pracující nad CPU daty, kde zkusíme měnit chybu a budeme pozorovat, zdali se bude přesnost předpovědi měnit. Konfigurace sítě zůstala na hodnotě `40-50-30-1` a jako učící algoritmus byl zvolen `Rprop`.

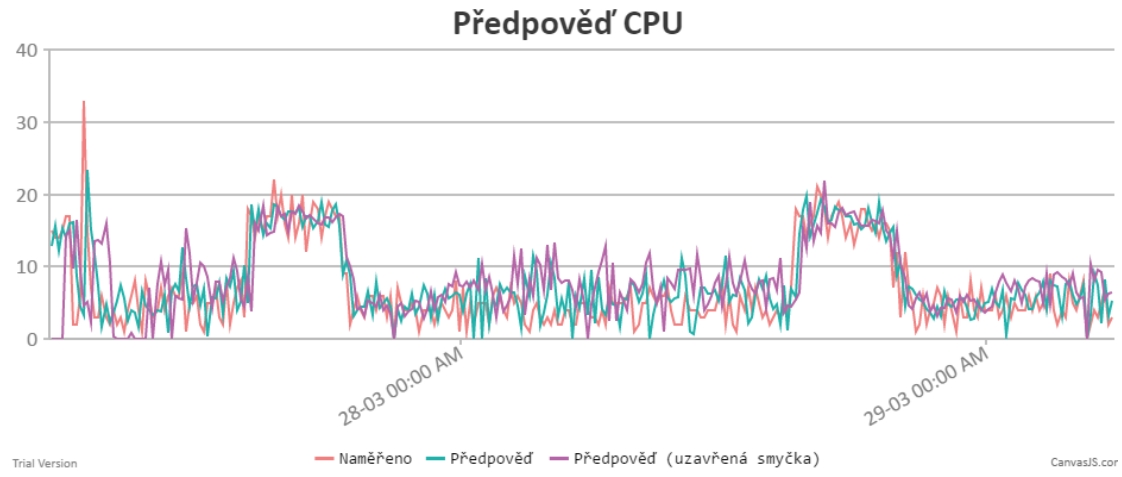
Pokud připustíme příliš vysokou chybu pro konec trénování, síť není schopná naučit se jakýkoliv detail. Výstup pro nastavenou chybu na hodnotu 0,1 je zobrazen na Obr. 5.7. Na obrázku lze vidět, že předpověď naprosto nekoresponduje s naměřenými daty a je proto nepoužitelná.

Pro chybu 0,01 je výstup zobrazen na Obr. 5.5. Lze vidět, že předpověď je mnohem přesnější, a přitom se síť učila pouze o vteřinu déle a potřebovala o 47 epoch více. Tento rozdíl mezi počty epoch není velký, nicméně v dalších kapitolách bude popsáno, jak by to vypadalo pokud bychom síti předkládali méně predikovatelná data (nebude se periodicky střídat aktivní a pasivní fáze využití systémového zdroje).



Obr. 5.7 Výstup aplikace při nastavení trénovací chyby na hodnotu 0,1.

Pokud budeme chybu postupně snižovat, uvidíme, že exponenciálně začne růst celkový čas trénování a počet epoch potřebných k natrénování sítě, viz Tab. 5.4. Nicméně na Obr. 5.8 a 5.9, kde byla požadovaná chyba snížena na hodnotu 0,005, resp. 0,001, lze vidět, že předpověď nebyla přesnější než s chybou 0,01 (viz. Obr 5.5).

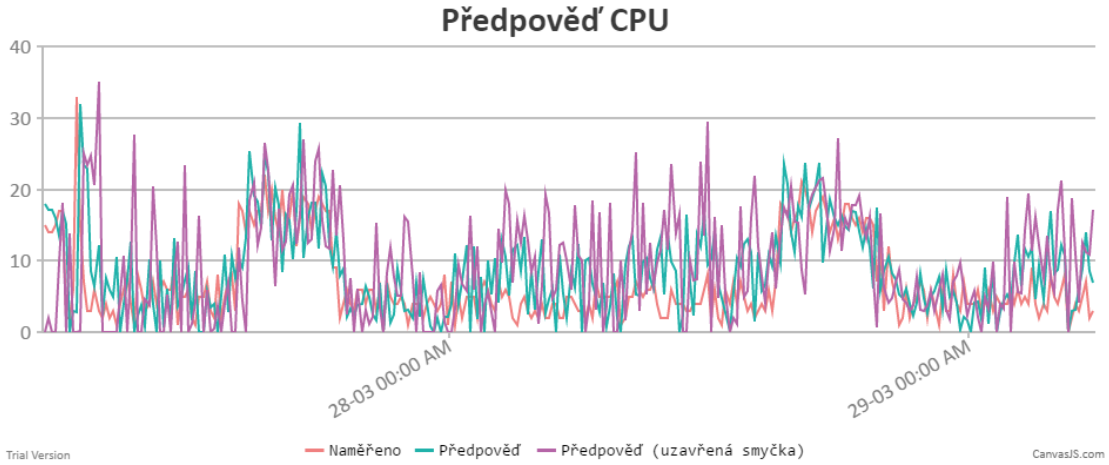


Obr. 5.8 Výstup aplikace při nastavení trénovací chyby na hodnotu 0,05.

Při snižování požadované chyby lze pozorovat, že dochází k přeučení sítě a nejenže dochází k delší době učení, ale také výsledky nejsou tak přesné jako v případě chyby 0,01.

V Tab.5.4 je znázorněno srovnání jednotlivých průběhů učení při nastavení výše popsaných chyb. Lze z ní vyčíst, že ne vždy se vyplatí mít pro validační množinu co nejmenší chybu. Optimální poměr mezi přesností predikce a délkou učení je v tomto případě pro nastavení chyby na hodnotu 0,01. Učení s chybou 0,001 trvalo nejdéle a nemá tak dobré výsledky jako učení s chybou 0,01.

Další problém při nastavení příliš nízké chyby nastává, pokud odebereme vnitřní



Obr. 5.9 Výstup aplikace při nastavení trénovací chyby na hodnotu 0,001.

Tab. 5.4 Výstupní parametry při různém nastavení trénovací chyby.

Nastavení chyby	0,1	0,01	0,005	0,001
Celkový čas trénování	0,161 s	1,18 s	3,892	50,253 s
Počet epoch pro naučení sítě	12	59	323	4390

vrstvu sítě a zůstane pouze vstupní okno. Pokud byla nastavena konfigurace sítě pouze na 30 vstupních neuronů, chybová funkce při učení uvázla v lokálním minimu na hodnotě 0,009 a učení bylo zastaveno.

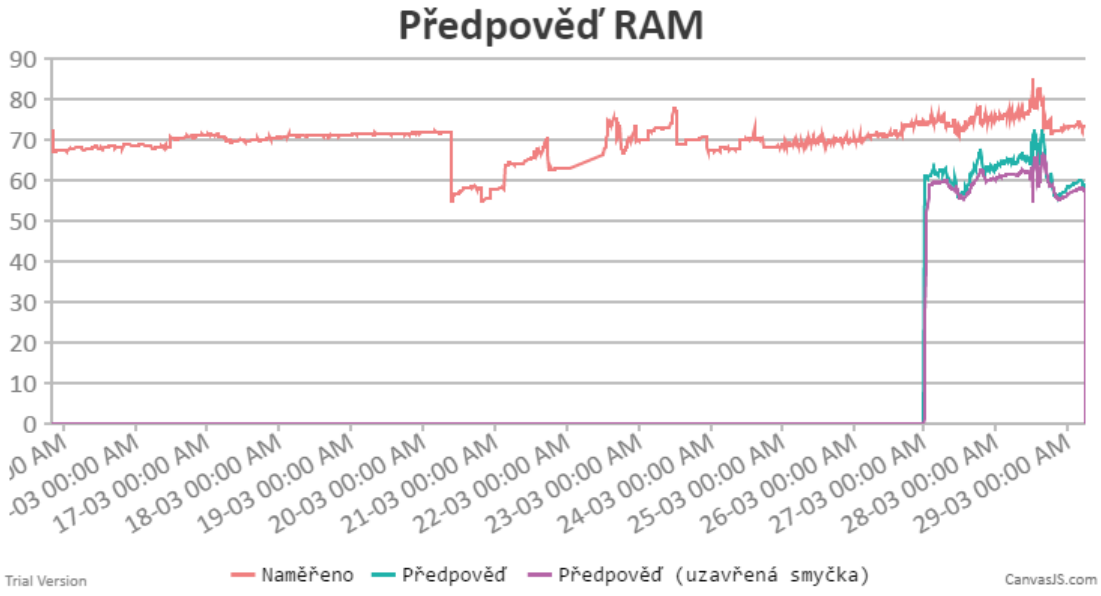
5.3 Téměř konstantní data

Téměř konstantní data byla pozorována na některých strojích hlavně u využití paměti RAM. Nejdříve si ukážeme na Obr. 5.10 průběh využití paměti RAM, tedy testovací množinu.

Neuronová síť měla s těmito daty větší problém než s periodicky se opakujícími daty. Síť sice byla schopna předpovědět tvar křivky, nicméně předpověď nebyla číselně přesná, jelikož výsledná predikce byla podhodnocena. Výsledek je zobrazen na Obr. 5.11.

Pokud snížíme chybu z 0,01 na 0,001, tvar křivky předpovědi zůstává podobný, nicméně jsme se přiblížili požadovaným hodnotám, viz Obr. 5.12.

V Tab. 5.5 lze vidět srovnání, kdy jsme se danou síť snažili učit se dvěma různými povolenými chybami na validační množině. V obou případech se sice povedlo odhadnout tvar, který kopíruje naměřené hodnoty využití paměti RAM, avšak obě předpovědi dávaly nižší čísla, než byla ve skutečnosti naměřená. Oproti předchozí kapitole 5.2, ale snížení povolené chyby pomohlo k tomu, aby se tento rozdíl zmenšil.



Obr. 5.10 Výstup aplikace při předpovědi nepříliš se měnících dat, zobrazena je i testovací množina.

Tab. 5.5 Výstupní parametry při různém nastavení trénovací chyby pro předpověď RAM.

Nastavení chyby	0,001	0,01
Celkový čas trénování	0,212 s	0,087 s
Počet epoch pro naučení sítě	65	27

Za výsledkem dosažení lepší předpovědi bylo experimentováno i s konfigurací sítě, nicméně nejlepších výsledků bylo na těchto datech dosaženo při vynechání skryté vrstvy a nastavení okna na hodnotu 4.

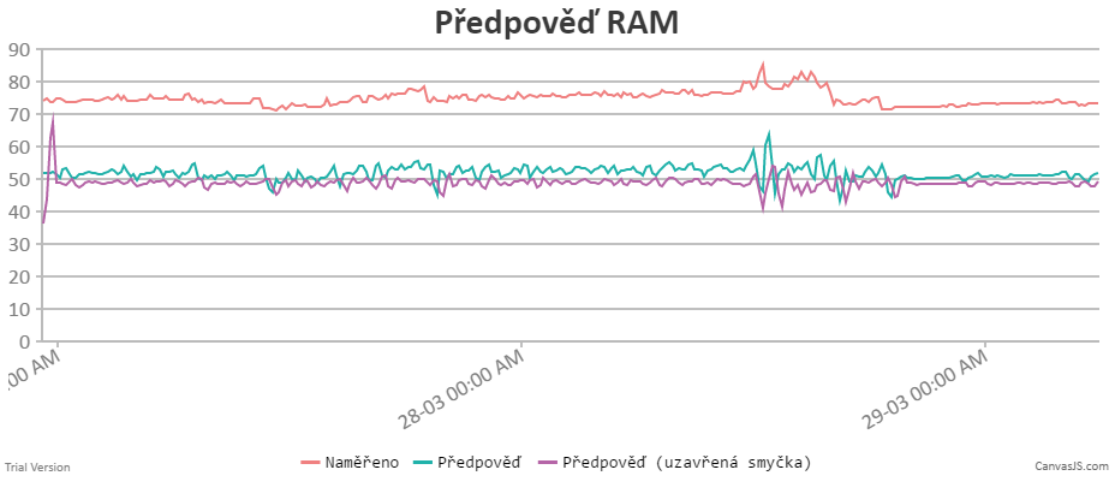
5.4 Stochastická data

V následující podkapitole bude předvedeno, jak dokáže aplikace fungovat nad stochastickými daty. Stochastická data jsou data, která se dají považovat za náhodná. V kontextu této práce tedy budeme pracovat nad naměřenými hodnotami využití CPU a RAM, která se periodicky neopakují a ani nedrží přibližně konstantní hladinu.

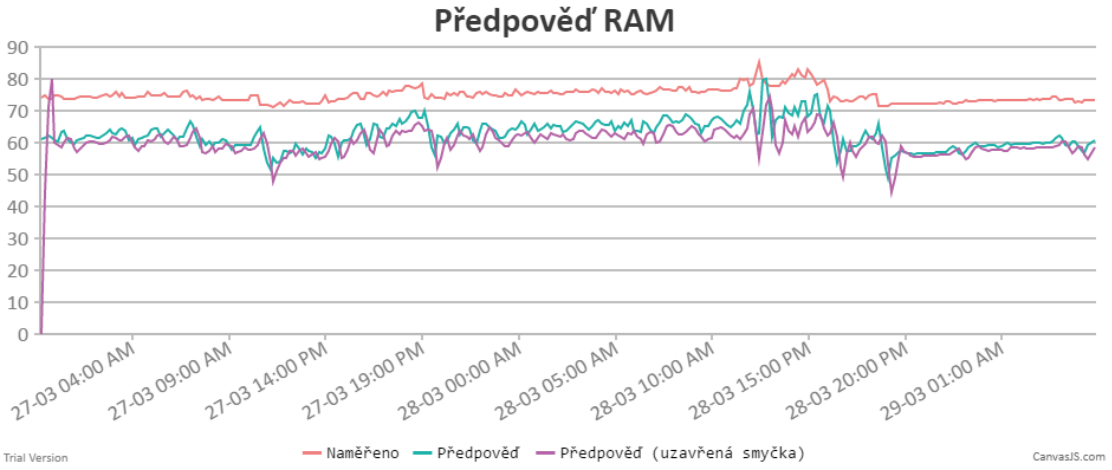
5.4.1 CPU

Nejdříve budou představena stochastická CPU data. Na Obr. 5.13 jsou zobrazena data, která byla zkoumána. Na první pohled data vypadají náhodně, nicméně neuronová síť si s nimi dokáže poradit.

Problémem v tomto případě je, že síť neumí předpovídat extrémní hodnoty, což



Obr. 5.11 Detailnější pohled na předpověď nepříliš se měnících dat.

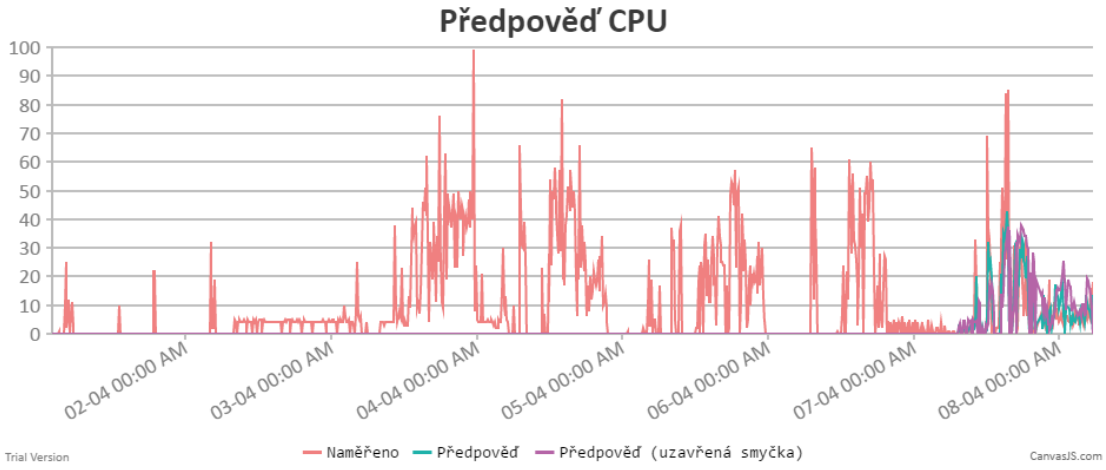


Obr. 5.12 Vývoj předpovědi, pokud zmenšíme přípustnou chybu při trénování na 0,001.

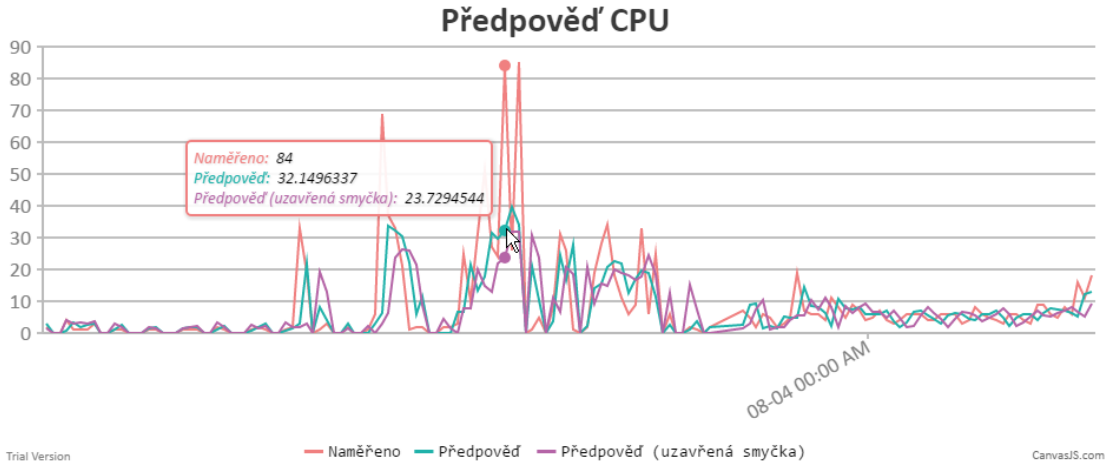
lze dobře vidět na detailnějším záběru na Obr. 5.14. Pro síť byla zvolena konfigurace `20-10-1`.

Dále bude ukázáno, zdali síť dokáže zachytit detailněji vztahy pomocí zvětšení posunujícího se okna a přidáním další skryté vrstvy neuronů. Na Obr. 5.15 je zobrazen predikce pomocí sítě, která měla konfiguraci `30-50-20-1`. Jak lze pozorovat, výsledek je téměř totožný. Rozdíl je ovšem v počtu epoch potřebných pro naučení sítě, kde síť s větším vstupním oknem a přidanou skrytou vrstvou potřebovala zhruba 2,5x méně epoch.

Pokud jsou skryté vrstvy odebrány, uvázne učící algoritmus při hledání nejmenší chyby v lokálním minimu a není schopen najít nejlepší řešení. V Tab. 5.6 jsou zobrazeny výstupní parametry aplikace pro různé konfigurace sítě.



Obr. 5.13 Vývoj předpovědi, pokud na vstupu do sítě jsou stochastická data.



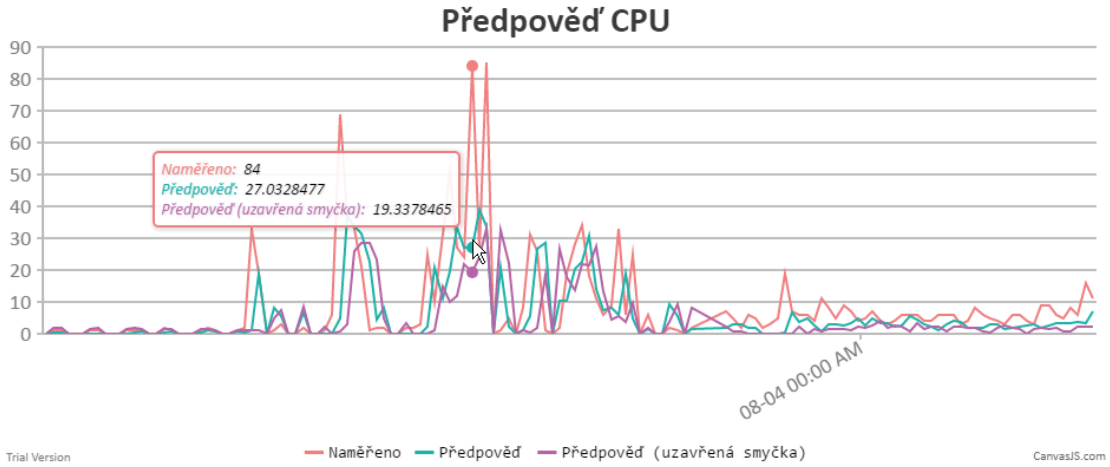
Obr. 5.14 Detailní pohled na předpověď, pokud na vstupu do sítě jsou stochastická data.

5.4.2 RAM

Obdobné experimenty, které byly předvedeny v předchozí podkapitole 5.4.1 nad naměřenými CPU daty budou popsány v této kapitole nad naměřenými hodnotami využití paměti RAM. Naměřené hodnoty jsou zobrazeny na Obr. 5.16

Stejně jako v předchozí podkapitole, nastavíme velikost vstupního okna na 20 datových vzorků a přidáme skrytou vrstvu o velikosti 10 neuronů. Výstup je vidět na Obr. 5.17. Pokud se na predikci podíváme z pohledu celé datové množiny (Obr. 5.16), vidíme, že předpověď s jistou mírou chyby kopíruje data z testovací množiny. Stejně jako u konstantních dat, síť nebyla schopna přesněji předpovědět hodnoty, kde byla data z testovací množiny v jistém intervalu konstantní.

Hlavní rozdíl oproti předpovědi CPU z minulé podkapitoly je rychlost předpovědi.



Obr. 5.15 Předpověď stochastických dat, pokud je konfigurace sítě `30-50-20-1`.

Tab. 5.6 Výstupní parametry při různém nastaveníkonfigurace sítě u stochatických CPU dat.

Konfigurace sítě	20-10-1	30-50-20-1
Celkový čas trénování	0,103 s	0,29 s
Počet epoch pro naučení sítě	101	41

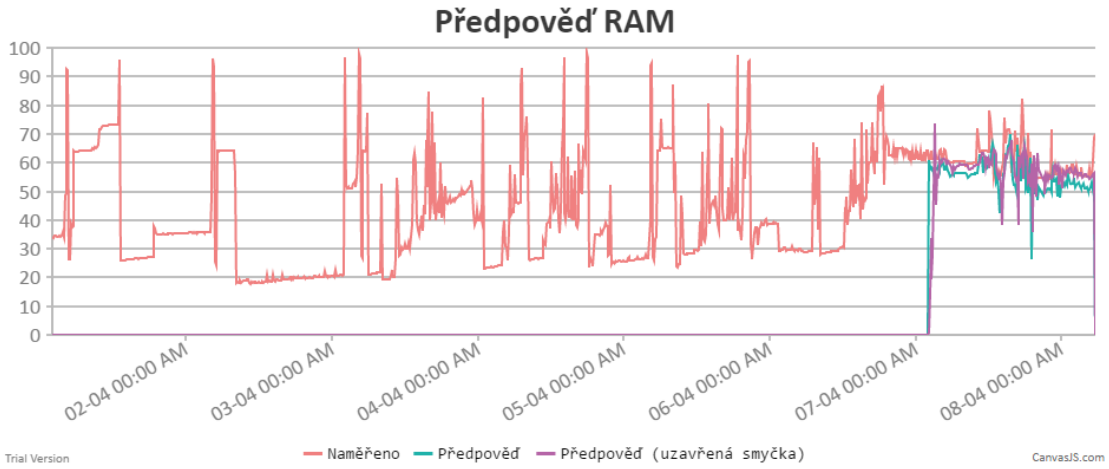
Jak lze vidět v tabulce 5.7, předpověď trvala přes 6 sekund a počet epoch potřebných k naučení sítě byl 7383. Přitom síť operovala nad stejným počtem dat v testovací množině, jako CPU v předchozí podkapitole, tedy 854. Pokud odebereme skrytou vrstvu, hledání nejmenší chyby uvázne v lokálním minimu.

Naopak, pokud přidáme další skrytou vrstvu a budeme používat síť s konfigurací `30-50-20-1`, pozorujeme nejen přesnější předpověď, ale také se síť dokáže učit rychleji a s menším počtem epoch, viz Obr. 5.18.

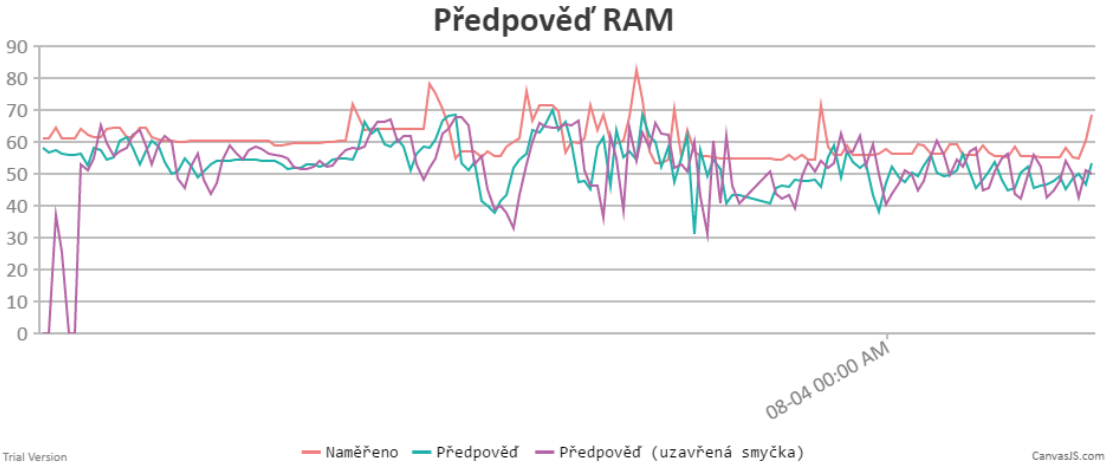
Z Tab. 5.7 lze vyčíst, že přidání další skryté vrstvy výrazně pomohlo ke snížení epoch pro naučení sítě, a také ke snížení potřebného času pro naučení sítě.

Tab. 5.7 Výstupní parametry při různém nastavení konfigurace sítě u stochatických RAM dat.

Konfigurace sítě	20-10-1	30-50-20-1
Celkový čas trénování	6,318 s	1,161 s
Počet epoch pro naučení sítě	7383	187



Obr. 5.16 Pohled na datový vzorek naměřených hodnot využití RAM.



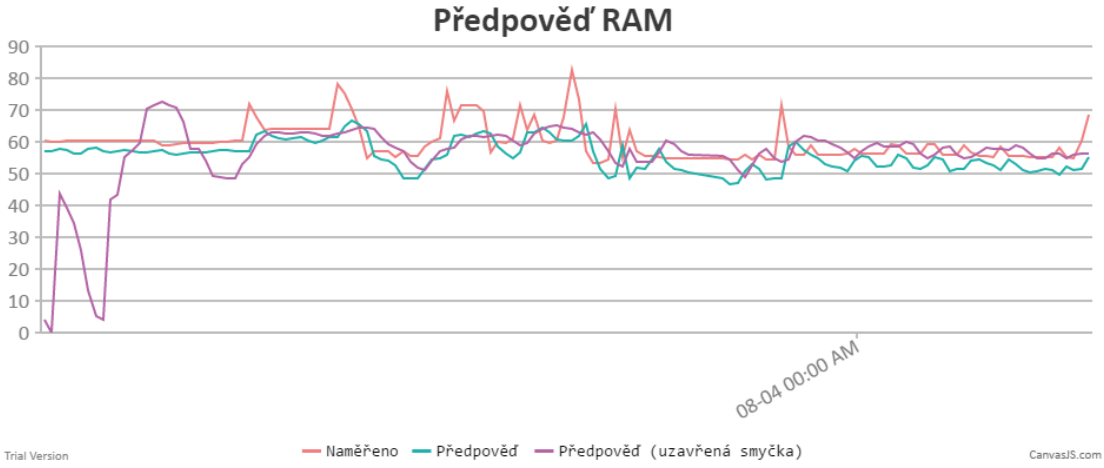
Obr. 5.17 Předpověď RAM, pokud je konfigurace sítě nastavena na `20-10-1`.

5.4.3 Předpověď CPU na základě využití RAM a CPU

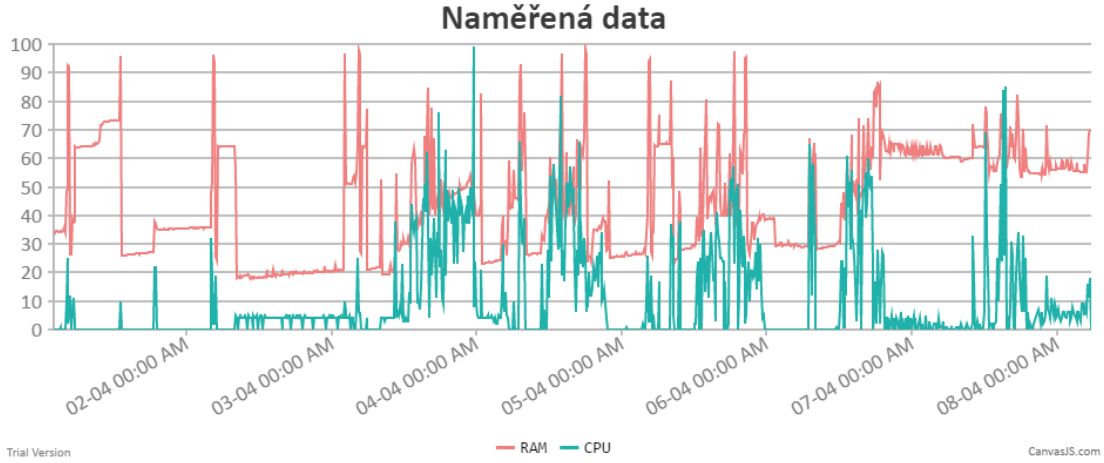
Jak již bylo popsáno v kapitole 3.1, jednou z možností analýzy časových řad je mapování časové řady na jinou časovou řadu. V této kapitole bude ukázáno mapování v praxi. Budeme se snažit na základě naměřených hodnot využití CPU a RAM předpovědět budoucí hodnoty CPU. Jinými slovy se budeme snažit zjistit, zdali nám přidání další řady zpřesní předpověď pomocí nalezení nových vztahů mezi daty v trénovací množině. Posunující se okno tedy obsahuje několik naměřených hodnot pro CPU i pro RAM.

Bude využit stejný datový soubor jako v předchozí kapitole. Naměřená data lze vidět na Obr. 5.19, kde červenou křivkou je zobrazeno využití RAM a zelenou využití CPU.

Na Obr. 5.20 je zobrazen pokus o předpověď na základě více časových řad. Chyba při trénování byla nastavena na hodnotu 0,01, konfigurace sítě byla `30-50-20-1`. Jak lze vidět v porovnání s Obr. 5.15 (kde předpověď probíhala pouze na základě předchozích



Obr. 5.18 Předpověď RAM, pokud je konfigurace sítě nastavena na `30-50-20-1`.



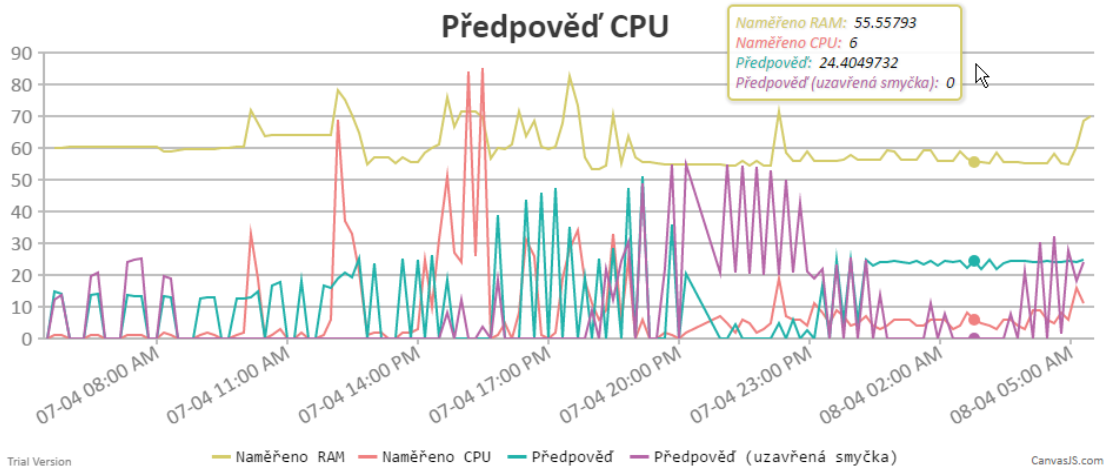
Obr. 5.19 Naměřené hodnoty CPU a RAM.

hodnot), předpověď je spíše zkreslená než přesnější. Neuronová síť tedy nedokázala najít spojitost mezi naměřenými hodnotami využití CPU a RAM.

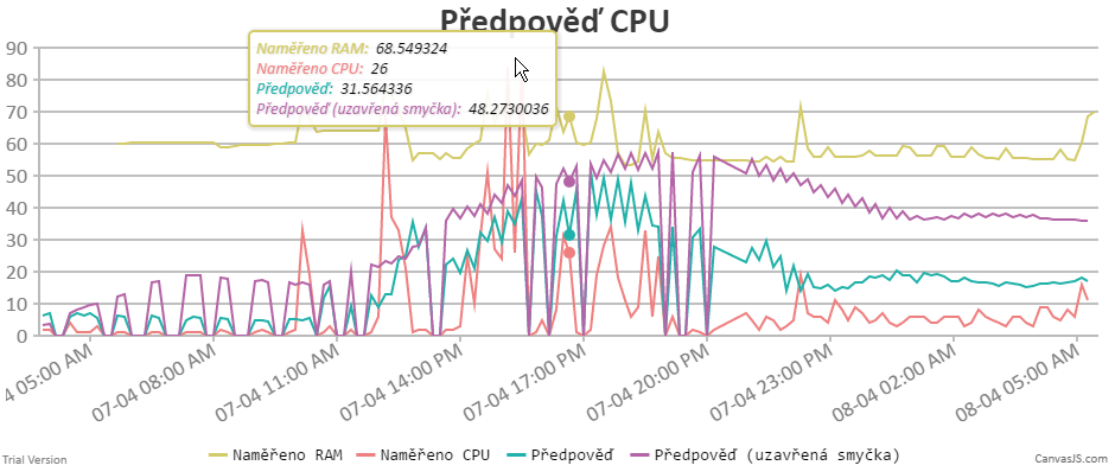
Pokud nastavíme konfiguraci sítě na `20-10-1`, tedy zmenšíme okno, odebereme jednu skrytou vrstvu a snížíme povolenou chybu při trénování na 0,001, předpověď se mírně zpřesní, viz Obr. 5.21. Nicméně v porovnání s původní předpovědí pouze na základě jedné řady (Obr. 5.14), je tato předpověď opět horší.

Pokud se zbavíme skrytých vrstev, zmenšíme vstupní okno na velikost 10 a povolenou chybu ponecháme na hodnotě 0,001, dostaneme nejlepší předpověď z příkladů uvedených v této podkapitole, viz Obr. 5.22. Tato předpověď je ale stále horší, než předpověď pouze na základě historických hodnot.

V Tab. 5.8 lze vidět, že trénovací časy a počty epoch při trénování sítí se skrytými vrstvami byly téměř totožné. Ovšem nejlepší trénovací čas (i když za pomoci nejvíce epoch) měla síť s konfigurací `10-1`.



Obr. 5.20 Předpověď CPU na základě naměřených hodnot CPU a RAM při konfiguraci sítě `30-50-20-1`.

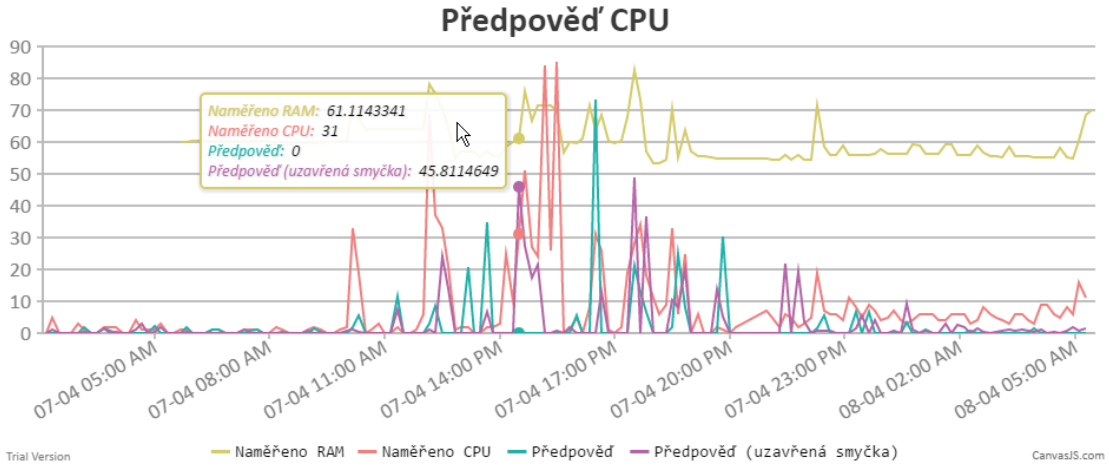


Obr. 5.21 Předpověď CPU na základě naměřených hodnot CPU a RAM při konfiguraci sítě `20-10-1`.

5.4.4 Předpověď RAM na základě využití RAM a CPU

V této podkapitole se pokusíme o předpověď budoucích hodnot RAM za využití naměřených hodnot CPU a RAM, jedná se tedy o podobný postup jako v předchozí podkapitole, ale na vstup budeme dávat naměřené hodnoty RAM. Při konfiguraci sítě `30-20-10-1` lze výsledek vidět na Obr. 5.23. Lze vidět, že stejně jako při předpovědi RAM pouze na základě naměřených hodnot této řady, je předpověď podhodnocená.

Pokud ale zvýšíme počet neuronů ve skryté vrstvě, předpověď se stane přesnější, viz Obr. 5.24. Průběh výsledné křivky předpovědi je podobný jako testovací množina, ale předpověď je stále podhodnocená. Předpověď není tak přesná, jako na Obr. 5.17 a 5.18, kde byla prováděna pouze na základě naměřených hodnot jedné řady.



Obr. 5.22 Předpověď CPU na základě naměřených hodnot CPU a RAM při konfiguraci sítě `10-1`.

Tab. 5.8 Výstupní parametry při předpovědi CPU pomocí naměřených CPU a RAM dat.

Konfigurace sítě	30-50-20-1	20-10-1	10-1
Nastavená chyba pro trénování	0,01	0,001	0,001
Celkový čas trénování	0,113 s	0,14 s	0,04 s
Počet epoch pro naučení sítě	12	11	23

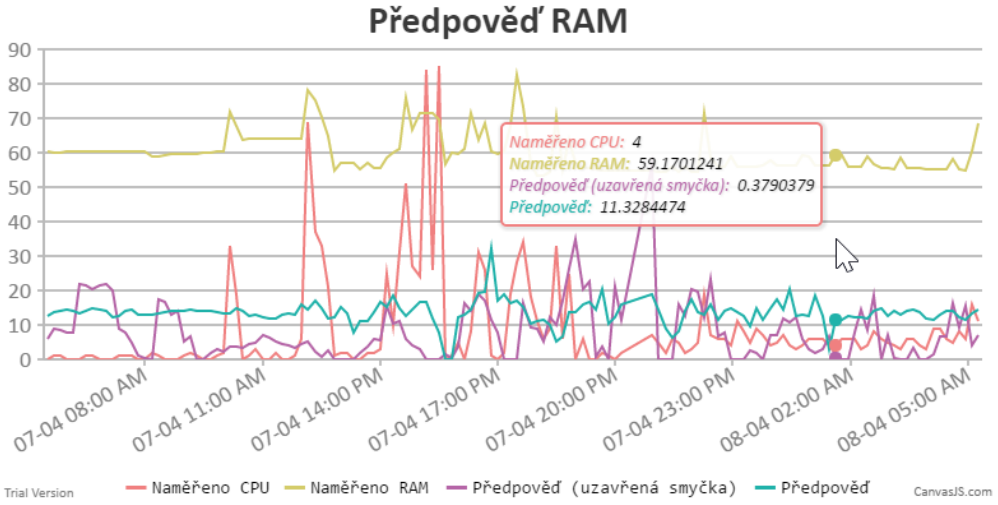
5.5 Shrnutí

V předchozí kapitole byl v praxi ukázán výstup aplikace na několika vzorcích dat pro CPU i RAM. Nejlepší výsledky byly dosaženy, pokud síť byla schopna nalézt jistý vzor, tedy pokud se vstupní data s jistou mírou šumu pravidelně opakovala. Byl ukázán rozdíl, pokud se síť učí pomocí metody *backpropagation* a její modifikace *resilient propagation*, která se jeví jako o mnoho rychlejší. Dále byl ukázán vliv nastavení chyby pro ukončení trénování, kdy je potřeba najít kompromis mezi potřebnou přesností a zároveň rozumnou dobou učení.

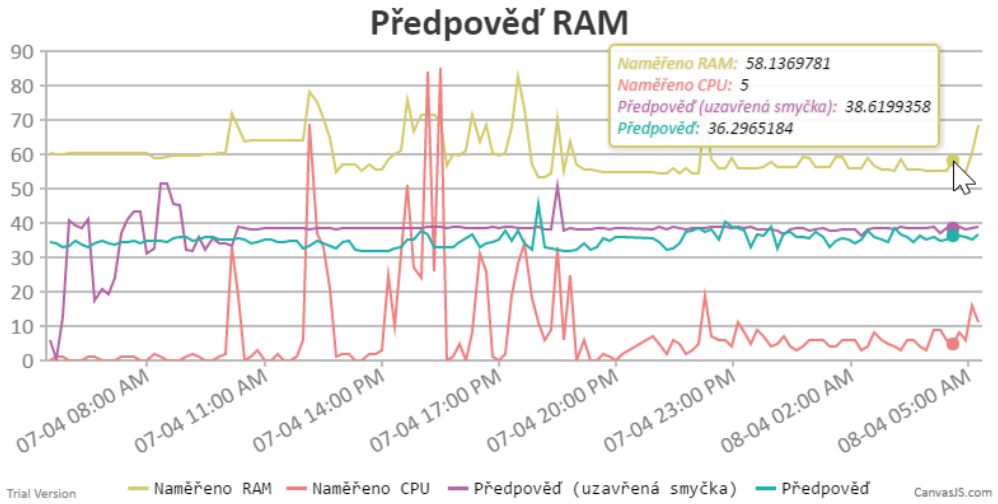
Síť si nejhůře vedla s daty, která se příliš nemění. Nebyla schopna najít vzor, podle které by dokázala spolehlivě předpovídat budoucí hodnoty. Se stochastickými daty byla již předpověď lepší, nicméně síť ve většině případů nebyla schopna předpovědět extrémní hodnoty, což platilo i pro konstatní data i pro periodicky se opakující data.

V poslední části byla navrhnuta předpověď zdrojů na základě hodnot z více řad (RAM a CPU), do vstupního okna tedy vstupovaly dvě veličiny. Nicméně se ukázalo, že předpověď pouze na základě historických hodnot jedné řady byla přesnější.

Dále byla vytvořena tzv. předpověď na základě uzavřené smyčky, kdy se síť snažila



Obr. 5.23 Předpověď RAM na základě naměřených hodnot CPU a RAM při konfiguraci sítě `30-20-10-1`.



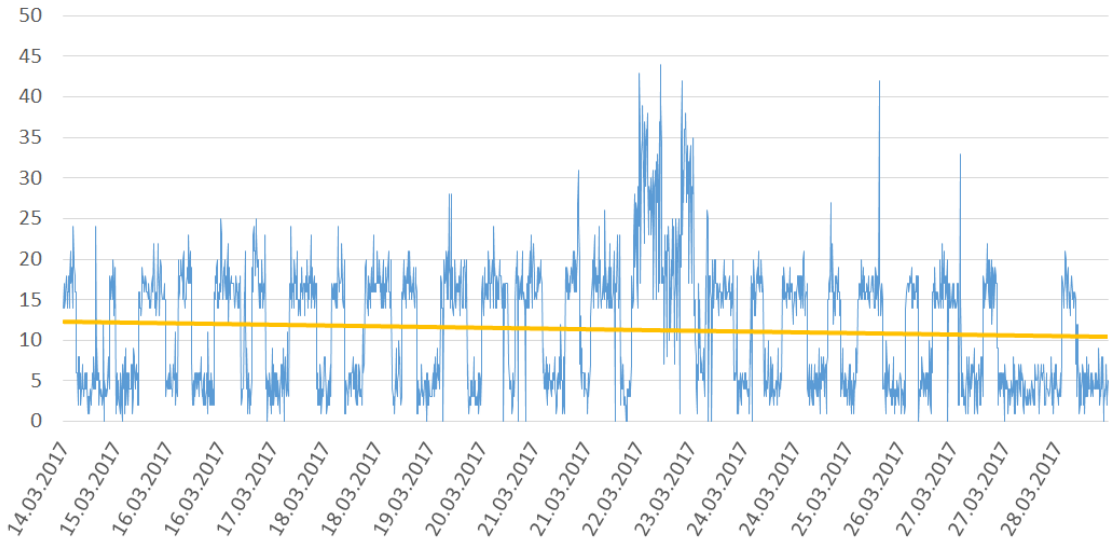
Obr. 5.24 Předpověď RAM na základě naměřených hodnot CPU a RAM při konfiguraci sítě `30-50-20-1`.

předpovídat již na základě předpovězených hodnot, tedy posunující se okno bylo plněno již předpovězenými hodnotami.

6 Srovnání

V předchozí kapitole byl výstup sítě (předpovězené hodnoty) srovnáván s daty v testovací množině. V následující kapitole bude porovnán přístup předpovědi pomocí lineární regrese (popsané v kapitole 1.3) a pomocí neuronových sítí. Pro srovnání byly využity stejné datasety jako pro experimenty s aplikací, popsané v předchozí kapitole.

Nejdříve budou srovnána periodicky se opakující CPU data z Obr. 5.2. Lineární regrese pro tento model je zobrazena na Obr. 6.1. Modrou křivkou jsou znázorněna naměřená data, žlutou je zobrazena lineární regrese. Jak lze vidět, lineární regrese nedokáže zachytit jednotlivé vztahy mezi naměřenými hodnotami a předpověď je téměř nepoužitelná.

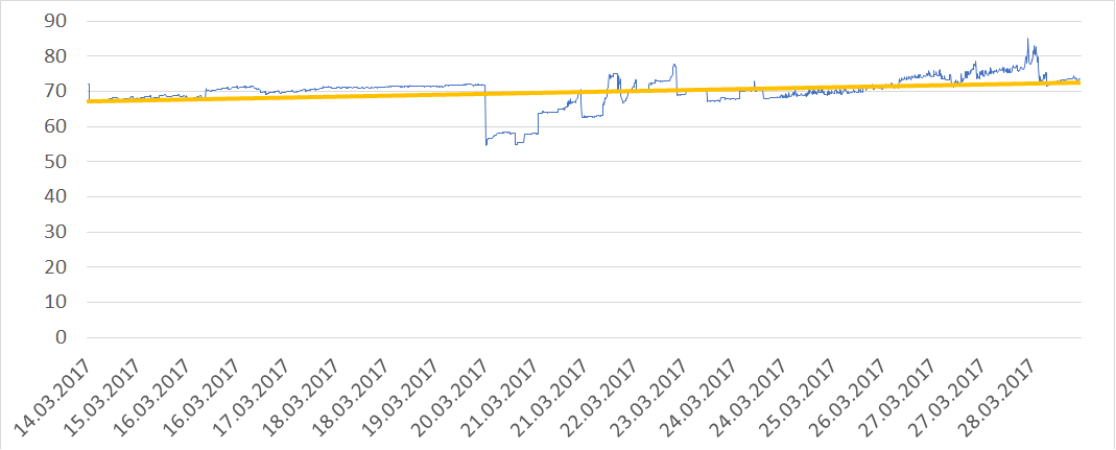


Obr. 6.1 Metoda lineární regrese aplikovaná na periodická data.

Pokud provedeme lineární regresi nad nepříliš se měnícími daty (v tomto případě využití RAM, viz Obr. 5.10), dostaneme výsledek zobrazený na Obr. 6.2. V porovnání s předpovědí pomocí neuronové sítě je na první pohled vidět, že proložená přímka více sedí do křivky naměřených dat. To je způsobeno tím, že data mají lineárnější charakter, než v předchozím případě.

Jedním z nástrojů, které používají lineární regresi je produkt *Network Performance Monitor*. Jak již bylo popsáno výše, tento program dokáže monitorovat zařízení v síti a mimo jiné měřit využití CPU a RAM. V kapitole 5 bylo pro predikci využito dat, která byla měřena v 10-minutových intervalech, *Network Performance Monitor*, nicméně tato data agreguje a pro predikci využívá data, která mezi sebou mají interval jeden den.

Předpověď využití CPU je zobrazena na Obr. 6.3. Lze pozorovat, že předpovídány jsou dvě hodnoty: *Average trend* a *Peak trend*. *Average trend* využívá průměrné hodnoty za den, zatímco *Peak trend* používá pro předpověď vždy nejhorší možnou variantu



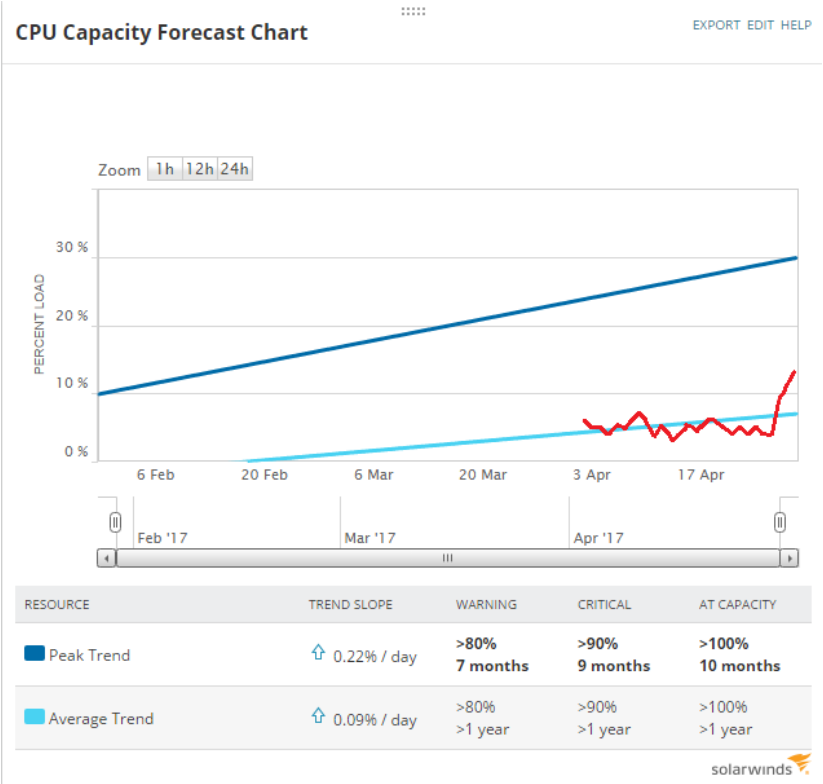
Obr. 6.2 Metoda lineární regrese aplikovaná na téměř konstantní data.

za celý den. *Peak trend* je tedy vhodný pro kritická zařízení, u kterých si nemůžeme dovolit překročit jistý práh hodnoty využití. Pod grafem jsou totiž zobrazeny sloupce které ukazují, za jak dlouho budou překročeny jisté prahové hodnoty: 80%, 90%, 100%. Na základě těchto hodnot se pak dají vyvolat další akce, které by mohly pomoct předejít případnému přetížení zařízení.

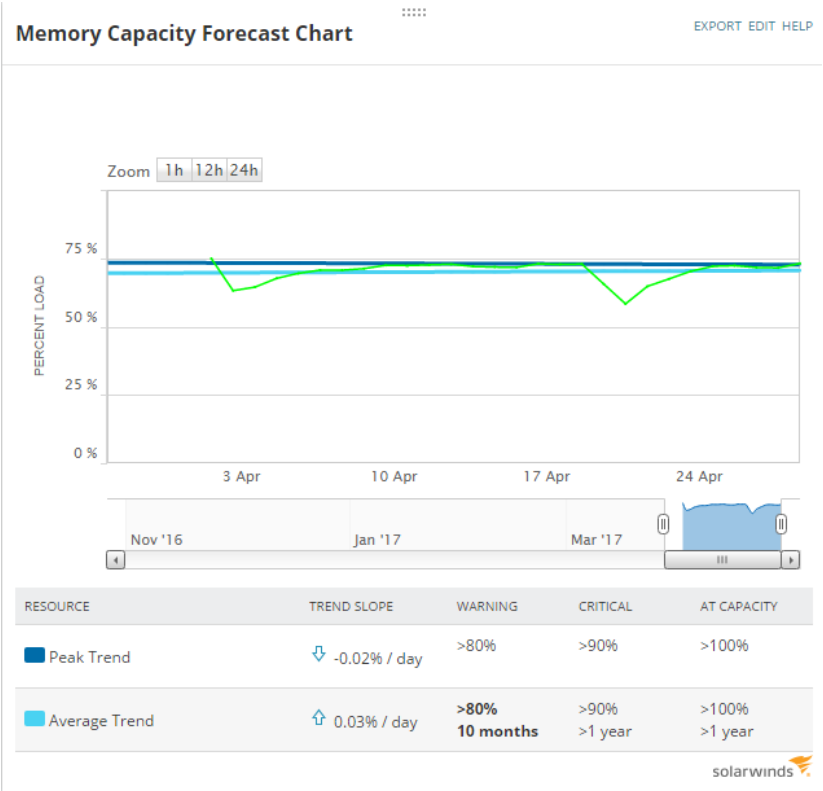
Obdobně je zobrazena i předpověď pro využití RAM, viz Obr. 6.4. Ve sloupci *Trend slope* pod grafem lze pozorovat, jak se přímka lineární regrese za jeden den změnila, což může být také nápomocné.

Obecně lze tedy říct, že pro předpověď za pomoci neuronových sítí je potřeba více dat než pro lineární regresi, aby síť byla schopna rozpoznat jednotlivé vztahy mezi daty. Pokud budeme mít lineární data, stačí nám podstatně méně dat, abychom byli schopni určit trend těchto dat. Naopak, pokud nebudou data lineární, tak nám nepomůže ani větší vzorek dat pro správnou předpověď.

Network Performance Monitor používá pro výpočet lineární regrese agregovaná data, která představují jeden den. V této práci bylo využito neagregovaných dat s desetiminutovými intervaly. To má za následek krátkodobější předpověď, nicméně pokud bychom zkoušeli předpovídat na základě denních dat, trvalo by nám několik let získat relevantní množinu dat.



Obr. 6.3 Předpověď využití CPU v programu Network Performance Monitor.



Obr. 6.4 Předpověď využití RAM v programu Network Performance Monitor.

ZÁVĚR

Hlavním cílem této práce bylo seznámení s neuronovými sítěmi, metodami pro předpověď pomocí neuronových sítí a hlavně převedení těchto poznatků do praxe a vytvořit model pro předpověď budoucích hodnot CPU a RAM na základě naměřených hodnot. Tato předpověď může najít uplatnění u správců zařízení v síti, kde by pro ně mohlo být užitečné efektivně řídit práci se systémovými zdroji.

Výstupem této práce je konzolová aplikace implementovaná v jazyce C#, která na vstupu očekává přístup k datům (buď přímo k databázi nebo formát CSV) a z těchto dat umí předpovědět budoucí hodnoty. Data byla získána pomocí nástroje *Network Performance Monitor*.

Vstupní data by se dala kategorizovat do třech skupin:

- periodicky se opakující data,
- nepříliš se měnící data,
- stochastická data.

Nejlepší výsledky byly dosaženy při periodicky se měnících datech, naopak nejhůř dopadla téměř konstantní data.

Dalším přístupem k předpovědi mohou být statistické metody, jednou z nich je lineární regrese. Tato metoda pracuje na principu proložení naměřených hodnot přímkou, která představuje budoucí vývoj hodnot. Ve srovnání s touto metodou, neuronové sítě byly přesnější pro periodicky se opakující data i pro stochastická data, nicméně pro téměř konstantní data dává lepší výsledky lineární regrese.

Z výše popsaných poznatků vyplývá, že každá metoda je vhodná pro různé typy datových vzorků. Vhodným rozšířením implementované aplikace by mohlo být rozšíření o práci s lineární regresí. Za účelem co nejpřesnější předpovědi by tedy předpověď mohla probíhat více způsoby. Jednoduchým předzpracováním dat by program mohl zjistit, jakého charakteru data jsou. Na základě této informace poté vybrat vhodnou metodu pro předpověď.

Dalším z rozšíření by mohl být generování emailových zpráv, pokud předpovězené hodnoty překročí daný práh. Tento práh by byl nastaven uživatelem. Aplikace by poté byla spouštěna periodicky a pokud by předpověď překročila danou prahovou hodnotu, mohla by vygenerovat emailovou zprávu pro správce zařízení. Ten by poté byl schopen na danou situaci zareagovat s předstihem. Správce nemusí reagovat pouze na překročení maximálních hodnot. Za účelem snížení nákladů na elektrickou energii také může sledovat, která zařízení budou minimálně využívaná a tato zařízení přepnout do úsporného režimu nebo vypnout.

Neuronové sítě tedy mohou být využity pro předpověď systémových zdrojů, ale existují datové vzorky, na kterých síť nedává tak přesnou předpověď jako lineární regrese. Proto můžeme tyto dvě metody zkombinovat, čímž bychom dostali model, který by si uměl poradit se vzorky dat různých charakterů. I když lze velkou část tohoto procesu předpovědi zautomatizovat, od získání dat až po zobrazení výsledků, stále bude potřeba osoby, která tyto předpovědi bude vyhodnocovat. Na základě svých zkušeností a výstupu z aplikace může efektivně tyto zdroje řídit.

SEZNAM POUŽITÉ LITERATURY

[1] *Úvod do regresní analýzy* [online]. [cit. 2017-04-22]. Dostupné z: http://www.statsoft.cz/file1/PDF/newsletter/2014_26_03_StatSoft_Uvod_do_regresni_analyzy.pdf

[2] *Moore's law really is dead this time* [online]. [cit. 2017-04-08]. Dostupné z: https://arstechnica.com/information-technology/2016/02/moores-law-really-is-dead-this-time

[3] *Types of Slots on a Motherboard* [online]. [cit. 2017-04-17]. Dostupné z: http://smallbusiness.chron.com/types-slots-motherboard-56027.html

[4] FARAHNAKIAN, Fahimeh, Pasi LILJEBERG a Juha PLOSILA. LiRCUP: Linear Regression Based CPU Usage Prediction Algorithm for Live Migration of Virtual Machines in Data Centers. *2013 39th Euromicro Conference on Software Engineering and Advanced Applications* [online]. IEEE, 2013, 357-364 [cit. 2017-04-14]. ISBN 978-0-7695-5091-6. Dostupné z: http://ieeexplore.ieee.org/document/6619533/

[5] *Hard drive occupation prediction with R* [online]. [cit. 2017-04-14]. Dostupné z: http://www.lpenz.org/articles/df0pred-1/index.html

[6] ŠÍMA, Jiří a Roman NERUDA. *Teoretické otázky neuronových sítí.* Praha: MATFYZPRESS, 1996. ISBN 80-85863-18-9.

[7] VOLNÁ, Eva.*Neuronové sítě 1* [online]. [cit. 2016-12-16]. Vyd. 2, Ostrava: Ostravská univerzita v Ostravě, 2008. Dostupné z: http://www1.osu.cz/~volna/Neuronove_site_skripta.pdf

[8] HEATON, Jeff. *Programming Neural Networks with Encog 3 in C#*. Heaton Research, 2011. ISBN 978-1-60439-027-8.

[9] ROJAS, Raúl. *Neural networks: a systematic introduction* New York: Springer-Verlag, 1996. ISBN 3-540-60505-3.

[10] DORFFNER, Georg. *Neural Networks for Time Series Processing.* Dept of Medical Cybernetics and Artificial Intelligence University of Vienna and Austrian Research Institute for Artificial Intelligence [cit. 2017-04-08]. Dostupné z: http://machine-learning.martinsewell.com/ann/Dorf96.pdf

[11] KŘIVAN, Miloš. *Úvod do umělých neuronových sítí.* Vyd. 3., přeprac. Praha: Oeconomica, 2014. ISBN 978-80-245-2024-7.

[12] PRIDDY, Kevin L. a Paul E. KELLER. *Artificial neural networks: an introduction*. Bellingham, Wash.: SPIE Press, 2005. ISBN 08-194-5987-9.

[13] *How To Use Resilient Back Propagation To Train Neural Networks* [online]. [cit. 2017-04-14]. Dostupné z: `https://visualstudiomagazine.com/articles/2015/03/01/resilient-back-propagation.aspx`

[14] *Prediction using neural networks* [online]. [cit. 2017-04-14]. Dostupné z: `http://www.obitko.com/tutorials/neural-network-prediction/introduction.html`

[15] FRANK, R.J., N. DAVEY a S.P. HUNT. *Time Series Prediction and Neural Networks*. Department of Computer Science, University of Hertfordshire, Hatfield, UK.

[16] J. Xue, F. Yan, R. Birke, L. Y. Chen, T. Scherer and E. Smirni. PRACTISE: Robust prediction of data center time series. *2015 11th International Conference on Network and Service Management (CNSM)* [online]. IEEE, 2015, 126-134 [cit. 2017-04-14]. ISBN 978-0-7695-5091-6. Dostupné z: `http://ieeexplore.ieee.org/document/6619533/`

[17] *False nearest neighbors* [online]. [cit. 2017-03-15]. Dostupné z: `http://www.mpipks-dresden.mpg.de/~tisean/TISEAN_2.1/docs/chaospaper/node9.html`

[18] ZHANG, Guoqiang, Eddy PATUWO a Michael Y. HU. *Forecasting with artificial neural networks: The state of the art* [online]. [cit. 2017-04-14]. Dostupné z: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.138.4828&rep=rep1&type=pdf`

[19] THIESING, Frank M. a Oliver VORNBERGER. *Time Series Prediction by Neural Networks* [online]. [cit. 2017-04-15]. Dostupné z: `http://www2.inf.uos.de/papers_pdf/xplorer_report_frank.pdf`

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

CPU	Central processing unit
HDD	Hard Disk Drive
IoT	Internet of Things
JSON	JavaScript Object Notation
RAM	Random Access Memory
Rprop	Resilient propagation
SQL	Server Query Language
TDNN	Time delay neural network

SEZNAM OBRÁZKŮ

SEZNAM TABULEK

SEZNAM PŘÍLOH

PŘÍLOHA P I. NÁPOVĚDA PRO PRÁCI S APLIKACÍ

```
System Resources Predictor, Author: Martin Basel\n" +
Help:
SystemResourcesPredictor [options]
options:
      -?, -h, --help zobrazeni napovedy
      --inputFile=[filename] vstupni soubor
      --resultType=[Csv|Chart] vystupni metoda, bud vystup do csv
            souboru nebo do canvasjs tabulky
      --trainingMethod=[BackPropagation|ResilientPpropagation]
            ucici metoda
      --inputDataType=[CpuUsage|RamUsage|CpuAndRamUsage]
            typ dat, ktera vstupuji do trenovaci mnoziny
      --outputDataType=[CpuUsage|RamUsage|CpuAndRamUsage]
            typ dat, ktera se predpovidaji
      --displayOnlyPredictedRange=[true|false] prepinac, zdali
            se maji zobrazit vsechna data, nebo jen predpovezena
      --trainingSetBoundary=[0..N] interval pro vyber trenovaci
            mnoziny
      --testingSetBoundary=[0..N] interval pro vyber testovaci
            mnoziny
      --networkConfig=[string] ciselne hodnoty oddelene pomlckou,
            kde kazda hodnota urcuje pocet neuronu ve vrstve
      --learningError=[double] nastavuje akceptovatelnou chybu
            pro ukonceni trenovani
      --learningRate=[double] nastavuje parametr Learning rate
            pro algoritmus backpropagation
      --momentum=[double] nastavuje parametr momentum pro
            algoritmus backpropagation
      --dbServer=[string] databazova instance
      --dbName=[string] nazev databaze
      --dbUser=[string] ucet pro pristup k databazi
      --dbPassword=[string] heslo pro pristup k databazi
      --nodeId=[int] identifikace zarizeni v databazi
```

PŘÍLOHA P II. CD SE ZDROJOVÝMI KÓDY APLIKACE

Přiložené CD obsahuje zdrojové kódy naimplementované aplikace, ukázky výstupů ve formě *.html* souborů, trial verzi knihovny CanvasJS a vyexportované soubory z databáze, obsahující naměřená data.