TAJMAC-ZPS

Zlín

Neuronové sítě - velmi stručný úvod

Petr Lukašík

Do TEXu nasázel Petr Lukašík, 24. října 2022.

Obsah

1 Úvod

Umělé neuronové sítě se inspirovaly strukturou lidské nervové soustavy. Základním prvkem přirozené i umělé neuronové sítě je neuron neboli perceptron. Neurony jsou navzájem propojeny a předávají si signály. Platí přitom, že každý neuron může mít více vstupů, ale jen jeden výstup (tento výstup ale může být poslán i více než jednomu dalšímu neuronu). Vstupem neuronu může být buď výstup z jiného neuronu nebo informace z vnějšku (v našem příkladu by se jednalo o naměřené parametry stroje). Každý vstup má určitou váhu.

Primárním úkolem systému neuronových sítí je adaptabilní a učící se systém autonomního sběru dat a jejich analýzy. V oblasti sběru dat z obráběcích strojů je významný proces analýzy a následné flexibilní vyhodnocení anomálií a poruchových stavů. Proces vyhodnocování poruchových stavů lze v průběhu času zpřesňovat pomocí modelů strojového učení. Poruchové hodnoty a mezní stavy jsou neustále zpřesňovány v průběhu sledovaného procesu.

Klasická konstrukce počítače Von Neumannova nebo Harwardského typu potřebuje pro úspěšné plnění zadaného úkolu přesný algoritmus v podobě programu, který je vybudován nad přísnými zákony matematické logiky. A samozřejmě úspěch výsledného řešení je závislý na kvalitě vloženého programu. Příkladem může být úkol k řešení triviální kvadratické rovnice. Pokud je předložen kvalitní algoritmus, počítač vrátí vždy správný výsledek. Pokud je předložen chybný algoritmus, bude vrácen vždy nesprávný výsledek nebo může dokonce spadnout program při neošetřené výjimce dělení nulou.

Neuronová síť pracuje na zcela odlišném principu a to přestože je obvykle tvořena výpočetními jednotkami Von Neumannovského nebo Harwardského typu. Díky vlastnosti procesu učení nepotřebuje vstupní algoritmus. Naučit

řešit neuronovou síť kvadratickou rovnici by asi nebyl problém. Výsledkem by však nikdy nebylo přesné řešení, ale vždy něco co se ke kýženému výsledku blíží. Proto se neuronové sítě používají v oblastech, kde nelze použít exaktního algoritmu. Příkladem může být robot, jehož úkolem je zavázat si tkaničky od bot. Exaktní algoritmus (přestože si boty úspěšně zavazujeme dnes a denně) by byl patrně pěkný oříšek. Ale naučit robota, jehož mozek je tvořen neuronovou sítí, zavazovat boty metodou pokusů a omylů není až taková utopie. Pokud by se nám pro tento účel podařilo vytvořit skutečně exaktní algoritmus, pak by tkaničky byly zavázány vždy naprosto přesně a exaktně. Vždy by byly utaženy stejnou silou a pokaždé by mely například naprosto stejně dlouhé smyčky. Neuronová síť si s tímto úkolem poradí také, ale poněkud odlišně. Tkaničky nebudou nikdy stejně zavázány. Řešení se bude vždy blížit k optimu (v jistém tolerančním pásmu), ale pokaždé bude jiné.

Počátky výzkumu v oblasti neuronových sítí lze datovat do roku 1943 kdy pánové Warren McCulloch a Walter Pitts, vytvořili první modely umělých neuronů. Fungovaly tak, že jejich výstup (log.0 nebo 1) závisel, zda vážená suma vstupních signálů překročila jistou prahovou hranici. V padesátých letech navrhuje Frank Rosenblatt první neuronovou síť perceptron, který již obsahuje učící se pravidla. První perceptrony byly využívány k rozpoznávání vzorů. Nutno poznamenat, že myšlenka perceptronu je převratná, protože konstrukce perceptronu tvoří jádro všech současných ANN (Artificial Neural Network) sítí.

Počáteční nadšení z neuronových sítí však upadá, když bylo zjištěno, že navržený perceptron umí řešit jen tzv. lineárně separovatelné úlohy. Neumí například vyřešit veledůležitou logickou funkci XOR. Tím pádem začíná zájem o neuronové sítě upadat. V 80 letech byl navržen vícevrstevný perceptron, který je již využitelný pro libovolné klasifikační problémy. K učení tohoto typu sítě je využit algoritmus backpropagation. Díky tomu, zájem o neuronové sítě začíná výrazně narůstat. Zejména v roce 1982, kdy John Hopfield navrhl nový typ, tzv. auto-asociativní neuronovou síť.

V roce 1988 Kohonen přišel s úplně novou myšlenkou, představující typ sítě, která nepotřebuje učitele, ale sama o sobě mění své vnitřní chování, aniž by se

od člověka dozvěděla, zda se chová dobře nebo špatně. Jádrem této sítě je tzv. samoorganizující se mapa (SOM), jejíž princip je založen na „soutěživosti“ mezi jednotlivými neuronovými buňkami. Kohonen tuto síť použil k analýze lidské řeči. Neuronové síťě typu SOM vynikají v analýze dat technické diagnostiky, finančních transakcí a marketingových aktivitách.

Výraznou a ceněnou vlastností ANN je schopnost zobecňovat problémy. Není možno v procesu učení síť seznámit se všemi možnými situacemi. Proto je důležité aby sít měla i tuto zásadní vlastnost. Paradoxně se zde naráží na tzv. implementační problém, kdy velmi rozsáhlé sítě sice umí přesněji rozhodnout, pokud se pohybují v naučených problémech, ale mnohem hůře uplatňují vlastnost problémy zobecňovat. V praxi to tedy znamená, že ne vždy velmi rozsáhlá síť přinese lepší výsledky. Obecně jako ostatně všude i zde platí že, mnohdy „méně znamená více...“

Rád bych předem varoval, před přílišným optimismem, že nasazení neuronových sítí je vše spasitelné a vše objímající. Takových skvělých nadšení již v minulosti bylo dost, abychom na základě těchto zkušeností střízlivě uvážili, že současné neuronové sítě mají řadu nedostatků a nesou s sebou spoustu problémů, které zbývá vyřešit.

Jedním ze zásadních nedostatků je, že neuronovým sítím, díky tomu, že jsou vybudovány na křemíkových strukturách, chybí řekněme „třetí rozměr“ uvažování. Stávající sítě umí perfektně rozhodovat ve dvou rozměrech, což zjednodušeně znamená matematicky přesné, jestliže X pak s vysokou pravděpodobností Y.

Pro úspěšné rozhodnutí problému jsou však nutné ještě další vlastnosti, jako například soucit, zodpovědnost za rozhodnutí, pokora, nebo úcta [1], které do rozhodování přinášejí velmi důležitou další dimenzi. Toto současné neuronové sítě neumějí. Klidně, bez mrknutí oka rozpoutají třetí světovou válku, pokud uváží že právě vzniklá konstelace společenských problémů bude tímto způsobem optimálně vyřešena. Bez problémů odmítnou poskytnout úvěr matce

[1] Co si budeme povídat, tyto vlastnosti chybí i mnoha lidem, kteří se i na těch nejvyšších postech, snaží bravurně rozhodovat.

samoživitelce, protože díky naučeným kritériím ji ohodnotí jako velmi rizikového klienta. Takže, nasazení neuronových sítí je velmi výhodné pro řadu úloh, kde je nutné rychlé a přesné rozhodování, ale pro řadu situací jejich nasazení je amorální a bohužel poměrně časté.

V předkládaném textu se pokusím shrnout základní vlastnosti neuronových sítí typu vícevrstvý perceptron, konvoluční neuronová síť (CNN), rekurentní neuronová síť (RNN) a sítě typu Self Organizing Network - SOM.

Pro každý typ sítě se pokusím prezentovat typický příklad, v Pythonu, který problematiku lépe vysvětlí. Python, díky své jednoduché interpretaci a triviální modifikovatelnosti kódu (který výrazně usnadňuje ladění) je skvělým a ve většině projektů primárním programátorským prostředím pro návrh neuronových sítí. Pozor však na to, že Python představuje nevrchnější vrstvu rozhraní neuronová síť - programátor.

Skutečné jádro představuje knihovna TensorFlow [2] a knihovna Keras, která tvoří vrstvu mezi TenzorFlow a našim Pythonovským kódem. TensorFlow jak název napovídá je velmi rozsáhlá knihovna poctivě napsaná v C a C++, využívající paralelní procesory grafické karty Nvidia. [3] K pochopení a plnému využití knihovny TenzorFlow je dobrá znalost základů lineární algebry. Pro oživení této oblasti doporučuji [21][6][16].

[2] Open Source Licence Googlu a díky své kvalitě, obecně nejvíce používaná.
[3] GPU však není podmínkou nutnou, běží i na CPU

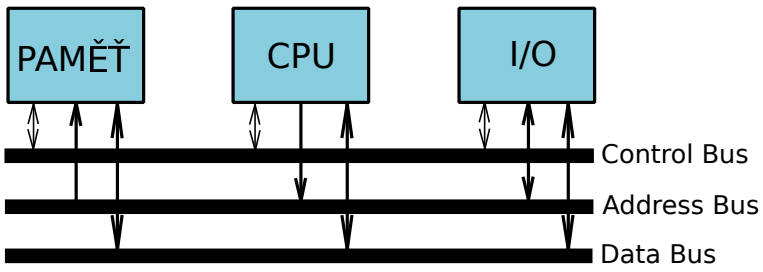
2 Architektury počítačů pro paralelní zpracování

Návrhové schéma architektury počítače je inspirováno řadou podnětů založených na čistě technických předpokladech (Turingův stroj, Von Neumannova nebo Harwardská koncepce), jejichž předpokladem je čistý algoritmický přístup, kdy vložením jistého algoritmu získáme jistý výsledek. Druhá oblast architektury využívá principů založených na evolučních zákonitostech Přírody, kam lze zahrnout neuronové sítě nebo genetické algoritmy a kvantové výpočty.

2.1 Architektury založené na technickém návrhu

Von Neumannova architektura

Von Neumannova architektura představuje jednoduché schéma programovatelného počítače, které používá jednu sběrnici, na niž jsou připojeny všechny aktivní prvky (procesor, paměť, vstupy a výstupy).

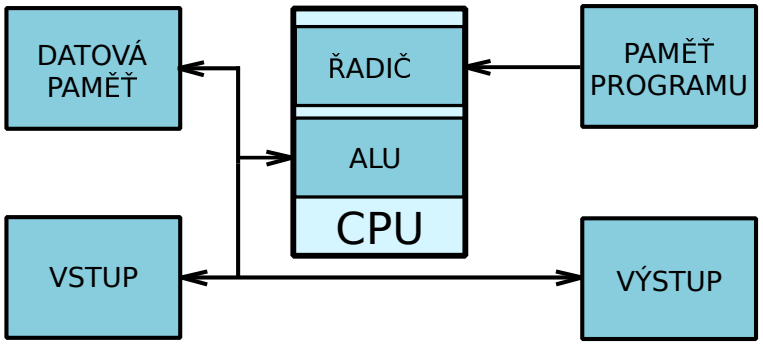


Obrázek 2.1: *Schema von Neumannovy architektury*

Nevýhodou Von Neumannovy koncepce je výrazně vyšší rychlost zpracování instrukcí než rychlost komunikace s pamětí. To výrazně snižuje výkonnost Von Neumannova konceptu.(memory wall, - paměťová zeď). Tuto nevýhodu do jisté míry řeší paměťové cache, do kterých se potřebná data a instrukce z pomalejší hlavní paměti načítají před vlastním zpracování.

Harwardská architektura

Harwardská architektura, se vyznačuje dvěma samostatnými paměťmi a sběrnicemi pro program a data. To umožňuje výraznou optimalizaci výkonu a paralelního zpracování instrukcí. Instrukce v programové paměti jsou prováděny pomocí jednoúrovňové pipeline. Během provádění jedné instrukce je následující instrukce předem načtena z programové paměti. Tento koncept výrazně optimalizuje provádění jednotlivých instrukcí, z nichž většina je zpracována v jediném hodinovém cyklu.



Obrázek 2.2: *Schema Harwardské koncepce*

Nevýhodou této architektury je velmi komplikovaný návrh sběrnice. Proto se tato architektura využívá ve speciálních případech, zejména u jednoúčelových procesorů, například v oblastech paralelních výpočtů (grafické karty). Dále je použita například v mikrokontrolerech Atmel kde je tato sběrnice včetně pamětí integrována v jediném čipu, což i přes komplikovanější sběrnici přináší výrazný nárůst výpočetního výkonu.

2.2 Architektury založené na principu Přírodních zákonů

Současný trend masivního paralelního zpracování není samospasitelný, protože naráží díky Amdahlovu zákonu na technologická omezení. Mooreův zákon o růstu tranzistorů v integrovaném obvodu také narazí na svoji technologickou hranici. [1] Kniha autorů Greenlaw, Hoower a Ruzzo [9] definuje hranice současných paralelních výpočetních systémů.[2]

Na scénu proto nastupují další zajímavé obory, které si budují své místo na výsluní informatiky. Kromě systémů založených na paralelismu biologických struktur se výrazně prosazuje obor kvantové informatiky, jejíž prvotní základy položili Richard Feynman a Ed Fredkin [27].

Význam kvantových výpočtů je patrný z následujícího tvrzení. Máme-li klasický osmibitový registr, jsme schopni v něm provést najednou 1^8 výpočtů. Máme-li ovšem k dispozici kvantový osmi qubitový registr, jsme schopni provést v jednom časovém okamžiku 2^8 výpočtů. A to opravdu stojí za pozornost. (Kvantový registr o délce 2^{500} dokáže provést paralelně v jeden okamžik tolik výpočtů, kolik je zhruba atomů v pozorovatelném vesmíru [20].

Bioinformatika

Charles Bennett z výzkumného centra IBM popsal v roce 1982 pozoruhodnou podobnost mezi využitím DNA struktur Přírodními mechanismy a způsobem, jak jsou používány databáze při řešení konkrétních úkolů. Leonard Adleman (jeden z autorů RSA šifry) předvedl, že shluk DNA molekul může řešit jednoduché kombinatorické úlohy. A v roce 2002 předvedl systém, který je schopen řešit úkol s dvaceti proměnnými. Společnou vlastností DNA počítačů je předpo-

[1] Definice Mooreova zákona je silně závislá na technologických možnostech a fyzikálních limitách polovodičových materiálů. Odhaduje se, že za 20, až 40 let narazí klasická technologie výroby polovodičových čipů na svoji hranici [7].

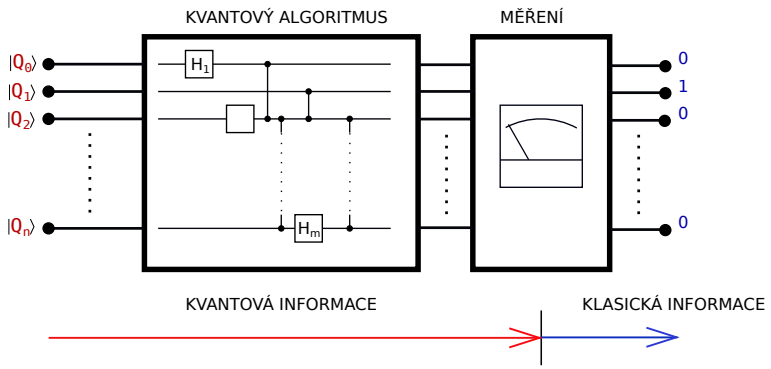
[2] Vypadá to že jsme narazili na hranici toho, čeho je možné dosáhnout s počítačovými technologiemi. Člověk by si ale měl dávat pozor na takováto tvrzení, protože do 5 let se obvykle ukáží jako pěkná pitomost. (John von Neumann, 1949)

kládaný masivní paralelismus. Leonard Adleman popisuje v článku [1] princip využití DNA struktur pro masivní paralelní výpočty.

Ehud Saphiro, Tom Ran a Shai Kaplan z Weizmann Institute of Science zveřejnili článek [22], kde představují principy využití biologických molekulárních struktur, vhodných pro řešení paralelních úloh.

Přestože zatím existuje mnoho nevyřešených překážek, lze s velkou pravděpodobností předpovídat velmi úspěšné řešení a perspektivu. Jedná se o nesmírně zajímavou oblast, kterou Příroda dovedla k absolutní dokonalosti. No upřímně řečeno, měla dvě obrovské výhody. Za prve dostatek času, a za druhé moc jí do tohoto vývoje nikdo nezasahoval. Proto má zatím v této oblasti absolutní prim. Nám nezbývá nic jiného nežli se úporně snažit, abychom dosáhli také skvělých výsledků. Bohužel však nemáme k dosažení podobné dokonalosti ani zlomek času, který měla k dispozici Příroda.

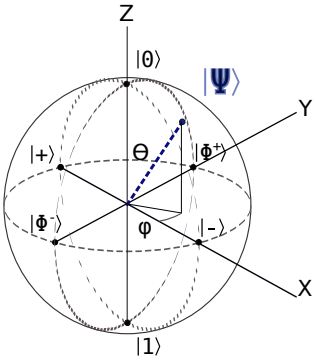
Kvantový výpočetní systém



Obrázek 2.3: *Hrubý nástin kvantového výpočtu*

Využití kvantových vlastností částice (elektronu, fotonu), jako nositele informace. V klasickém výpočetním systému se využívá deterministického přístupu kódování informace na základě binárního kódu, jehož hodnotami jsou předem dohodnuté dvě hladiny napětí, které představují základní informační jednotku - bit. Kvantově mechanický přístup definuje informační jednotku - qubit, která nabývá nekonečné množství hodnot mezi nulou a jedničkou. Tento

přístup připomíná princip analogového počítače, který byl využíván pro modelování dynamických jevů v oblasti regulace. Klasický analogový počítač však postrádá masivní paralelismus, který je ovšem kvantovému počítači vlastní.



Obrázek 2.4: *Schema Qubitu*

Obecně, každá klasická výpočetní operace (NAND, XOR,...) je prováděna nevratným způsobem. [3] To znamená, že se nelze vrátit na začátek výpočtu, nebo dokonce nelze rekonstruovat, z jakých vstupních hodnot vznikl výsledek. Obecně víme, že výsledkem binární operace NAND jsou dvě vstupní hodnoty. Pro $log(1)$ na výstupu hradla nezáleží, zda byla $log(0)$ na jednom ze vstupů a na kterém nebo na obou – důležitý je výsledek. To vede jednoznačně ke ztrátě informace a nárůstu entropie systému. Ten se díky tomuto jevu zahřívá [20].[4] Tím je určena technologická hranice klasických výpočetních systémů, která je právě omezena nárůstem vnitřních energetických ztrát křemíkových čipů při snižující se velikosti jednotlivých konstrukčních prvků.

Jakkoliv nalezené kvantové algoritmy vypadají slibně, zásadní problém spočívá v tom, že není vůbec snadné takový kvantový systém sestrojit. Kvantové bity jsou buď velmi citlivé na rušivé vlivy okolí, nebo je značně obtížné je přinutit, aby spolu navzájem komunikovaly. Má-li být kvantovou informací například spin elektronu, musíme jej velmi dobře izolovat od okolí, pokud nechceme, aby se z $log(1)$ jedničky stala samovolně $log(0)$. Protože spin je velmi

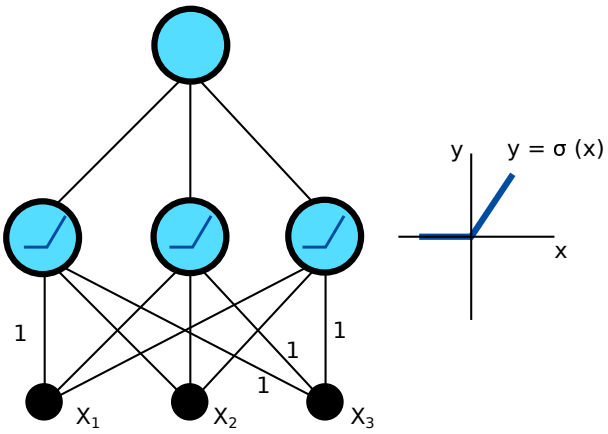
[3] I triviální operace součtu je nevratná - z výsledku nelze vyčíst, z čeho byl složen $1+2=3$ nebo $2+1=3$ nebo $1+1+1=3$.

[4] Samozřejmě, že k růstu teploty na hradle přispívá do rovnice energetické bilance soustavy více činitelů. Ztráta informace je však jedním z nich.

citlivý na magnetické pole. Dalším problémem jsou fázové posuvy při interakci atomů s okolím. Kvantové superpozice jsou velmi křehké. V tom také spočívá jedna z odpovědí, proč nepozorujeme Schrödingerovy kočky, tedy superpozice dvou různých stavů (živá a zároveň mrtvá) u makroskopických objektů. Vlivem i nepatrné interakce s okolím, se systém dostává do jednoho z normálních stavů, které jsme zvyklí pozorovat [20].

Neuronové sítě

Umělá neuronová síť (ANN, artificial neural network), využívá ke své práci množství propojených procesorů a výpočetních cest. ANN je inspirována architekturou lidského mozku. Velmi důležitou vlastností je schopnost učit se a adaptovat se na rozsáhlé a komplexní množiny dat, které klasickým algoritmem nelze efektivně zvládnout, a nebo jen velmi těžko zvládnou.[5]



Obrázek 2.5: *Schema dopředné neuronové sítě*

Požadované řešení síť sama abstrahuje a zobecňuje. Charakter řešení hledá v adaptivním režimu procesu učení ze vzorových příkladů. V tomto smyslu neuronová síť připomíná inteligenci člověka, který získává mnohé své znalosti a dovednosti ze zkušenosti, kterou ani není ve většině případů schopen formulovat analyticky podle příslušných pravidel či algoritmu.

[5] Viz úvod, jak naučit robota zavazovat boty.

První umělou neuronovou síť navrhl v roce 1957 psycholog Frank Rosenblatt. Byla nazvána Perceptron a smyslem její existence mělo být modelování postupu, při kterém lidský mozek zpracovává vizuální data a učí se rozeznávat objekty.

Toto jsou jen v krátkosti nastíněné základní problémy, kde za každým z nich se skrývá obrovské množství velmi zajímavého vědeckého výzkumu, a které napovídají, že výpočetní systémy založené na biologických a kvantových zákonech jsou velkou výzvou. A to i přes technologické překážky, které představují.

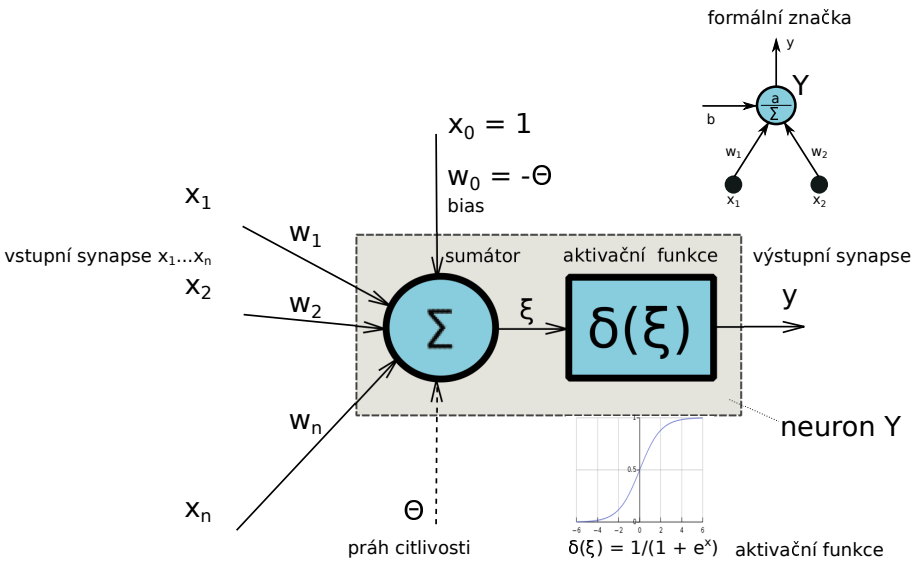
3 Použité značení

w_{ij}	–	Váha vstupní synapse neuronu
$w_{0i} = b_i$	–	Bias i-tého neuronu
ξ_i	–	Excitační úroveň (aktivační potenciál neuronu)
$\mathbf{W}$	–	Váhová matice $\mathbf{W} = \{w_{ij}\}$
$\mathbf{w_i}$	–	Vektor vah $\mathbf{w_i} = (w_{1i}, ..., w_{ni})$
$\|x\|$	–	Norma (velikost) vektoru $\mathbf{x}$
Θ_j	–	Práh citlivosti neuronu (treshold)
$\mathbf{s}$	–	Tréninkový vstupní vektor $\mathbf{s} = (s_1, ..., s_n)$
$\mathbf{t}$	–	Tréninkový výstupní vektor $\mathbf{t} = (t_1, ..., t_m)$
$\mathbf{x}$	–	Vstupní vektor $\mathbf{x} = (x_1, ..., x_n)$
Δw_{ij}	–	Změna váhy $\Delta w_{ij} = [w_{ij(\text{new})} - w_{ij(\text{old})}]$
α	–	Koeficient učení
σ	–	Aktivační (přenosová) funkce
$y_{ij} = \sigma(\xi_{ij})$	–	Výstupní hodnota neuronu i,j
$\overline{y}$	–	Požadovaná výstupní hodnota neuronu
Y_{ij}	–	Neuron i,j
$J(y)$	–	Nákladová funkce (MSE - Mean Squared Error)

4 Formální neuron - Perceptron

Perceptron je nejjednodušším modelem dopředné neuronové sítě. Sestává pouze z jednoho neuronu. Perceptron navrhl Frank Rosenblatt v roce 1957. Přes úvodní nadšení bylo později zjištěno, že jeho užití je velmi omezené, neboť je možné ho použít pouze na lineárně separovatelné množiny. Jeho rozšířením je vícevrstevný perceptron, který má již mnohem širší možnosti použití.

4.1 Formální neuron - definice



Obrázek 4.1: *Schema formálního neuronu*

Definice

Základní stavební dílec neuron má jeden výstup a libovolný počet vstupů. Každý výstup je prezentován jednou hodnotou, která je dále šířena neuronovou sítí. Vážená suma vstupních hodnot ξ (hodnot z výstupu předchozího neuronu) představuje vnitřní potenciál neuronu Y:

$$\xi = \sum_{i=1}^{n} w_i x_i \tag{4.1}$$

Synapse

Synapse jsou spojení mezi jednotlivými neurony. Počet synapsí určuje velikost či mohutnost neuronové sítě, tzn. že větší počet synapsí v neuronové síti má možnost uchovat větší množství dat.

Momentum - hybnost

Momentum (hybnost) je koeficient, který pomáhá při učení sítě v udržení zpětného šíření algoritmu. V praxi to znamená asi něco takového, že udržuje učící algoritmus v dynamice učení.

Hybnost je základní fyzikální veličina, která nutí pohybující se těleso, pokračovat ve své trajektorii. V podstatě stejný koncept hybnosti platí i pro neuronové sítě, kdy během tréninku má směr aktualizace tendenci odolávat změnám. To znamená že při procesu učení se může neuronová síť zastavit v lokálním (a neoptimálním) minimu. Uplatnění faktoru hybnosti umožňuje lokální minimum přeskočit a zastavit se v hlubším (optimálním) lokálním minimu.

$$\Delta w_{ij} = \left(\eta \frac{\partial J}{\partial w_{ij}} \right) \tag{4.2}$$

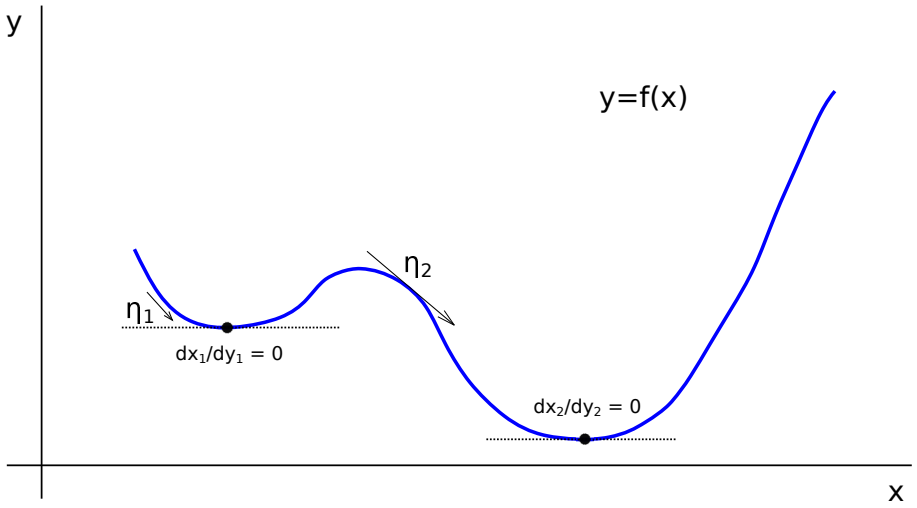
kde: Δw_{ij} je inkrement váhy, η je rychlost učení a $\partial E / \partial w_{ij}$ je gradient váhy.

Pro zavedení faktoru momentum (hybnost) platí stejné zásady jako při uplatňování stability systému. Příliš velká hybnost, může systém rozkmitat,

to znamená, že systém nikdy nenalezne optimální hodnotu gradientu a tudíž optimálního minima.

Problémem velmi rozšířených gradientních metod učení, je nalezení lokálního minima. Toto nalezené minimum však nemusí v řadě případů představovat skutečné globální minimum, které může být v jiné části hledané funkce. To znamená, že vzniká nebezpečí, že se proces učení zastaví v některém z lokálních minim, které však nepředstavuje skutečné hledané minimum.

Tento problém lze odstranit právě změnou hybnosti soustavy zavedením konstanty - rychlost učení $\eta \in [0,1]$. Touto konstantou lze optimalizovat rychlost výpočtu gradientu funkce a tím zabránit, aby se výpočet zastavil v některém z lokálních minim.



Obrázek 4.2: *Moment hybnosti v procesu učení*

Na obrázku 4.2 je názorná představa hybnosti soustavy v souvislosti s vyhledáváním minima funkční závislosti. Jinými slovy, pokud je rychlost učení příliš malá, může mít za následek, proces vyhledávání zamrzne v některém z lokálních minim a na druhou stranu při příliš velké rychlosti učení může globální minimum přeskočit.

Bias

Druhou možností, jak zabránit síti spočinout v lokálních minimech je zavedení další vstupní proměnné biasu b neboli zkreslení, které nedovolí při procesu učení zamrznout v lokálních minimech, ale pořád ji nepatrně rozvažuje. Tím nutí algoritmus konvergovat ke globálnímu minimu. Každý neuron, kromě první vrstvy, má vstup s váhou w_0, která se nazývá bias. Pak lze zapsat rovnici vnitřního potenciálu neuronu.

$$\xi = \sum_{i=1}^{n} w_i x_i = w_0 + \sum_{i=1}^{n} w_i x_i = b + \sum_{i=1}^{n} w_i x_i \tag{4.3}$$

Bias se snaží přiblížit ke stavu, kde začíná mít hodnota nového neuronu smysl. Bias může mít kladnou i zápornou hodnotu.

Aktivace

Hodnota vnitřního potenciálu (vážené sumy vstupů, vah a biasu) ξ indikuje stav na výstupu neuronu y. Nelineární průběh výstupní hodnoty $y = f(\sigma)$ při dosažení hodnoty potenciálu - aktivace a je dán tzv. *aktivační* funkcí $\sigma(\xi)$. Jinými slovy, po dosažení této hodnoty, pošle neuron na výstup signál. Nejjednodušším typem aktivační funkce je jednotkový skok (ostrá nelinearita) která je popsána následovně.

$$y = \sigma(\xi) = \begin{cases} 1 & \text{jestliže} \quad \xi \geq 0 \\ 0 & \text{jestliže} \quad \xi < 0 \end{cases} \tag{4.4}$$

Pokud místo váhového biasu pracujeme s fixním prahem citlivosti Θ, pak aktivační funkce typu ostrá nelinearita vypadá následovně:

$$y = \sigma(\xi) = \begin{cases} 1 & \text{jestliže} \quad \xi \geq \Theta \\ 0 & \text{jestliže} \quad \xi < \Theta \end{cases} \tag{4.5}$$

Váha

V biologických sítích jsou zkušenosti uloženy v dendridech (stromech) - výběžcích nervových buněk. V umělých neuronových sítích jsou zkušenosti uloženy v jejich matematickém ekvivalentu - *váhách*. Váha je hodnota, která je násobitelem nesené informace. V podstatě se jedná o paměť hodnoty, která je představitelem řešeného algoritmu.

Váhou je násobena každá ze vstupních hodnot neuronu. Velikost váhy w_i vyjadřuje uložení zkušeností do neuronu. Čím je vyšší hodnota, tím je daný vstup důležitější. V biologickém neuronu práh Θ označuje prahovou hodnotu aktivace neuronu.

To znamená, že: je-li suma vstupních hodnot neuronu menší než práh citlivosti Θ, pak se neuron nachází v pasivním stavu.

$$\sum_{i=1}^{N} x_i w_i < \Theta \rightarrow \text{neuron je v pasivním stavu} \tag{4.6}$$

Vstupní vrstva

Speciální případ neuronové vrstvy. Vstupy, nejsou vzájemně propojeny, ale slouží jen jako vstup do další vrstvy. Zde se neuplatňují váhy. Ty v tomto případě mají implicitní hodnotu 1.

Skrytá vrstva

Obvykle obsahuje mnohonásobně více neuronů, než vrstva vstupní a výstupní. Z hlediska principu funkce není u této vrstvy podstatný způsob, jak jsou interpretovány mezivýsledky zpracování. Významným rozdílem proti vstupní vrstvě je, že každý jednotlivý vstup je vzájemně propojen se všemi výstupy vrstvy předchozí.

Každý ze vstupů skryté vrstvy je prezentován *jednou synapsí* a každá synapse na vstupu do neuronu disponuje *jednou váhou*, která určuje význam

vstupního signálu. Skrytá vrstva může být vícevrstvá. Vícevrstvé sítě se uplatňují zejména u úloh s nelineárním průběhem řešení.

Výstupní vrstva

Má stejné vlastnosti jako skrytá vrstva, podstatným rozdílem však je, že převádí výsledky skryté vrstvy (vstupy do vrstvy) na výstupy neuronové sítě. Jinými slovy, zde lze očekávat výsledky řešení neuronové sítě.

Práh a aktivační funkce

Práh je hodnota, při které se aktivuje výstup neuronu. Hodnotu prahu určuje tzv. *aktivační* (přenosová) funkce σ , která upravuje výstup na hodnoty, které se dále šíří v neuronové síti.[1]

Úkolem aktivační funkce, je změnit měřítko výstupu tak, aby byl v intervalu $[0, 1]$.

- Skoková aktivační funkce (0 a 1 kde práh definuje zlom mezi 0 a 1)
- Sigmoida
- Hyperbolický tangens
- Signum funkce
- a mnohé jiné, ale s podobným průběhem jako sigmoida...

Nejpoužívanější aktivační funkcí je sigmoida. Každý výstup je násoben aktivační funkcí a výsledná hodnota, (výstup) je poskytnuta synapsí dále, na vstup do další neuronové vrstvy.

Výstupy ve výstupní vrstvě (poslední vrstva) jsou potom zpracovány jako konečné výstupy z celého modelu neuronové sítě.

[1] značení aktivační funkce řeckou sigmou (σ) vyplývá z názvu nejčastěji používané funkce - sigmoidy

Nákladová funkce

Nákladová funkce vyjadřuje *přesnost* procesu učení. Úkolem této funkce je ohodnotit, jak se proces učení (reálný stav neuronové sítě) přiblížil k optimálnímu, námi požadovanému funkčnímu průběhu. Nejčastěji se používá funkce MSE (Mean Square Error - Střední kvadratická chyba. Podrobně je popsána v 5.10

$$\mathrm{J}(y) \stackrel{\mathrm{def}}{=} \mathbb{E}\left[(y-\overline{y})^2\right] \tag{4.7}$$

Kde, y je skutečný průběh funkce, $\overline{y}$ je optimální (požadovaný) průběh funkce a J je střední kvadratická chyba, představující rozdíl mezi skutečným a požadovaným průběhem. Cílem učení je docílit stavu kdy $J \to 0$.

Z výše uvedeného vyplývá že k procesu učení přispívá mnoho faktorů. Složitost modelu, parametry typu rychlost učení, aktivační funkce, velikost vstupní datové sady a další. A právě nákladová funkce je měřítkem kvality procesu učení každé neuronové sítě.

4.2 Matematická interpretace neuronu

Na základě výše uvedených definic lze zapsat základní matematickou funkci, která platí pro jeden neuron a tu následně lze rozšířit na celou síť neuronů.

Pro jeden neuron platí, viz obrázek 4.1:

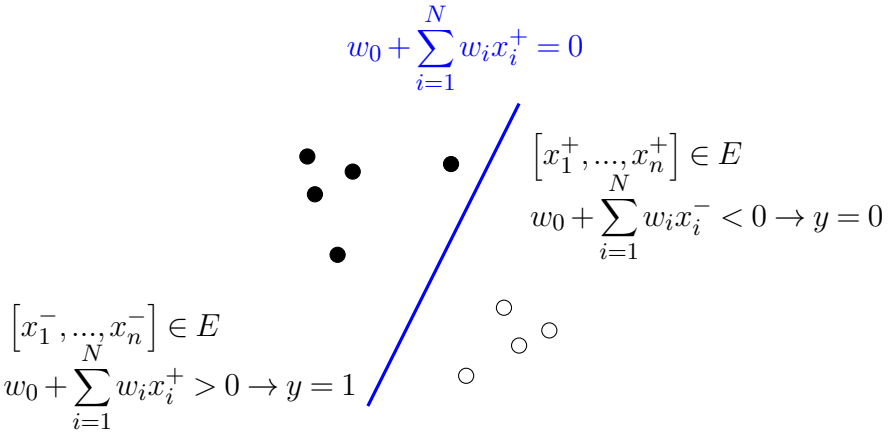
$$y = \sigma(\xi) = \sigma\left(\sum_{i=1}^{n} w_i x_i + w_0\right) \tag{4.8}$$

4.3 Geometrická interpretace neuronu

Pro vysvětlení geometrické interpretace, předpokládejme neuron se dvěma reálnými vstupy x_1, x_2 a odpovídajícími vahami w_1, w_2. Dále uvažujme, že vstupy

x_1 a x_2 představují body ve dvourozměrném prostoru. Dále předpokládejme, že výstupem neuronu je 0 nebo 1 $\Rightarrow$ realizuje zobrazení $\{R\} \rightarrow \{0,1\}$

Na základě těchto předpokladů provádí takto definovaný neuron klasifikaci bodů v rovině a rozděluje je do dvou skupin dle hodnoty aktivační aktivační funkce.



Obrázek 4.3: *Geometrická interpretace neuronu ve 2D prostoru*

Vztah vyjadřující odezvu neuronu, je definován pozicí jednotlivých bodů $\circ, \bullet$. V tomto konkrétním případě je zařazení bodů dáno jejich pozicí vůči přímce definované aktivační funkcí - přesněji vahami neuronu. Dělící přímka odlišující dvě skupiny bodů má tedy ve dvourozměrném prostoru rovnici:

$$x_1 w_1 + x_2 w_2 + b = 0 \tag{4.9}$$

kde x_1, x_2 jsou vstupní hodnoty a w_1, w_2 jsou váhy vstupních hodnot a b je bias.

Tato rovnice představuje ve 2D prostoru rovnici přímky, která dělí rovinu na dva podprostory, v nichž jsou jednotlivé body separovány. Neuron tedy klasifikuje ve kterém z obou podprostorů leží bod, který je určený vstupními souřadnicemi x_i, y_i a realizuje tzv. ditochomii (rozdělení celku do dvou vzájemně se nepřekrývajících částí) vstupního prostoru.

Ve vícerozměrném prostoru platí totéž:

$$x_1 w_1 + x_2 w_2 +, ..., + x_n w_n + b = 0 \tag{4.10}$$

Neuron Y_i je aktivní má li na výstupu $y_i = 1$ a pasivní má li na výstupu $y_i = 0$.

Logický neuron je binární prvek, který se vždy nachází právě v jednom ze dvou možných stavů, aktivním $log(1)$, anebo neaktivním $log(0)$

Neuronovou síť velikosti $[1, n]$, můžeme v každém časovém okamžiku jednoznačně reprezentovat binárním číslem délky $[n]$. Stav n-tého neuronu je reprezentován binární číslicí na n-tém místě binárního vektoru. Počet všech možných stavů celé sítě je roven 2^n přičemž délka binárního vektoru je n.

4.4 Logický neuron

McCulloch a Pitts ve svém článku [18] upozornili na to, že neuron definovaný dle 4.1 umí jednoznačně reprezentovat funkce matematické logiky. Původně se myslelo, že i lidský mozek funguje na principu binárních logických funkcí. To však je dnes díky těžkopádnosti takového modelu vyvráceno.

Logický neuron je definován jako výpočetní jednotka s několika vstupy a jedním výstupem.

Druhy vstupů:

- excitační, ve schématu kreslen jako šipka směřující k neuronu
- inhibiční, ve schématu kreslen jako šipka směřující k neuronu, která je navíc přeškrtnuta kolmou čarou

Na výstupu se objeví $log(1)$ právě tehdy, když se současně signály objeví na určitém počtu vstupů. Tento minimální počet aktivních vstupů se nazýván práh Θ. Práh může být roven jakémukoliv nezápornému číslu $\Theta > 0$.

Logický neuron představuje idealizaci reálného neuronu, ale zachovává tak nejdůležitější vlastnosti skutečného neuronu. Logický neuron mění svůj stav jen v diskrétních časových okamžicích. Výstup z jednoho neuronu se na vstu-

pu napojených neuronů objeví v následujícím časovém kroku. Síť logických neuronů se chová synchronizovaně. To znamená že okamžiky změny stavu jsou pro všechny neurony v síti totožné.

4.4.1 Formální definice logického neuronu

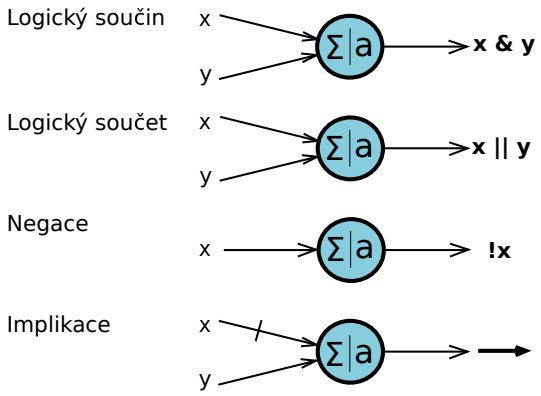
- Logický neuron se může nacházet v jednom ze dvou stavů, které se nazývají aktivní stav a neaktivní stav.
- Logický neuron má jeden výstup, který může být současně napojen do jednoho nebo více jiných neuronů. Může být napojen také zpětnovazebně na svůj vlastní vstup.
- Logický neuron má celkem $x_e + x_i$ vstupů, z nichž x_e jsou excitační vstupy a x_i vstupy inhibiční. Platí že $x_e > 0, x_i > 0$.
- Excitační vstupy jsou ohodnoceny jednotkovým váhovým koeficientem $w_e = 1$.
- Inhibiční vstupy jsou ohodnoceny jednotkovým váhovým koeficientem $w_i = -1$.
- Logický neuron má definován práh θ, přičemž $\theta \geq 0$
- Čas je kvantizován. To znamená že stav logického neuronu se může změnit pouze v diskrétních časových okamžicích (krocích), kde $t = k\Delta, k \in \mathbb{N}$ Stav logického neuronu zůstává nezměněn v časovém intervalu $(\Delta, k\Delta)$. Délka časového kroku Δ je stejná pro všechny neurony v síti. To znamená že neuronová síť je *synchronizována*.
- Daný vstup je aktivní v čase $(k+1)\Delta$ právě tehdy, když výstup neuronu, který je na tento vstup připojen, byl aktivní v čase $k\Delta$.
- $x_e(t)$ je počet aktivních excitačních vstupů v čase t.
- $x_i(t)$ je počet aktivních inhibičních vstupů v čase t.
- Platí : $x_e(t) \leq n_e, x_i(t) \leq x_i$ v každém časovém okamžiku t.

Logický neuron je aktivní ve všech časech $t \in [k\Delta, (k+1)\Delta$ právě když:

$$x_e(k\Delta t) - \phi x_i(k\Delta t) \geq \Theta. \tag{4.11}$$

Konstanta $\phi \geq 0$ charakterizuje požadovaný poměr mezi počtem aktivních excitačních a aktivních inhibičních vstupů pro dosažení aktivního výstupu neuronu.

4.5 Logický neuron prvního řádu



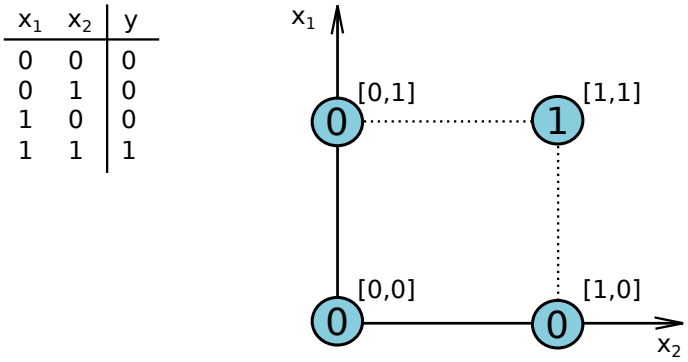
Obrázek 4.4: *Základní Booleovské operace s logickými neurony*

Neuronové sítě s neurony prvního řádu umí klasifikovat pouze lineárně separovatelné množiny dat. Lineární separabilita znamená, že hranice mezi jednotlivými separovatelnými objekty je přímka. Z matematického hlediska se jedná o Booleovské funkce AND (&), OR (|), NOT (!) a IMPLIKACE ($\Rightarrow$).

Vstupy do logického neuronu prvního řádu jsou *excitační*, které jsou popsány binárními proměnnými $(x_1, .., x_e)$ a *inhibiční*. Ty jsou popsány proměnnými $(x_{e+1}, .., x_{e+i})$. Excitační vstupy jsou násobeny váhovým koeficientem $(w = 1)$ a inhibiční vstupy jsou násobeny váhovým koeficientem $(w = -1)$.

Aktivitu logického neuronu prvního řádu lze definovat následovně:

$$y = f(\sigma) = \begin{cases} 1 & \text{jestliže} \quad \xi \geq 0 \\ 0 & \text{jestliže} \quad \xi < 0 \end{cases} \quad \text{kde } \xi = \sum_{i=1}^{e} x_i - \sum_{i=e+1}^{n} x_i + b \tag{4.12}$$

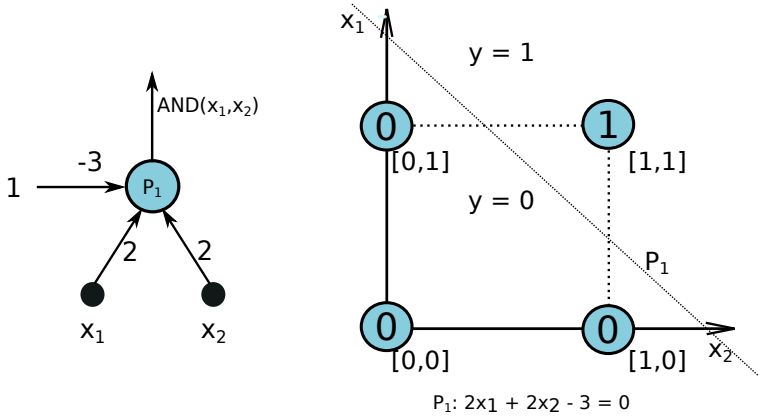


Obrázek 4.5: *Lineárně separabilní funkce AND*

Logické hradlo AND s pomocí neuronu prvního řádu

Z obrázku 6.4 vyplývá, že Booleovské hodnoty v jednotlivých vrcholech grafu lze separovat přímkou. To znamená, že funkce AND je lineárně separabilní. Obecně, řešením může být jakákoliv přímka rozdělující rovinu na nadroviny, které separují vrcholy s $log(0)$ a vrcholy s $log(1)$ Této konfiguraci vyhovuje například tato rovnice přímky.

$$x_1 + x_2 - \frac{3}{2} = 0 \tag{4.13}$$



Obrázek 4.6: *Grafické řešení lineárně separabilní funkce AND*

váhový vektor $w = [1, 1, -3/2]$ Pro každý vstupní vektor se nyní vypočítá výstupní signál jako

$$\xi = [1,1,-3/2] \begin{bmatrix} x_1 \\ x_2 \\ 1 \end{bmatrix}; \rightarrow \xi = \begin{cases} 1 & \text{jestliže} \quad \xi \geq 0 \\ 0 & \text{jestliže} \quad \xi < 0 \end{cases} \tag{4.14}$$

Na základě výše uvedeného lze jednoduché Booleovské funkce interpretovat následovně:

Rovnice logické funkce součin - AND

$$y_{[and]} = f(x_1 + x_2 - 3/2); \tag{4.15}$$

Rovnice logické funkce součet - OR

$$y_{[or]} = f(x_1 + x_2 - 1); \tag{4.16}$$

Rovnice logické funkce implikace - $\rightarrow$

$$y_{[\rightarrow]} = f(-x_1 + x_2 - 0); \tag{4.17}$$

Rovnice logické funkce negace - NOT

$$y_{[not]} = f(x_1 - 1); \tag{4.18}$$

4.6 Logický neuron vyššího řádu

Logický neuron prvního řádu neumí definovat důležitou logickou funkci XOR. Toto byl důvod, proč po slibných začátcích, odsunula výzkumná obec problémy neuronových sítí na vedlejší kolej.

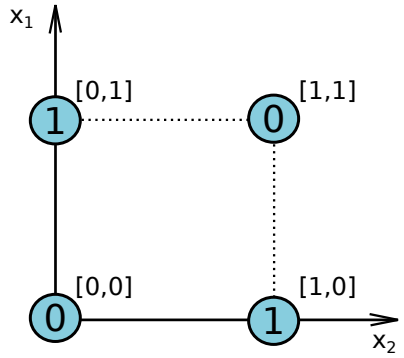
Omezení neuronu prvního řádu lze odstranit pomocí *logických neuronů vyšších řádů*, které jsou schopny zpracovat i množiny objektů nelineárně separovatelných (k separaci se nepoužívá přímka, ale obecná funkční závislost).

Definice neuronu vyššího řádu:

$$y = f\left(\sum_{i=1}^{N} x_i w_i + \sum_{i,j=k,i<j} w_{ij} x_i x_j + \cdots + b\right) \tag{4.19}$$

Tato definice obsahuje neuron prvního řádu plus kvadratický člen, případně případně další členy vyšších řádů.

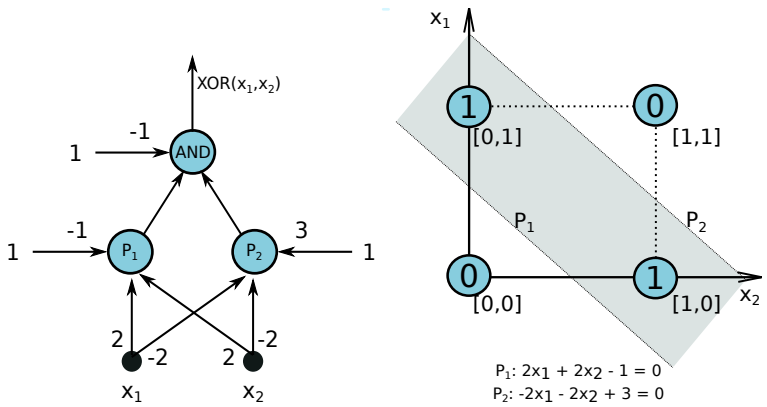
x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0



Obrázek 4.7: *Lineárně neseparabilní funkce XOR*

Příkladem řešení neuronem vyššího řádu může být například logická funkce XOR, která není lineárně separabilní.

Jinými slovy: pro řešení XOR problému si již nevystačíme s jedním neuronem, ale musíme do řešení zapojit dvouvrstvou neuronovou sít, která je schematicky znázorněna na obrázku: 4.8.



Obrázek 4.8: *Grafické řešení lineárně neseparabilní funkce XOR*

Řešením funkce XOR jsou ve 2D prostoru dvě přímky, jejichž souřadnice bodů jsou známy a jsou popsány touto obecnou rovnicí.[2]

$$\frac{y-y_1}{y_1-y_2}=\frac{x-x_1}{x_1-x_2} \tag{4.20}$$

jinak napsáno platí:

$$(y-y_1)(x_1-x_2)-(y_1-y_2)(x-x_1)=0 \tag{4.21}$$

Řešením této přímky pro body v rovině $[x_1=0,y_1=1/2]$ a $[x_2=1/2,y_1=0]$ je:

$$2y+2x-1=0 \tag{4.22}$$

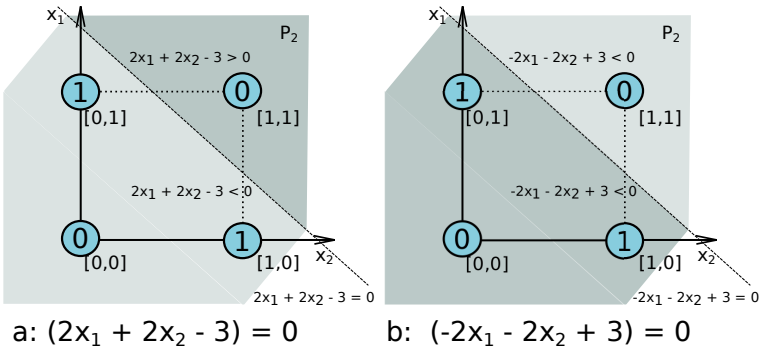
a pro body $[x_1=0,y_1=3/2]$ a $[x_2=3/2,y_1=0]$:

$$2y+2x-3=0 \tag{4.23}$$

[2] pro snazší pochopení jsou souřadnice přímek (vstupů do neuronové sítě) definovány v kartézské souřadnici $[x_1,x_2]$, ale grafické souřadnice vstupů neuronu se označují obecně $[x_1,\ldots,x_n]$.

Poznámka:

Rovnice 4.23 bude dávat obrácené výsledky. V případě nerovnosti $2y+2x-3<0$ bude výstupem z neuronové sítě $y=0$ Ale my hledáme řešení $(y=1)$ v pásmu mezi přímkami $p1: 2y+2x-1>0$ a $p2: 2y+2x-3<0$. To znamená, že řešení v tomto pásmu, (přesněji v nadrovině přímky p1 a v podrovině přímky p2). K tomu abychom dosáhli na správný výsledek, musíme rovnici přímky vynásobit (-1). Z hlediska matematiky se nic nestane, pouze se obrátí směr přímky, a my dostaneme kýžený výsledek.



Obrázek 4.9: *Vysvětlení důvodu násobení (-1) nadroviny dané přímkou* $2y+2x-3<0$

5 Proces učení

Učení je proces, jímž se individuum přizpůsobuje změněným životním podmínkám. Projeví se tím, že následné chování je jiné než předchozí. Dochází ke změnám, které jsou trvalé. Mění se postoje, rozvíjí se paměť, osvojují se určité metody učení, myšlení, odpovědnosti a další vlastnosti.[1]

Proces učení neuronových sítích má podobné vlastnosti a je založen na postupném nastavení jednotlivých vah neuronů, které jsou fundamentálním prvkem pro získávání zkušeností neuronových sítí. Obecně se jedná o rekurzivní proces, při němž se neustále optimalizuje nastavení vah neuronů. Cílem učení neuronové sítě je nastavit síť tak, aby dávala co nejpřesnější výsledky.

5.1 Učení s učitelem

[text převzat z Wikipedie]

Učení s učitelem (anglicky supervised learning) je metoda strojového učení pro učení funkce z trénovacích dat.

Trénovací data sestávají ze dvojic vstupních objektů (typicky vektorů příznaků) a požadovaného výstupu. Výstup funkce může být spojitá hodnota (při regresi) anebo může předpovídat označení třídy vstupního objektu (při klasifikaci). Úloha algoritmu učení je předpovídat výstupní hodnotu funkce pro

[1] Vlci na Aljašce loví jesetery, ale žerou jim jen mozky. Těla si nevšímají. Z pohledu nezasvěceného to může vypadat jako enormní plýtvání v potravním řetězci. Skutečnost je však taková, že si vzájemně předávají zkušenost z generace na generaci, že jedině mozek jesetera není infikován parazity, kteří by mohli vlka zahubit. Nezbývá, než se opět hluboce poklonit moudré matce Přírodě.

každý platný vstupní objekt poté, co zpracuje trénovací příklady (tj. dvojice vstup a požadovaný výstup). Aby to dokázal, musí algoritmus zobecnit prezentovaná data na nové situace (vstupy) "smysluplným" způsobem (viz induktivní bias). (Porovnejte s učením bez učitele). Analogická úloha v lidské a zvířecí psychologii se často nazývá učení konceptů.

Přeučení (overfitting) je stav, kdy je systém příliš přizpůsoben množině trénovacích dat, ale nemá schopnost generalizace a selhává na testovací (validační) množině dat. To se může stát např. při malém rozsahu trénovací množiny nebo pokud je systém příliš komplexní (např. příliš mnoho skrytých neuronů v neuronové síti. Řešením je zvětšení trénovací množiny, snížení složitosti systému nebo různé techniky regularizace [2] jako je zavedení náhodného šumu (což v zásadě odpovídá rozšíření trénovací množiny), zavedení omezení na parametry systému, které v důsledku snižuje složitost popisu naučené funkce, nebo předčasné ukončení (průběžné testování na validační množině a konec učení ve chvíli, kdy se chyba na této množině dostane do svého minima).

Při učení se používají trénovací data (nebo trénovací množina), testovací data a často validační data. Tyto množiny mají být disjunktní.

Pomocí trénovacích dat se naučí jeden nebo několik klasifikátorů, případně podle chování na validačních datech se z nich vybere nebo zkombinuje výsledný klasifikátor a jeho chování na testovací množině určuje, spíš odhaduje/aproximuje, celkovou úspěšnost přístupu a chování na nových, neznámých datech.

5.2 Učení bez učitele

[text převzat z Wikipedie]

[2] regularizace funkcí - například regularizace integrálem ve smyslu hlavní hodnoty (value principal)

Učení bez učitele (anglicky unsupervised learning) je jeden ze základních typů strojového učení. Na rozdíl od učení s učitelem algoritmy učení bez učitele nemají na vstupu data provázaná s cílovou proměnnou (labelem, targetem, závisle proměnnou...). Jejich cílem je vytvořit vhodnou a zpravidla jednodušší reprezentaci vstupních dat. Učení bez učitele si tedy lze představit jako kompresi vstupních dat: například snížení jejich dimenze (v analýze hlavních komponent a podobných metodách), jejich vyhlazení (odhad distribučních funkcí, na jejichž základě data vznikla) nebo jejich redukci na konečný počet diskrétních bodů (jako je tomu ve shlukové analýze, kde vstupní data reprezentujeme označeními shluků).

Praktické aplikace učení bez učitele zahrnují například:

- klasifikaci: obchodník tak může rozdělit svoje zákazníky podle jejich podobnosti do tržních segmentů anebo archeolog může podle charakteristik nalezených střepů keramiky definovat různé kulturní okruhy, k nimž patřili lidé, kteří kdysi keramiku vyráběli
- hledání anomálií: netypické datové body mohou signalizovat poruchy nebo jiné situace, na které je potřeba se zaměřit, například při detekci podvodů ve finančních a telekomunikačních firmách
- odhad latentních proměnných: psycholog může z řady výsledků jednotlivých testů stanovit inteligenci zkoumané osoby, politolog může zkoumat na základě dotazníkového šetření rekonstruovat základní dimenze politického systému v zemi

Typické algoritmy či třídy algoritmů učení bez učitele:

- shluková analýza (hierarchické shlukování, k-means a jiné)
- metody identifikující struktury kovariančních vazeb (analýza hlavních komponent, faktorová analýza, strukturní modelování a další)
- samoorganizující neuronové sítě a některé typy neuronových sítí v oblasti hlubokého učení (automatické generování textů...)

- vyhlazování a odhadování hustot pravděpodobnosti, například kernel density estimation

5.3 Příklad: proces učení XOR

Hluboké dopředné sítě, nazývané také dopředné neuronové sítě, nebo vícevrstvé perceptrony (MultiLayers Perceptrons - MLP), jsou zásadní modely hlubokého učení. Cílem dopředné sítě je přiblížit nějakou funkci f požadovanému tvaru výstupní funkce. Například, pro klasifikátor $y = f(x)$ mapuje vstup $[x]$ na výstupní funkci $[y]$. Dopředná síť definuje proces mapování $y = f(x; \Theta)$. Při procesu mapování se hledá optimální hodnota parametrů Θ. Výsledkem je co nejlepší aproximace požadované výstupní funkce.

Učení probíhá tak, že perceptronu je předložena množina příkladů se správným chováním. Proces učení následně nastaví váhy a práh tak, aby došlo k přiblížení výstupu sítě cíli. Rozhodovacích hranic může být nekonečný počet. Učení je tedy *rekurzivní proces* úpravy vah w z dané sady vstupně-výstupních vzorů. U jediného perceptronu je cílem procesu učení najít rozhodovací rovinu, která odděluje dvě třídy daných výcvikových vektorů vstup-výstup. Po dokončení učení bude každý vstupní vektor klasifikován do příslušné třídy. Jediný perceptron může klasifikovat pouze lineárně oddělitelné vzory.

V režimu učení je požadovaná funkce sítě perceptronů dána tréninkovou sadou:

$$T = \left\{ (\mathbf{x}_k, \mathbf{d}_k) \,\middle|\, \begin{array}{l} \mathbf{x}_k = (x_{k1}, \ldots, x_{kn} \in \mathbb{R}^n) \\ \mathbf{d}_k = \left(d_{k1}, \ldots, d_{kn} \in [0,1]^m\right) \end{array} \quad k = 1, \ldots, p \right\} \tag{5.1}$$

kde $\mathbf{x}_k$ je skutečný k-tý vstup tréninkového vzoru a d_k je odpovídající požadovaný binární výstup (daný učitelem). Cílem procesu učení je najít konfiguraci vah w takovou, aby pro každý vstup $x_k, (k = 1, \ldots, p)$ z tréninkové sady $\mathbf{T}$, síť reagovala požadovaným výstupem $\mathbf{d}_k$.

Cílem procesu učení je získat co nejlepší výsledek, což lze zapsat následovně:

$$\mathbf{y}\,(w, \mathbf{x}_k) = \mathbf{d}_k \quad k = 1, \ldots, p; \quad w = (w_{10}, \ldots, w_{mn}) \tag{5.2}$$

Předpokládejme, že netrénovaný perceptron bude generovat nesprávný výstup $y \neq d_k$ kvůli nesprávným hodnotám vah. O tomto výstupu můžeme uvažovat jen jako o odhadu požadovaného výstupu d_k. Rekurzivním procesem učení však dochází ke zpřesňování hodnot vah. Tím dochází ke zpřesňování počátečního odhadu.

Princip:

1. Na začátku procesu učení, v čase $t = 0$, jsou váhy konfigurace $w(0)$ inicializovány náhodně, hodnotami blízko nuly.

   $$w_{i,j}^{(0)} = rand(x) \in \langle -1, +1 \rangle \tag{5.3}$$

2. Síť perceptronů má diskrétní adaptivní dynamiku. V každém adaptačním časovém kroku $t = 1, 2, 3, \ldots$ je jeden vzor z tréninkové sady představen síti, která se jej pokusí naučit. Skutečný výstup y_j se porovná s požadovaným výstupem d_{kj}

   $$y_j = \sigma\left(w_{i,j}^{(t)}.\mathbf{x}_i\right) \tag{5.4}$$

   přičemž chyba e_{kj} se vypočítá následovně:

   $$\varepsilon_{kj} = d_{kj} - \sigma_j\left(w_{i,j}^{(t)}.x_i\right) \quad \text{nebo} \quad \varepsilon_{kj} = \sigma_{kj} - y_j \tag{5.5}$$

jestliže:

$$d_{kj}, y_{kj} \in \{0, 1\} \tag{5.6}$$

pak:

$$\varepsilon_{kj} \in \{-1, 0, 1\} \tag{5.7}$$

3. V každém kroku se váhy upraví následovně:

$$w_{ij}^{t+1} = w_{ij}^{t} + \eta \varepsilon_{kj} \mathbf{x}_i \tag{5.8}$$

kde $\eta \in \langle 0, 1 \rangle$ je rychlost učení (zisk), který řídí rychlost adaptace.

Příliš velká rychlost režimu učení může narušit předchozí proces. Proto přidáme zlomek do váhového vektoru, abychom vytvořili nový váhový vektor s jemnějším dělením. Rychlost režimu učení musí být upravena tak, aby byla zajištěna konvergence algoritmu.

Výraz pro úpravu vah lze přepsat následovně:

$$w_{ij}^{t+1} = w_{ij}^{t} + \eta \mathbf{x}_i \left(d_{kj} - y_j(w^t, \mathbf{x}_k) \right) \tag{5.9}$$

Výraz $\left(d_{kj} - y_j(w^t, \mathbf{x}_k) \right)$ v rovnici 5.9 je rozdíl mezi aktuálním *j-tým* výstupem pro *k-tý* vstup vzoru a odpovídající požadovaný výstup tohoto vzoru. To tedy určuje chybu *j-tého* síťového výstupu s ohledem na *k-tý* tréninkový vzor. Je zřejmé, že pokud je tato chyba nula, základní váhy se nezmění. Jinak může být tato chyba buď 1 nebo -1, protože jsou brány v úvahu pouze binární výstupy.

Rosenblatt, ukázal, že adaptivní dynamika popsaná rovnicí 5.9 zaručuje, že síť najde svou konfiguraci v adaptivním režimu (za předpokladu, že existuje) po konečném počtu kroků, pro které správně klasifikuje všechny tréninkové vzorce. Chyba sítě vzhledem k tréninkové sadě je nula. Tím je splněna pod-

mínka rovnice 5.9.

Pořadí vzorců během procesu učení je předepsáno takzvanou strategií výcviku a může být například uspořádáno na analogii lidského učení. Jeden student několikrát čte učebnici, aby se připravil na zkoušku, další se při prvním čtení naučí všechno správně, případně na konci oba revidují části, které nejsou správně zodpovězeny. Adaptace sítě perceptronů se obvykle provádí v takzvané tréninkové epoše, ve které jsou systematicky prezentovány všechny vzorce z tréninkové sady (některé dokonce několikrát).

Například v čase $t = (x-l)p+k; \quad k \in \langle 1,p \rangle$, odpovídající *x-té* tréninkové epoše, se neuronová síť učí *k-tý* tréninkový vzor.

Ještě jednou příklad XOR

XOR je reprezentován cílovou funkcí $y = f(x)$. Našim úkolem je tuto funkci naučit jednoduchý (dvouvrstvý) perceptron. Cílový model je popsán funkcí $y = f(x, w_i, b)$ a náš učící algoritmus musí přizpůsobit parametry w_i, b tak, aby y se co nejvíce podobala cílové funkci $\overline{y} = f(x)$ [8].

V tomto jednoduchém příkladu se nebudeme zabývat statistickým zobecněním. Chceme, aby naše síť fungovala správně ve čtyřech bodech:

$$\mathbb{X} = [0,0], [0,1], [1,0] a [1,1]$$

Pokusme se natrénovat síť ve všech čtyřech těchto bodech.

S aproximací funkce $f(x, w_i, b)$ lze zacházet jako s regresním problémem a použít funkci střední kvadratické chyby (MSE - mean squared error) kterou budeme značit $J(y)$.

Střední kvadratická chyba je veličina vyjadřující přesnost odhadů pomocí střední hodnoty druhých mocnin rozdílů mezi odhadem či měřením a skuteč-

ností. V praxi se pod tímto jménem používají dvě různé blízce příbuzné veličiny, a sice buď střední hodnota čtverce chyby (tedy rozptyl chyb), nebo odmocnina této střední hodnoty (tedy směrodatná odchylka chyb). Kvalita učení neuronové sítě je hodnocena nejčastěji nákladovou funkcí:

$$\mathrm{J}(y) \stackrel{\mathrm{def}}{=} \mathbb{E}\left[(y-\overline{y})^2\right] \tag{5.10}$$

kde:

J(y)	–	střední kvadratická chyba
y	–	reálný (naučený) průběh výstupní funkce
$\overline{y}$	–	naučený průběh výstupní funkce

Dosazením odhadované funkce: $y = f(x)$ a požadované funkce: $y(xor) = f(x, w_i, b)$ dostaneme:

$$\mathrm{J}(y) = \mathbb{E}\left(f*(x) - f(x, w_i, b)\right)^2 \tag{5.11}$$

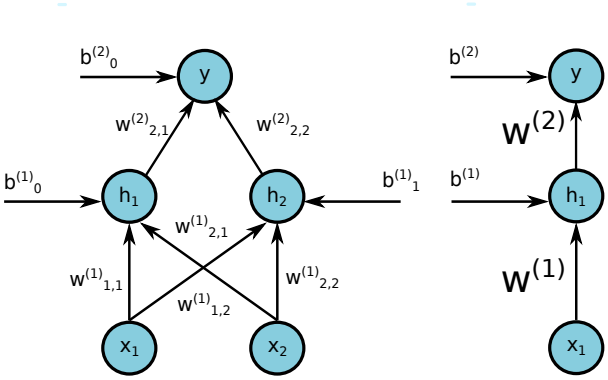
Připomeňme, že vstupní vektor $\mathbb{X}$ nabývá $[0,0]; [0,1]; [1,0]; [1,1]$.

Nyní zvolme tvar modelované funkce, $f(x, w_i, b))$. Předpokládejme, že jsme si vybrali lineární model, který je funkčně závislý na vektoru vah $\mathbf{w}$ a konstantě b. Náš model je definován rovnicí přímky.

$$f\left(\mathbf{x}, w, b\right) = \mathbf{x}^T w + b \tag{5.12}$$

Minimalizací rovnice 5.11 (řešení normálových rovnic) dostaneme $w = 0$ a $b = 1/2$, což je konstanta, která platí v celém zkoumaném prostoru. Obrázky 4.7 a 5.2 ukazují, že lineární model (jehož minimalizací je konstanta) není schopen reprezentovat funkci XOR. Jedním ze způsobů, jak vyřešit tento problém, je použít model, který se naučí jiný prostor funkcí, ve kterém je lineární model schopen reprezentovat řešení.

K řešení problému zavedeme jednoduchou dopřednou síť (viz obr.5.1) s jednou skrytou vrstvou obsahující dva skryté neurony. Tato dopředná síť má



Obrázek 5.1: *Graf neuronové sítě pro logickou funkci XOR*

vektor vstupních neuronů $[x_1, x_2]$, vektor skrytých neuronů $\mathbf{h}$, pro které platí funkce $f^{(1)}(\mathbf{x}; w_1, b_1)$.

Výstupní hodnoty těchto skrytých neuronů se následně použijí jako vstup pro druhou vrstvu. Druhá vrstva je výstupní vrstva sítě. Výstupní vrstva představuje druhou vrstvu, jejímiž vstupy jsou výstupy ze skryté vrstvy h. Síť tedy je popsána dvěma funkcemi, které jsou zřetězeny.

Funkce skryté vrstvy:

$$\mathbf{h} = f^{(1)}(\mathbf{x}; w_1, b_1) \tag{5.13}$$

Funkce výstupní vrstvy:

$$\mathbf{y} = f^{(2)}(\mathbf{h}; w_2, b_2) \tag{5.14}$$

Matematický model celé neuronové sítě tedy můžeme zapsat jako:

$$\mathbf{y} = f\left(\mathbf{x}; w_1, b_1, w_2, b_2\right) = f^{(1)}\left(f^{(2)}(\mathbf{x})\right) \tag{5.15}$$

Jakou funkci má vypočítat $f^{(1)}$? Lineární modely nám zatím dobře sloužily a může být lákavé udělat $f^{(1)}$ také lineárním. Bohužel, pokud by $f^{(1)}$ bylo lineární, pak by dopředná síť jako celek zůstala lineární funkcí vstupu.

$$f^{(1)}(\mathbf{x}) = w_1^{\top}\mathbf{x}$$

a zároveň

$$f^{(2)}(\mathbf{h}) = \mathbf{h}^\top w_2;$$

pak:

$$f(\mathbf{x}) = \mathbf{x}^\top w_1 w_2$$

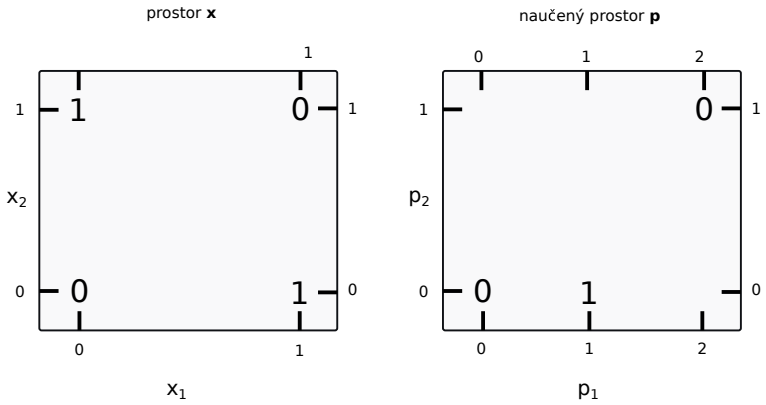
Tuto funkci lze zjednodušit:

$$f(\mathbf{x}) = \mathbf{x}^\top w' \quad \text{kde} \quad w' = w_1 w_2;$$

Je zřejmé, že k řešení problému musíme použít *nelineární* funkci. Většina neuronových sítí tak činí pomocí afinní transformace, která je řízena naučenými parametry. Výstup každého neuronu, je násoben nelineární aktivační funkcí. Jejím úkolem je vnést do lineárního řešení nelinearitu.

$$y = \sigma\left(w^T \mathbf{x} + b\right)$$

Pro aplikaci logické funkce XOR je vhodné použití funkce $ReLU$ - (Rectified Linear Unit) která je definována takto: $\sigma(x) = max\{0, x\}$. Viz obrázek 5.4.



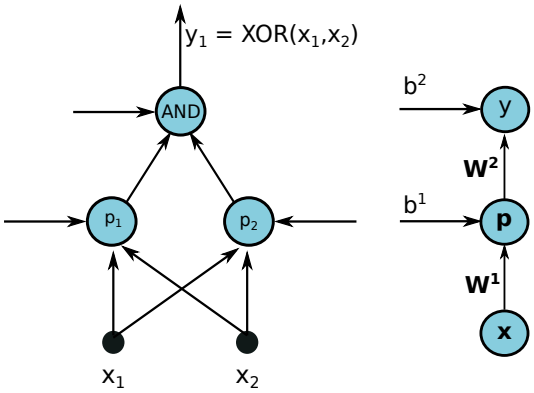
Obrázek 5.2: *Znázornění procesu učení*

Na obrázku 5.2 je znázorněn princip učení problému XOR. Tučná čísla vytištěná na grafu označují hodnotu, kterou musí naučená funkce dávat v každém

bodě. Lineární model (znázorněný v levé části) aplikovaný přímo na původní vstup nemůže implementovat funkci XOR.

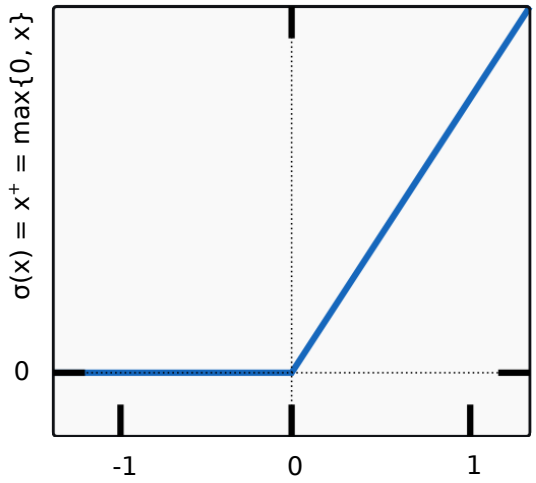
Jestliže $x_1 = 0$, pak výstup modelu musí růst s růstem x. Jestliže $x_1 = 1$, výstup modelu musí klesat, v souvislosti s růstem hodnoty x_2. Lineární model musí použít fixní koeficient w_2 až x. Hodnotu x_1 nemůže použít ke změně koeficientu na x_2 a tudíž nemůže tento problém vyřešit, (znázorněno vlevo).

V transformovaném prostoru představovaném prvky extrahovanými neuronovou sítí může nyní problém vyřešit lineární model. V našem příkladu existují dva body $x = [1,0]^T$, a $x = [0,1]^T$, jejichž výstupem musí být 1. Našim úkolem je najít funkci, která by oba body transformovala do jediného a tak umožnila řešení $y = \mathrm{XOR}(\mathbf{x})$. Jinými slovy, nelineární funkce $\mathbf{p}$ mapuje body $x = [1,0]^T$, a $x = [0,1]^T$ do jediného bodu v prostoru prvků, $h = [1,0]^T$.Lineární model můžeme nyní popsat funkci, která roste v h_1 a klesá v h_2.



Obrázek 5.3: *Znázornění procesu učení*

Obrázek 5.3: ukazuje příklad dopředné sítě nakreslený dvěma různými styly. Konkrétně se jedná o dopřednou síť, kterou používáme k řešení příkladu XOR. V levé části obrázku je naznačena jedna skrytá vrstva obsahující dvě jednotky. V tomto stylu je nakreslena každá jednotka jako uzel orientovaného grafu. Obrázek vlevo je explicitní a jednoznačný. Pro rozsáhlé sítě se styl kreslení zjednodušuje, tak jak je znázorněno na pravé straně obrázku.



Obrázek 5.4: *Aktivační funkce typu "ReLU"*

Obrázek 5.4: znázorňuje aktivační funkci typu $ReLU$, jejíž průběh lze přirovnat k průběhu napětí na usměrňovači. Aktivační funkce je obecně definována jako pozitivní část jejího argumentu:

$$\sigma(x) = x^+ = max(0, x)$$

Funkce $\sigma(x)$ je výchozí aktivační funkcí, která je doporučena pro použití ve většině dopředných neuronových sítí. Působením této funkce na výstup lineární transformace se získá transformace nelineární.

Funkce $ReLU$ zůstává velmi blízká lineární, nicméně pouze v tom smyslu, že se jedná o lineární funkci se dvěma lineárními částmi. Tím zachovává mnoho vlastností, díky nimž lze lineární modely snadno optimalizovat pomocí metod založených na gradientu. Díky těmto vlastnostem se lineární modely dobře zobecňují.

Běžným principem je, že dokážeme poskládat komplikované systémy z minimalistických komponent. Vzhledem k tomu, že paměť Turingova stroje potřebuje ukládat pouze stavy $log(0)$ nebo $log(1)$, je možno pomocí $ReLU$ funkcí vytvořit univerzální aproximátor.

Nyní můžeme definovat kompletní neuronovou síť jako:

$$f(\mathbf{x}; w_1, b_1, w_2, b_2) = w_2^\top max\left\{0, w_1^\top \mathbf{x} + b_2\right\} + b_1 \tag{5.16}$$

Vzhledem k tomu, že síť perceptronů je schopná vypočítat pouze omezenou třídu funkcí, je význam tohoto modelu spíše teoretický. Kromě toho výše uvedená věta o konvergenci pro adaptivní režim nezaručuje efektivitu učení. To bylo potvrzeno časově náročnými experimenty. Schopnost zobecnění tohoto modelu také není příliš velká, protože síť perceptronů lze použít pouze ve speciálních případech. Například při rozpoznávání vzorů.

Tento jednoduchý model je však základem pro složitější konfigurace neuronových sítí, jakou je například obecná dopředná síť s algoritmem zpětného učení (backpropagation).

6 Topologie neuronových sítí

Topologie neuronové sítě popisuje způsob propojení neuronů a je důležitým faktorem pro fungování sítě a procesu učení. Typickou a nejběžnější topologií v ANN při uplatnění učení s dozorem, je plně propojená třívrstvá dopředná síť typu Dense. Všechny vstupní hodnoty do sítě jsou připojeny ke všem neuronům ve skryté vrstvě (skryté, protože nejsou viditelné na vstupu nebo výstupu), výstupy skrytých neuronů jsou připojeny ke všem neuronům ve výstupní vrstvě a aktivace výstupní neurony tvoří výstup celé sítě.

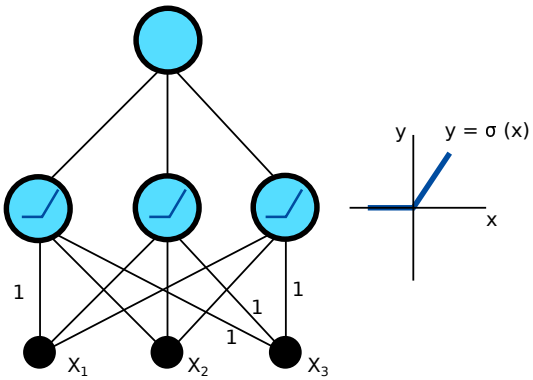
Rozšířenou topologií sítě pro učení bez dozoru je přímé mapování vstupů do kolekce jednotek, které představují kategorie (např. Kohonenova síť SOM - Samoorganizující se mapy).

Existuje mnoho typů neuronových sítí, které mohou být ve fázi vývoje. Mohou být klasifikovány v závislosti na jejich: Struktuře, toku dat, použitých neuronech a jejich hustotě, vrstvách a jejich filtrech aktivace hloubky atd. Různé typy neuronových sítí používají při určování vlastních pravidel různé principy. Existuje mnoho typů umělých neuronových sítí, z nichž každá má své jedinečné přednosti. [23]

6.1 MLP - vícevrstvý perceptron

MLP - Dopředná neuronová síť (Multilayer Perceptron) byla prvním a nejjednodušším typem umělé neuronové sítě. V této síti se informace pohybují pouze jedním směrem - dopředu - ze vstupní vrstvy přes skryté vrstvy (pokud existují), do výstupní vrstvy. V síti nejsou žádné cykly ani smyčky.[1]

Představuje klasický model neuronové sítě, která byla inspirována lidským okem. Modelovala tzv. percepci. Odtud tedy pochází pojem perceptron. Frank Rosenblatt tak simuloval rozpoznávání jednotlivých znaků v mapě 20x20 tvořených pixely. Pojem perceptron se v oblasti umělé inteligence ujal, Bez ohledu na původní význam je používán pro všechny typy dopředných neuronových sítí.



Obrázek 6.1: *Topologie sítě typu perceptron*

Topologie sítě je orientovaný graf, kde jednotlivé uzly jsou spojeny hranami. Váhy těchto hran určují šíření signálu neuronovou sítí. Signál (data) se šíří jednotlivými vrstvami postupně od vstupu přes skryté vrstvy až k výstupu. Spojení mezi neurony je orientováno. To znamená že je přesně určeno jakými cestami se signál šíří.

[1] V knihovnách TensorFlow a Keras je tato síť nazvána „Dense Layer“ a představuje základní vrstvu neuronových sítí.

6.1.1 Algoritmus Forward-Propagation

Algoritmus Forward-Propagation vypočítává a ukládá mezilehlé proměnné ve výpočetním grafu, který je definován neuronovou sítí. Postupuje od vstupní přes skryté vrstvy, do výstupní vrstvy. U každého neuronu ve skryté nebo výstupní vrstvě probíhá zpracování ve dvou krocích:

- **Preaktivace**: Je vážená suma vstupních hodnot a jejich vah.
- **Aktivace**: Vypočtená vážená suma všech vstupů vstupů se předá aktivační funkci. Aktivační funkce je matematická funkce, která dodává síti nelinearitu. Na základě výsledku vstupní sumy a aktivační funkce neuron rozhodne, zda budou tyto informace předány dále a nebo ne.

Označíme-li všechny výstupy (aktivace) jako: $\mathbf{y}_{neuron}^{(vrstva)}$, kde například $\mathbf{y}_3^2$, je aktivovaný výstup čtvrtého neuronu (počítáno od nuly) ve třetí vrstvě (opět počítáno od nuly). Jinými slovy horní index (superskript) znamená číslo vrstvy a dolní index (subskript) značí číslo neuronu ve vrstvě, korespondující se superskriptem.

Dále označme všechny vstupy (váhy) následovně: $\mathbf{w}_{od,do}^{(vrstva)}$ kde „do“ je označeno jako n a „od“ jako m Například $\mathbf{w}_{2,5}^{(1)}$ značí váhu, která je tvořena výstupem šestého neuronu, vstupující do třetího neuronu ve druhé vrstvě (opět počítáno od nuly). Pro výpočet všech výstupních hodnot v aktuální vrstvě, potřebujeme všechny výstupní hodnoty (aktivace) předchozí vrstvy. Například pro výpočet výstupů vrstvy 2 potřebujeme výstupy z vrstvy 1. To znamená že potřebujeme vektor výstupů vrstvy 1, $[y_0^0, y_1^0, \ldots, y_n^0]$. Dále potřebujeme všechny váhy (vstupy), které jsou propojeny do aktuální vrstvy.

Začněme tím, že zapíšeme jednotlivé aktivace (výstupy) a váhy předchozí vrstvy do odpovídající matice.

$$\begin{bmatrix} y_0^{(k+1)} \\ y_1^{(k+1)} \\ \vdots \\ y_m^{(k+1)} \end{bmatrix} \tag{6.1}$$

všechny váhy, které tvoří spojení se všemi neurony v následující vrstvě.

$$\begin{bmatrix} w_{10}^{(k+1)} & w_{11}^{(k+1)} & \dots & w_{1n}^{(k+1)} \\ w_{20}^{(k+1)} & w_{21}^{(k+1)} & \dots & w_{2n}^{(k+1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k0}^{(k+1)} & w_{k1}^{(k+1)} & \dots & w_{kn}^{(k+1)} \end{bmatrix} \tag{6.2}$$

a následně přidáme aktivační funkce vrstvy (k):

$$\begin{bmatrix} \sigma_0^{(k+1)}, \sigma_1^{(k+1)}, \dots, \sigma_n^{(k+1)} \end{bmatrix} \tag{6.3}$$

dostaneme výslednou maticovou rovnici jedné vrstvy neuronové sítě, kterou lze zapsat takto:

$$\begin{bmatrix} y_0^{(k+1)} \\ y_1^{(k+1)} \\ \vdots \\ y_m^{(k+1)} \end{bmatrix} = \begin{bmatrix} \sigma_{0,\dots,n}^{(k+1)} \end{bmatrix} \left[\begin{bmatrix} w_{10}^{(k+1)} & w_{11}^{(k+1)} & \dots & w_{1n}^{(k+1)} \\ w_{20}^{(k+1)} & w_{21}^{(k+1)} & \dots & w_{2n}^{(k+1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k0}^{(k+1)} & w_{k1}^{(k+1)} & \dots & w_{kn}^{(k+1)} \end{bmatrix} \begin{bmatrix} y_0^{(k)} \\ y_1^{(k)} \\ \vdots \\ y_n^{(k)} \end{bmatrix} + \begin{bmatrix} b_0^{(k+1)} \\ b_1^{(k+1)} \\ \vdots \\ b_n^{(k+1)} \end{bmatrix} \right] \tag{6.4}$$

Maticovou rovnici lze zjednodušit. Pro aktivaci neuronové sítě se vždy používá jen jedna aktivační funkce, takže lze zapsat: $\sigma_0^{(k)} = \sigma_1^{(k)} = \dots = \sigma_n^{(k)}$.

$$\begin{bmatrix} y_0^{(k+1)} \\ y_1^{(k+1)} \\ \vdots \\ y_m^{(k+1)} \end{bmatrix} = \sigma \left[\begin{bmatrix} w_{10}^{(k+1)} & w_{11}^{(k+1)} & \dots & w_{1n}^{(k+1)} \\ w_{20}^{(k+1)} & w_{21}^{(k+1)} & \dots & w_{2n}^{(k+1)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{k0}^{(k+1)} & w_{k1}^{(k+1)} & \dots & w_{kn}^{(k+1)} \end{bmatrix} \begin{bmatrix} y_0^{(k)} \\ y_1^{(k)} \\ \vdots \\ y_n^{(k)} \end{bmatrix} + \begin{bmatrix} b_0^{(k+1)} \\ b_1^{(k+1)} \\ \vdots \\ b_n^{(k+1)} \end{bmatrix} \right] \tag{6.5}$$

Což lze zapsat do kompaktního tvaru maticové rovnice, která matematicky vyjadřuje *(k)*-tou neuronovou vrstvu.

$$\mathbf{y}^{(k+1)} = \sigma\left(\mathbf{w}^{(k+1)}\mathbf{y}^{(k)} + \mathbf{b}^{(k+1)}\right), k \in [0, 1, \ldots, n] \tag{6.6}$$

Poznámka

Tyto rovnice nepopisují prakticky nic jiného nežli to, že vezmeme všechny výstupy z předchozí vrstvy, ty vzájemně vynásobíme se vstupními vahami aktuální vrstvy, přidáme matici biasů a nakonec ještě všechno vynásobíme s aktivační funkcí. Tím získáme výstupní hodnoty aktuální vrstvy, které následně pošleme na vstupy vrstvy následující. Výše popsaný postup opakujeme, dokud se nedostaneme k vrstvě výstupní.

6.1.2 Optimalizace procesu Forward-Propagation

Predikované hodnoty, které jsou skrytými aktivacemi, závisí na třech parametrech

- Vstupy
- Aktivační funkce
- Váhy

Vstupy jsou závisle proměnné. Ty samozřejmě optimalizovat nelze

Aktivační funkci také nelze v průběhu optimalizace měnit. Změnit aktivační funkci lze až na základě rozhodnutí, že právě probíhající trénink sítě dosáhl svého nejlepšího optima, a není možno výsledek dále zlepšovat. Rozhodnutí o změně aktivační funkce může (ale nemusí) přinést lepší výsledek optimalizačního procesu.

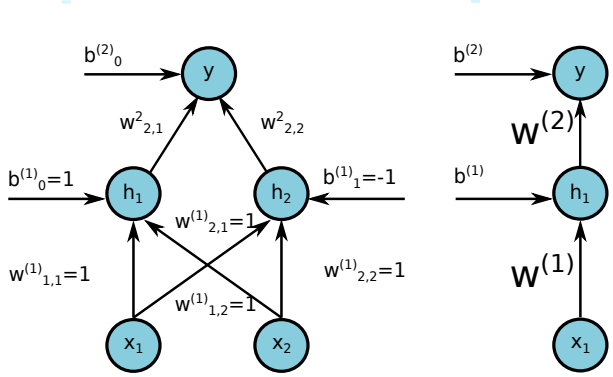
Aktivační funkce se pro celou síť volí pouze jedna. Nepoužívá se například pro každou skrytou vrstvu jiná aktivační funkce. To znamená, že iterační změna aktivační funkce nepřichází v úvahu.

Váhy jsou tím správným nástrojem pro optimalizaci výstupů neuronové sítě. Pro získání optimálního výstupu, však nelze měnit váhy libovolně a v libovolném počtu jednotlivých vah. Znamenalo by to neúměrně velkou časovou náročnost pro získání minimální chyby, mezi optimálním a námi trénovaným průběhem neuronové sítě. Nastavení vah lze provádět metodou „pokus-omyl“ a to malou změnou hodnoty váhy a dále sledováním směru této změny.

Pro optimalizaci vah neuronové sítě, se však metoda pokusů a omylů nepoužívá (snad jen pro velmi malé sítě), díky časové náročnosti a ne zrovna optimálně dosaženými výsledky. K výraznému zlepšení procesu učení přispívá metoda Backward Propagation, která optimalizuje hodnotu vah pomocí výpočtu gradientního průběhu jednotlivých vah sítě.

6.1.3 Příklad Forward-Propagation - funkce XOR.

Jako jednoduchý příklad pro demonstraci algoritmu Forward-Propagation uveďme již výše řešenou funkci XOR.[2]



Obrázek 6.2: *Graf naučené neuronové sítě pro funkci XOR*

Nechť matice vah neuronové vrstvy 1 má tyto hodnoty:

$$\mathbf{w}^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \tag{6.7}$$

[2] váhy a biasy jsou v tomto příkladu zvoleny tak, aby výpočet algoritmu proběhl v jediném kroku, na “první dobrou”. V praxi je situace mnohem komplikovanější.

a matice biasů vrstvy 1 je nastavena takto:

$$\mathbf{b}^{(1)} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \tag{6.8}$$

a následně matice vah vrstvy 2:

$$\mathbf{w}^{(2)} = \begin{bmatrix} 1 \\ -2 \end{bmatrix} \tag{6.9}$$

a matice biasů vrstvy 2 (matice o rozměru 1x1):

$$\text{a} \quad b^{(2)} = 0; \tag{6.10}$$

Nyní můžeme představit postup, jakým způsobem zpracovává model dávku vstupů. Nechť $\mathbf{x}$ je návrhová matice obsahující všechny čtyři body v binárním vstupním prostoru:

$$\mathbf{x} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \tag{6.11}$$

Násobením matice vstupů $\mathbf{x}$ maticí vah $\mathbf{w}$ dostaneme:

$$\mathbf{x} \times \mathbf{w}^{(1)} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix} \tag{6.12}$$

a přičtením vektoru biasu $\mathbf{b}^{(1)}$ dostaneme:

$$\left(\mathbf{x} \times \mathbf{w}^{(1)}\right) + \mathbf{b}^{(1)} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 0 \end{bmatrix} \tag{6.13}$$

V tomto prostoru leží všechny příklady podél linie se sklonem 1. Jak se pohybujeme po tomto řádku, výstup musí začínat na 0, poté růst na 1, pak klesat zpět na 0. Lineární model nemůže takovou funkci implementovat.

Pro dokončení výpočtu hodnoty h pro každý vstup použijeme usměrněnou lineární transformaci, to znamená, že tento výsledek vynásobíme váhovým vektorem w:

$$y = \left(\left(\left(\mathbf{x} \times \mathbf{w}^{(1)}\right) + b^{(1)}\right) \times \mathbf{w}^{(2)}\right) + b^{(2)} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} \tag{6.14}$$

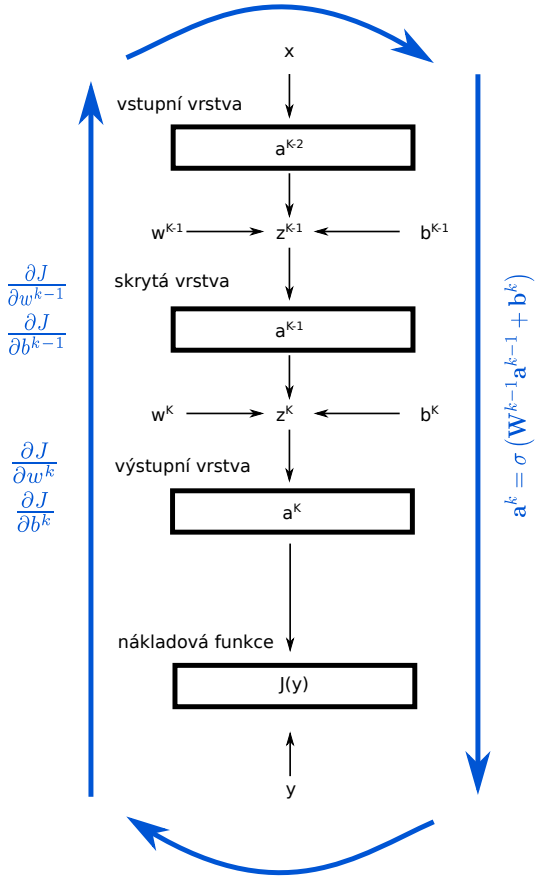
A toto je výsledné řešení funkce **XOR** metodou Forward-Propagation.

6.1.4 Algoritmus Back-Propagation

Algoritmus Back-Propagation je gradientní metoda učení umělé neuronové sítě, založená na principu vyhledávání minima průběhu funkce. Používá se u vícevrstvých sítí při učení s učitelem. Metoda vyžaduje, aby výstup sítě byl dopředu známý. U této metody se uplatňuje minimalizace metodou gradientního sestupu.

V předchozím byla popsána dopředná neuronová síť, která při průchodu informace ze vstupu směrem k výstupu šíří informaci do všech skrytých neuronových vrstev do té doby, dokud nezískáme na výstupu výsledek. Tato metoda se nazývá **Forward-Propagation** (algoritmus dopředného šíření informace).

algoritmus Back-Propagation lze shrnout do těchto kroků.



Obrázek 6.3: *Schema algoritmu Back-Propagation*

- Aplikují se vstupní data a postupně se směrem vpřed napočítá výstup (vstupní signál se sítí šíří směrem dopředu) - metoda Forward-Propagation.
- Výstup ze sítě se porovná s požadovaným výstupem, tj. spočte se chyba, pomocí chybové funkce 4.7.
- Počítá se gradient chybové funkce, to znamená jak vzniklá odchylka od požadované funkční závislosti, závisí na vahách neuronů. Výpočet postupuje zpětně, od výstupní až po vstupní vrstvu. Pro výpočet parciálních derivací (gradientu) v následujících vrstvách využívá řetězové pravidlo. Změna jednotlivých vah se mění dle jejich vlivu na chybu. Gradientní metoda poskytuje mnohem lepší výsledky nežli metoda pokusů a omylů, která je zmíněna u algoritmu Forward-Propagation.

Ve strojovém učení je Back-Propagation velmi často používaným algoritmem pro výcvik dopředných neuronových sítí. Zobecnění algoritmu Back-Propagation existuje i pro jiné umělé neuronové sítě (ANNs) a pro funkce obecně. Při učení neuronové sítě se vypočítává zpětné šíření gradientní funkce ztráty s ohledem na váhy sítě pro jeden uzel vstupu a výstupu. Tato metoda je efektivní, na rozdíl od naivního přímého výpočtu gradientu s ohledem na každou váhu zvlášť. Tato účinnost umožňuje provádět gradientní metody pro trénování vícevrstvých sítí a aktualizovat váhy tak, aby se minimalizovaly ztráty. Běžně se používá gradientní minimalizace nebo varianty stochastické gradientní minimalizace. Algoritmus Back-Propagation funguje tak, že spočítá gradient funkce ztráty s ohledem na každou váhu řetězovým pravidlem, vypočítá gradient po jedné vrstvě a iteruje zpětně od poslední vrstvy, aby se zabránilo nadbytečným výpočtům mezilehlých výrazů.

Termín Back-Propagation a jeho obecné použití v neuronových sítích byl popsán v článku [**Rumelhart1986a**]. Tato technika však byla mnohokrát znovuobjevena samostatně a měla mnoho předchůdců v šedesátých letech. Moderní pohled na problematiku poskytuje [8]

Algoritmus Back-Propagation není chápán pouze jako metoda specifická pro vícevrstvé neuronové sítě, ale jako metoda, která dokáže vypočítat gradienty jakékoliv funkce. A to i v případě, že v některých případech gradient neexistuje, je výsledkem hlášení o neexistenci derivace této funkce. V učících se algoritmech je gradient počítán z aktivačních funkcí s ohledem na parametry optimalizace funkce J(y).

Během tréninku pokračuje algoritmus procesu učení do té doby, dokud nevytvoří výslednou optimální funkci $J(y)$. Jinými slovy, Forward-Propagation (neboli dopředný průchod) se týká výpočtu a ukládání mezilehlých proměnných (včetně výstupů) pro neuronovou síť v pořadí od vstupní vrstvy k výstupní a následný Backpropagation zpětně optimalizuje průběh požadované funkční závislosti neuronové sítě.

Architektura sítě vyžaduje určení její hloubky, šířky a aktivačních funkcí použitých v každé vrstvě. Hloubka je počet skrytých vrstev a šířka je počet jednotek (neuronů) v každé skryté vrstvě.

Algoritmus **Back-Propagation** zpětného šíření informace, často nazývaný *backprop*, funguje tak, že informace z aktivačních funkcí se šíří dopřednou sítí zpětně. To znamená že na rozdíl od metody Forward-Propagation, se vytváří síť zpětných vazeb. Snahou této strategie je minimalizovat gradient.

6.1.5 Výpočet gradientu

Metoda Back-Propagation používá dvě strategie, které tvoří fundamentální myšlenku vlastního výpočtu.

- Algoritmus zpětného šíření: Back-Propagation, který je založen na myšlence minimalizace gradientního průběhu požadované výstupní funkce
- Optimalizátory - nástroje pro trénování sítě pomocí vypočtených gradientů

Výpočet začíná vždy od výstupní vrstvy a šíří se zpětným chodem k vrstvě vstupní, přičemž v každém kroku jsou aktualizovány váhy a biasy pro každou vrstvu.

Myšlenka je jednoduchá: upravit váhy a biasy v celé síti tak, abychom ve výstupní vrstvě dostali požadovaný výstup. Řekněme že chceme, aby výstupní neuron byl ve stavu 1. Pak musíme posunout váhy a bias tak, abychom dostali výstup co nejblíže k 1.

Můžeme však měnit jen váhy a biasy. Aktivační funkce však musí zůstat pro celou neuronovou síť nezměněna. Změnou aktivační funkce v průběhu optimalizace bychom proces výrazně zkomplikovali.

Pro definici následujících vztahů, zopakuji pravidla značení

k	–	vrstva neuronů
k-1	–	předchozí vrstva neuronů atd...
l	–	vrstva
y	–	Výstupní hodnota - predikovaná
$\overline{y}$	–	Výstupní hodnota - požadovaná
C	–	Nákladová funkce
w	–	Váha
a	–	Aktivace
b	–	Bias $w_0 = b$
z	–	Součin vah vrstvy k a aktivací vrstvy k-1 plus bias $w_0 = b$

Způsob, jak vypočítat přechody v algoritmu Back-Propagation, je promyslet tuto otázku:

Jak lze změřit změnu nákladové funkce ve vztahu ke konkrétní váze $\mathbf{w}$, biasu $\mathbf{w_0} = \mathbf{b}$ nebo aktivaci $\mathbf{a}$?

Z výše uvedeného vyplývá, že v průběhu optimalizace nelze měnit aktivační funkci, proto nám zbývají jediné dvě hodnoty, které mohou být zahrnuty do optimalizačního algoritmu.

změna váhy $\mathbf{w}$
změna biasu $\mathbf{b_0}$
změna aktivační funkce σ

Pro pochopení výpočtu gradientu je třeba mít ujasněno co je to derivace a jaké má vlastnosti, protože nám umožní optimalizovat vztah mezi skutečným (naučeným) výstupem neuronové sítě a výstupem požadovaným, a to minimalizací nákladové funkce. Optimalizaci lze tedy provést změnou příslušných vah a biasů.

Principem je rekurzivní iterace, při níž se neustále porovnává výstup z nákladové funkce. Principem je získat minimální odchylku od optimálního, tedy požadovaného průběhu výstupní funkce od funkce, která je výsledkem zvolené-

ho učícího se algoritmu neuronové sítě, v našem případě algoritmu Back-Propagation.

Nejprve začneme definicí příslušných rovnic. Pro každou každou vrstvu neuronu lze zapsat základní rovnici neuronu v této zjednodušené podobě.

$$a^{(k)} = \sigma(\sum_{i=1}^{n} w_i^{(k)} a_i^{(k-1)} + b^{(k)}) \tag{6.15}$$

pro další zjednodušení položme:

$$z^k = \sum_{i=1}^{n} w_i^{(k)} a_i^{(k-1)} + b^{(k)} \tag{6.16}$$

takže:

$$a^{(k)} = \sigma\left(z^{(k)}\right) \tag{6.17}$$

Nákladovou funkci napíšeme následovně:

$$J = (a^{(k)} - \overline{y})^2 \tag{6.18}$$

Našim úkolem je nalézt extrémy (minima) nákladové funkce J požadovaném intervalu. Pro nalezení extrému funkce je nutné nákladovou funkci parciálně derivovat, dle jednotlivých proměnných. To znamená, že se snažíme najít minimum funkce J v závislosti na vstupních vahách w, biasech b a aktivačních funkcích σ. Tím dostaneme tři rovnice, kde každá představuje minimalizační funkci dle konkrétní proměnné.

Minimalizace J podle w

$$\frac{\partial J}{\partial w^{(k)}} = \frac{\partial (a^{(k)} - \overline{y})^2}{\partial w^{(k)}} = \frac{\partial J}{\partial a^{(k)}} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial w^{(k)}} \tag{6.19}$$

Výsledek:

$$\frac{\partial J}{\partial w^{(k)}} = 2\left(a^{(k)} - \overline{y})\right) \sigma^{'}\left(z^{(k)}\right) a^{k-1} \tag{6.20}$$

Minimalizace J podle b:

$$\frac{\partial J}{\partial b^{(k)}} = \frac{\partial (a^{(k)} - \overline{y})^2}{\partial b^{(k)}} = \frac{\partial J}{\partial a^{(k)}} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial b^{(k)}} \tag{6.21}$$

Výsledek:

$$\frac{\partial J}{\partial b^{(k)}} = 2\left(a^{(k)} - \overline{y})\right) \sigma^{'}\left(z^{(k)}\right) \tag{6.22}$$

Minimalizace J podle a:

$$\frac{\partial J}{\partial a^{(k-1)}} = \frac{\partial (a^{(k)} - \overline{y})^2}{\partial w^{(k)}} = \frac{\partial J}{\partial a^{(k)}} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial a^{(k-1)}} \tag{6.23}$$

Výsledek:

$$\frac{\partial J}{\partial a^{(k-1)}} = 2\left(a^{(k)} - \overline{y})\right) \sigma^{'}\left(z^{(k)}\right) w^{k} \tag{6.24}$$

přičemž postup řešení je znázorněn v dodatku Hledání extrémů funkce výpočtem gradientu.

Tyto rovnice pouze měří poměr toho, jak konkrétní váha ovlivňuje nákladovou funkci, kterou chceme optimalizovat. Optimalizace probíhá krokováním těchto rovnic ve směru od výstupu ke vstupu neuronové sítě.

Každá parciální derivace vah a biasů tvoří gradientní vektor, jehož dimenze je dána počtem vah a biasů:

$$-\nabla C\left(w_1, b_1, \ldots, w_n, b_n\right) = \left[\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial b_1}, \ldots, \frac{\partial C}{\partial w_n}, \frac{\partial C}{\partial b_n}\right] \tag{6.25}$$

Při procesu učení je také dobré sledovat chování aktivačních funkcí. Ty umožní náhled, jak síť reaguje na změny.

Neukládáme je však do vektoru přechodu. Důležité je, že nám také pomohou zjistit, které váhy jsou dominantní.

$$w^{(l)} = w^{(l)} - \eta \times \frac{\partial C}{\partial w^{(l)}} \tag{6.26}$$

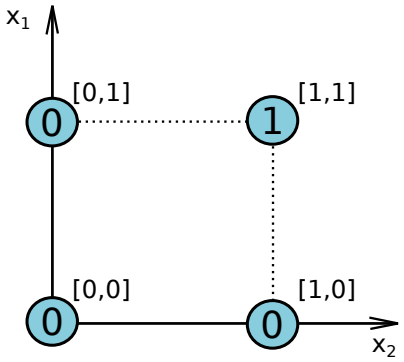
$$b^{(l)} = b^{(l)} - \eta \times \frac{\partial C}{\partial b^{(l)}} \tag{6.27}$$

Kde η je rychlost učení.

6.1.6 Příklad Back-Propagation - funkce AND

Příklad algoritmu Back-Propagation vysvětlíme na dobře známé logické funkci AND, která má tuto pravdivostní tabulku.

x_1	x_2	y
0	0	0
0	1	0
1	0	0
1	1	1



Obrázek 6.4: *Pravdivostní tabulka logické funkce AND*

Neuronová síť, má dva vstupy, které dávají čtyři kombinace vstupních hodnot. Pro trénink neuronové sítě vystačíme jen se dvěma vstupy.

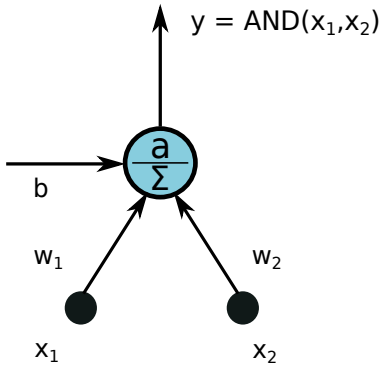
$$\begin{aligned} x_1 &= [1,1] \\ x_2 &= [0,1] \end{aligned}$$

Pro první příklad tréninku je požadovaný výstup neuronové sítě 1 a pro druhý příkladu tréninku je výstupem 0. Naše neuronová síť má 2 neurony ve vstupní vrstvě a 1 neuron ve výstupní vrstvě:

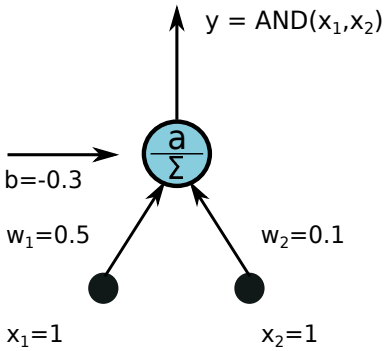
Proces učení metodou back-propagation lze rozložit do dvou částí.

První krok, Forward Propagation

V prvém kroku použijeme algoritmu Forward-Propagation, který spustíme ve dvou fázích, a to pro vstupní hodnoty $[1,1]$ a pak pro $[0,1]$. Nejprve náhodně inicializujeme váhy a zkreslení:

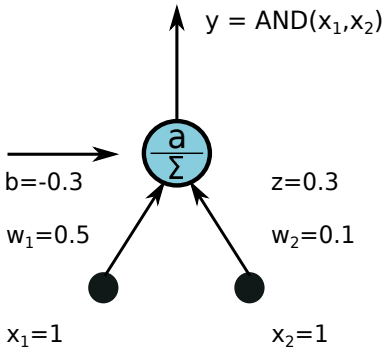


Obrázek 6.5: *Schema neuronové sítě pro logickou funkci AND*



Obrázek 6.6: *Inicializace neuronové sítě pro logickou funkci AND*

Nastavíme náhodně bias na hodnotu -0.3



Obrázek 6.7: *Nastavení biasu neuronové sítě pro logickou funkci AND*

Následně spočítáme váženou sumu z, do níž vstupují výpočty vah w, vstupů x a biasu b.

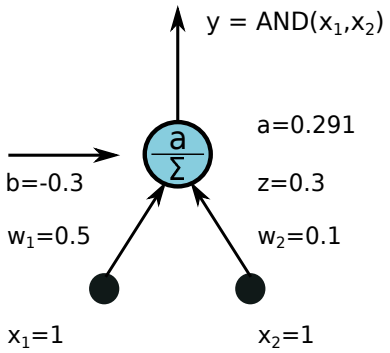
$$z = x_1 w_1 + x_2 w_2 + b$$

$$z = 1 \times 0.5 + 1 \times 0.1 - 0.3 = 0.3$$

Na výsledné z aplikujeme aktivační funkci. Zvolili jsme hyperbolický tangens.

$$y = \sigma(z) = \tanh(z)$$

$$y = \tanh(0.3) = 0.291$$

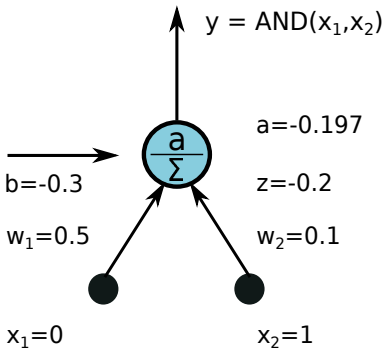


Obrázek 6.8: *Aktivace sítě pro vstup [1,1]*

Pro průchod neuronovou sítí jsme pro vstupní hodnotu $x = [1,1]$ dostali výsledek 0.291. My však požadujeme 1. To můžeme vyladit algoritmem Back-Propagation. Nejdříve však musíme udělat druhý krok, to znamená vypočítat výstupní hodnoty i pro vstup $x = [0,1]$.

V druhém kroku jsme stejným postupem získali výstupní hodnotu $y = -0.197$.

Rekurzivním postupem s využitím metody pokusů a omylů bychom mohli pomocí algoritmu Forward-Propagation získat požadovaný výsledek. V následujícím však ukážeme, že algoritmus Backward-Propagation nás dovede k cíli také, ale mnohem efektivněji.



Obrázek 6.9: *Aktivace sítě pro vstup [0,1]*

Druhý krok, Back-Propagation

Pro zjištění kvality učícího procesu neuronové sítě můžeme definovat nákladovou funkci, která určí cenu námi získaného výstupu.

$$J_k = (a^{(k)} - \overline{y})^2$$

Výsledkem nákladové funkce pro vstup $x = [1,1]$ je:

$$J_1 = (0.291 - 1)^2 = 0.502$$

a pro vstup $x = [0,1]$ je:

$$J_2 = (-0.197 - 0)^2 = 0.038$$

Celkový výsledek je počítán jako průměr ze všech dílčích výsledků nákladové funkce, to znamená:

$$J = \frac{1}{n} \sum_{k=1}^{n} C_k$$

Pro náš případ dostaneme:

$$J = \frac{0.502 + 0.038}{2} = 0.27$$

Našim úkolem je zlepšit výkon neuronové sítě. Optimalizace neuronové sítě probíhá opět rekurzivním postupem, zde ovšem nepoužíváme metody pokusů a omylů při nastavování vstupních vah a biasů, ale cíleně hledáme optimální průběh výstupní funkce minimalizací nákladové funkce, gradientní metodou při úpravách hodnot vah a biasů.

Pro výpočet použijeme řetězové pravidlo derivace funkce, které uplatníme na jednotlivé vstupy a jednotlivé biasy námi trénované neuronové sítě:

$$\frac{\partial J_k}{\partial w_1} = \frac{\partial J_k}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w_1}$$

$$\frac{\partial J_k}{\partial w_2} = \frac{\partial J_k}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial w_2}$$

$$\frac{\partial J_k}{\partial b} = \frac{\partial J_k}{\partial y}\frac{\partial y}{\partial z}\frac{\partial z}{\partial b}$$

přičemž:

$$y = \sigma(z) = \tanh(z)$$

Řetězovým pravidlem výpočtu parciálních derivací funkce pro vstup w_1 dostaneme:

$$\frac{\partial J_k}{\partial y} = \frac{\partial (a^{(k)} - \overline{y})^2}{\partial y} = 2\,(y - \overline{y})$$

$$\frac{\partial y}{\partial z} = \frac{\partial}{\partial z}\tanh(z) = \frac{1}{\cosh^2(z)}$$

$$\frac{\partial z}{\partial w_1} = \frac{\partial x_1 w_1 + x_2 w_2 + b}{\partial w_1} = x_1$$

pro vstup w_2 dostaneme:

$$\frac{\partial J_k}{\partial y} = \frac{\partial (a^{(k)} - \overline{y})^2}{\partial y} = 2\,(y - \overline{y})$$

$$\frac{\partial y}{\partial z} = \frac{\partial}{\partial z} \tanh(z) = \frac{1}{\cosh^2(z)}$$

$$\frac{\partial z}{\partial w_2} = \frac{\partial x_1 w_1 + x_2 w_2 + b}{\partial w_1} = x_2$$

a pro bias b dostaneme:

$$\frac{\partial J_k}{\partial y} = \frac{\partial (a^{(k)} - \overline{y})^2}{\partial y} = 2\,(y - \overline{y})$$

$$\frac{\partial y}{\partial z} = \frac{\partial}{\partial z} \tanh(z) = \frac{1}{\cosh^2(z)}$$

$$\frac{\partial z}{\partial b} = \frac{\partial x_1 w_1 + x_2 w_2 + b}{\partial w_1} = 1$$

Takže: Výpočet parciálních derivací pro první tréninkový příklad je:

$$\frac{\partial J_k}{\partial y} = 2\,(y - \overline{y}) = 2(0.291 - 1) = -1.418$$

$$\frac{\partial y}{\partial z} = \frac{1}{\cosh^2(z)} = 0.915$$

$$\frac{\partial z}{\partial w_1} = 1$$

$$\frac{\partial z}{\partial w_2} = 1$$

$$\frac{\partial z}{\partial b} = 1$$

takže pro první tréninkový krok platí:

$$\frac{\partial J_1}{\partial w_1} = (-1.418) \times 0.915 \times 1 = -1.297$$

$$\frac{\partial J_1}{\partial w_2} = (-1.418) \times 0.915 \times 1 = -1.297$$

$$\frac{\partial J_1}{\partial b} = (-1.418) \times 0.915 \times 1 = -1.297$$

a pro druhý tréninkový krok platí:

$$\frac{\partial J_2}{\partial w_1} = 0$$

$$\frac{\partial J_2}{\partial w_2} = -0.379$$

$$\frac{\partial J_2}{\partial b} = -0.379$$

Nyní vypočítáme parciální derivace s ohledem na celkové náklady. Parciální derivace celkových nákladů vzhledem k w_1 je průměr všech dílčích derivací jednotlivých nákladových funkcí vzhledem k w_1:

$$\frac{\partial J}{\partial w_1} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial J_k}{\partial w_1}$$

V našem tréninkovém příkladu dostaneme pro w_1:

$$\frac{\partial J}{\partial w_1} = \frac{(-1.297) + 0}{2} = -0.648$$

A stejně pro w_2 a bias:

$$\frac{\partial J}{\partial w_2} = \frac{(-1.297) + (-0.379}{2} = -0.838$$

$$\frac{\partial J}{\partial b} = \frac{(-1.297) + (-0.379}{2} = -0.838$$

Poté aktualizujeme váhy a zkreslení: vynásobíme částečné derivace tzv. rychlostí učení a odečteme výsledky od vah a zkreslení. zvolme rychlost učení $\alpha = 0.6$

Aktualizace váhy w_1:

$$w_1^+ = w_1 - \left(\alpha \times \frac{\partial J}{\partial w_1}\right)$$

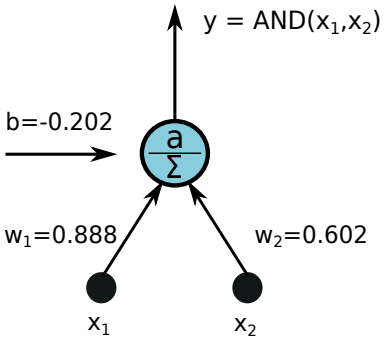
$$w_1^+ = 0.5 - (0.6 \times (-0.648)) = 0.888$$

Stejným postupem získáme novou hodnotu w_2 a biasu.

$$w_2^+ = 0.602$$

$$b^+ = 0.202$$

Po updatu vah a biasu bude naše neuronová síť vypadat následovně:



Obrázek 6.10: *Váhy a bias po první iteraci Back-Propagation pro logickou funkci AND*

Toto je výsledek první iterace algoritmu Back-Propagation. Pro další iteraci bychom mohli pokračovat tak, že po dosazení zpřesněných hodnot w_1, w_2 a b, spustíme algoritmus Forward-Propagation, spočítáme nákladovou funkci J a pak znovu použijeme algoritmus Back-Propagation. A toto následně opakovat. Proces učení neuronové sítě můžeme do jisté míry přirovnat k jistému druhu „alchymie“, avšak výsledky, které lze neuronovou sítí získat jsou naprosto přesvědčivé.

6.1.7 Síť typu Dense Layer

V neuronové síti typu Dense Layer, je každá z vrstev úplně propojena s vrstvou předchozí. To znamená, že neurony vrstvy následující jsou spojeny s každým neuronem vrstvy předchozí. Dense Layer je nejběžněji používanou vrstvou v topologiích neuronových sítí.

Každý neuron vrstvy Dense Layer je tedy propojen s výstupy všech neuronů předchozí vrstvy. Výsledkem je, že neurony ve vrstvách Dense Layer provádějí násobení maticového vektoru. Násobení maticového vektoru je procedura, kde se řádkový vektor výstupu z předchozích vrstev rovná sloupcovému vektoru husté vrstvy. Obecným pravidlem násobení matice-vektor je, že řádkový vektor musí mít tolik sloupců jako sloupcový vektor.

6.1.8 Definice sítě typu MLP (Dense) v knihovnách TensorFlow a Keras

Knihovna Keras poskytuje základní vrstvu MLP sítě pod názvem Dense Layer (hustá vrstva) prostřednictvím následující syntaxe:

```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Parametry vrstvy Dense Layer v knihovně Keras

Units: Jednotky jsou jedním z nejzákladnějších a nejnutnějších parametrů Dense Layer vrstvy Keras, která definuje velikost výstupu. Musí to být kladné celé číslo, protože představuje dimenzi výstupního vektoru.

Activation: V neuronových sítích je aktivační funkce funkcí, která se používá pro transformaci vstupních hodnot neuronů. V podstatě zavádí nelinearitu do sítí neuronových sítí tak, aby se sítě mohly naučit vztah mezi vstupními a výstupními hodnotami. Pokud v této vrstvě Keras není definována žádná aktivace, bude uvažována lineární aktivační funkce. K dispozici jsou tyto aktivační funkce.

Funkce Relu – rektifikovaná funkce aktivace lineární jednotky
Sigmoid – $sigmoid(x) = 1/(1 + exp(-x))$
Softmax – převádí vektor hodnoty na rozdělení pravděpodobnosti.
Softplus – $softplus(x) = log(exp(x) + 1)$
Softsign – $softsign(x) = x/(abs(x) + 1)$
Tanh – hyperbolický tangens
Selu – Scaled Exponencial Linear Unit (SELU).
Elu – Exponenciální lineární jednotka.
Exp – Funkce exponenciální aktivace.

use_bias: Parametr use_bias se používá pro rozhodnutí, zda chceme, aby hustá vrstva používala vektor zkreslení nebo ne. Pokud není definován, jedná se o booleovský parametr, pak je use_bias nastaven na hodnotu true.

kernel_initializer: Parametr pro inicializaci matice vah jádra. Matice vah je matice vah, které se vynásobí vstupem, aby se extrahovaly relevantní jádra funkcí.

bias_initializer: Parametr pro inicializaci vektoru zkreslení. Vektor zkreslení lze definovat jako další sady vah, které nevyžadují žádný vstup a odpovídají výstupní vrstvě. Default: 0.

Kernel regularizer: Parametr pro regularizaci matice hmotnosti jádra, pokud jsme inicializovali jakoukoli matici v kernal_initializer.

bias_regularizer: Parametr pro regularizaci vektoru zkreslení, pokud jsme inicializovali jakýkoli vektor v bias_initializer. Default: None.

activity_regularizer: Slouží k regularizaci aktivační funkce, kterou jsme definovali v aktivačním parametru. Aplikuje se na výstup vrstvy. Default: None.

kernel_constraint: Parametr se používá k aplikaci omezující funkce na matici hmotnosti jádra. Default: None.

bias_constraint: Parametr slouží k aplikaci omezující funkce na vektor zkreslení. Default: None.

Vstup: N-D tenzor s tvarem: (batch_size, ..., input_dim). Nejběžnější situací by byl 2D vstup s tvarem (batch_size, input_dim).

Výstup: N-D tenzor s tvarem: (velikost_dávky, ..., jednotky). Například pro 2D vstup s tvarem (batch_size, input_dim) bude mít výstup tvar (batch_size, jednotky).

Listing 6.1: *Inicializace vrstvy Dense Layer*

```
from Keras import layers
import numpy as np
import tensorflow as tf


model = tf.keras.models.Sequential()
```

```
model.add(tf.keras.Input(shape=(16,)))
model.add(tf.keras.layers.Dense(32, activation='relu'))
print(model.output_shape)
print(model.compute_output_signature)
.
.
.
```

6.2 CNN - Konvoluční neuronové sítě

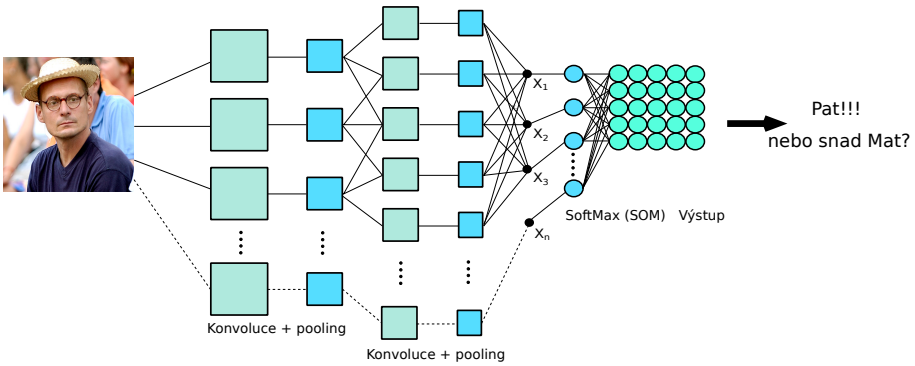
Konvoluční sítě, také známé jako konvoluční neuronové sítě neboli CNN, jsou specializovaným druhem neuronové sítě pro zpracování dat, která mají topologii podobnou mřížce. Proto je velmi vhodná pro aplikace rozpoznávání obrazových informací.

Například naměřená data v časové posloupnosti mohou být považována za 1-D matici (tenzor prvního řádu), která je vzorkována v pravidelných časových intervalech.

Obrazová data představují 2-D matici pixelů. Případ můžeme zobecnit, kdy si například místo pixelu v souřadnici matice [x,y], představíme vektor, atd... čímž dostaneme zobecnění tenzoru n-tého řádu.

Konvoluční sítě jsou v aplikacích, kde data jsou obecně popsána jako tenzor n-tého řádu, nesmírně úspěšné.

Název „konvoluční neuronová síť“ naznačuje, že síť využívá matematickou operaci zvanou konvoluce. Konvoluce je specializovaný druh lineární operace, která zpracovává dvě funkce jejímž výsledkem je odezva jedné funkce v závislosti na funkci druhé. Konvoluce má zásadní význam v oblasti teorie signálů.



Obrázek 6.11: *Topologie sítě typu CNN*

Konvoluční neuronové sítě se s úspěchem nasazují v oblastech rozsáhlých dat, kde nasazení klasické neuronové sítě by představovalo technicky nerealizovatelný problém. Například pro zpracování obrázku o velikosti 28x28x3 pixelů bychom potřebovali 2352 neuronů. To nepředstavuje příliš velký technický pro-

blém. Pokud bychom však chtěli manipulovat s obrázkem 2000x2000x3 pixelů, bude klasická neuronová síť velmi těžkopádná a při překročení jistých dimenzí obrázku, technicky nerealizovatelná.

Nasazení konvolučních sítí umožňuje úspěšně pracovat i s velmi rozsáhlými datovými strukturami, díky vlastnostem matematického operátoru konvoluce.

6.2.1 Konvoluce

Konvoluce je matematická operace mezi dvěma funkcemi $x_1(t)$ a $x_2(t)$ téhož argumentu definovaný v případě spojitých funkcí integrálem

$$x(t) = x_1(t) * x_2(t) = \int_{-\infty}^{\infty} x_1(\tau) x_2(t-\tau) d\tau \tag{6.28}$$

Funkce $x_2(t)$ se obvykle nazývá konvoluční filtr. Při zpracování signálů působí konvolučního jádro (též konvoluční jádro či maska) na zdrojový signál. Výsledkem operace konvoluce je „násobení“ jednotlivých prvků zdrojového signálu s odpovídajícími prvky konvolučního jádra a následná sumace všech takto vzniklých dílčích výsledků.[3].

6.2.2 Diskrétní konvoluce

Na diskrétní konvoluci lze pohlížet jako na násobení vektoru maticí. Diskrétní konvoluce je obvykle popsána velmi řídkou maticí (matice jejichž položky jsou většinou rovny nule). Důvodem je, že jádro je obvykle mnohem menší než vstupní data. Jakýkoli algoritmus neuronové sítě, který pracuje s násobením matice a nezávisí na konkrétních vlastnostech struktury matice, by měl pracovat s konvolucí, aniž by vyžadoval jakékoli další změny v neuronové síti. Typické konvoluční neuronové sítě toho využívají další specializace za účelem efektivního řešení velkých vstupů. Jsou například velmi výhodné pro aplikace rozpoznávání obrazu.

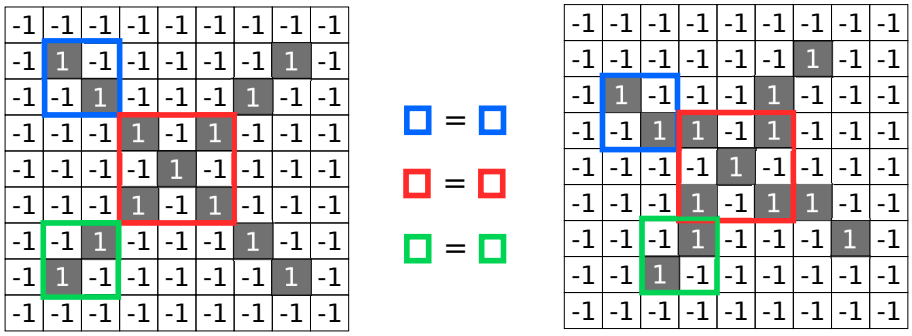
$$x(n) = x_1(n) * x_2(n) = \sum_{m=0}^{n} x_1(m) x_2(n-m) \tag{6.29}$$

Algoritmus výpočtu konvoluce dvou konečných posloupností spočívá v součtu dílčích součinů prvků posloupnosti x_1 a v čase invertované a o n prvků směrem v kladné časové ose posunuté posloupnosti x_2.

Vezmeme malé části pixelů, které se nazývají filtry, a pokusíme se je spojit na odpovídajících blízkých místech, abychom zjistili, zda dostaneme shodu. Díky tomu CNN síť mnohem lépe vyhledává podobnosti, než když se přímo pokouší porovnat celý obraz. Konvoluce tak představuje filtr, který redukuje počet vstupních hodnot.

6.2.3 Konvoluce obrazu

Konvoluce má tu pěknou vlastnost, že je invariantní vůči operaci posunutí. Intuitivně to znamená, že každý konvoluční filtr představuje jisté společné rysy (např. Pixely v písmenech) přičemž algoritmus CNN (Convolutional Neural Network) se učí, které funkce obsahují výslednou referenci (tj. Abecedu). Algoritmus není triviální, proto se jej pokusím vysvětlit na jednotlivých obrázcích.



Obrázek 6.12: *Tvary písmene X*

Na obrázku 6.12 je v pixelové formě vyjádřeno písmeno X. Na levém straně je X ve vzpřímené poloze zatímco na pravé straně obrázku, je X v šikmé znakové sadě. Každý ze znaků se v detailech odlišuje, lze však nalézt společné rysy. Na obrázku 6.12 jsou tyto oblasti (ne všechny), barevně vyznačeny. Pro člověka, schopného číst (což v řadě případů bohužel není samozřejmostí [3]) je

[3] Stačí jen letmo nahlédnout na stávající politickou scénu, kolik se tam vyskytuje negramotných jedinců

na levé i pravé straně písmeno X, nezávisle jak vypadá, protože v době, kdy se učil číst a psát, mu byla vlastnost více reprezentací téhož (zobecňování) přirozeně „vštípena“.

Pokud bychom naučili klasický deterministický algoritmus vyhodnotit, například přesným popisem souřadnic jednotlivých pixelů, že na levé straně je X, pak by si tentýž algoritmus s X na pravé straně nevěděl rady. Na konvoluční neuronové síti si ukážeme, jak tento problém skvěle vyřešit, aniž bychom museli popisovat všechny tvary písmene X, $\mathbf{X}, X, \mathcal{X}, \mathsf{X}$

6.2.4 Algoritmus konvoluční neuronové sítě

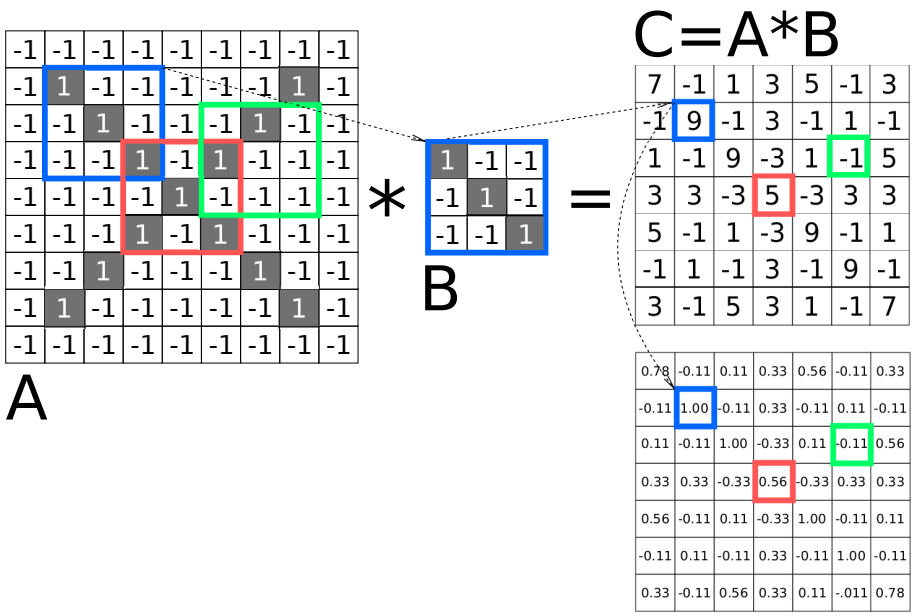
Na vstupní data se provede operace konvoluce. A to tak, že na vstupní data (signál) působí konvoluční filtr. Konvoluční filtr představuje pro jednorozměrná data vektor, pro vícerozměrná data tenzor příslušného řádu.

V našem příkladu učíme neuronovou síť rozeznávat tvary, tím pádem se v našem případě jedná o dvojrozměrná data a konvoluční filtr je dán maticí. Filtrační matice se nejčastěji používá v rozměru 3x3, nejvýše však 5x5. Pro konvoluci v neuronové síti je vhodné zvolit filtrační matici co nejjednodušší. Obecně však platí, že v oblasti zpracování obrazové informace, je volba velikosti filtrační matice relativně komplikovaná (stojí za tím poměrně velká matematika), nicméně pro určité typy filtračních operací (zaostření, rozostření, detekce hran) jsou doporučeny jisté, praxí vyzkoušené tvary.

Výpočet konvoluce

- Zarovnejte funkci a vstupní data
- Vynásobte každou hodnotu vstupního signálu odpovídající hodnotou filtrační funkce
- Výsledné hodnoty jednotlivých násobků sečtěte
- Vydělte výsledný součet celkovým počtem hodnot filtrační funkce

Princip je naznačen na obrázku 6.13.



Obrázek 6.13: *Konvoluce*

Matice B v roli filtrační funkce se postupně posouvá po matici dat a počítá hodnoty konvoluční funkce, přičemž výsledky ukládá v matici C. Matice C je v poslední fázi výpočtu dělena velikostí filtrační matice, (v našem případě 9).

Výpočet jednotlivých prvků konvoluční matice probíhá následovně: (na obrázku znázorněno například modrou barvou)

$$\begin{aligned} c_{[2,2]} = \quad & (1*1) + (-1*-1) + (-1*-1) + \\ & (-1*-1) + (1*1) + (-1*-1) + \\ & (-1*-1) + (-1*-1) + (1*1) \\ = \quad & 9 \end{aligned} \tag{6.30}$$

Pozor! Výpočet konvoluce není klasické násobení matice maticí, ale násobení a součet odpovídajících si prvků matice vstupu a filtrační matice.

V případě výpočtu „modré“ hodnoty (viz obr: 6.13) se vzájemně násobí a sečítají prvky matice A a B s těmito koeficienty.

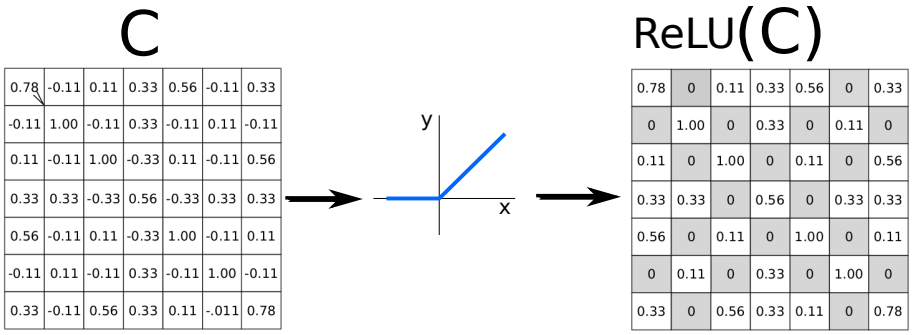
$$\begin{aligned} c_{[2,2]} &= a_{[2,2]}b_{[1,1]} + a_{[2,3]}b_{[1,2]} + a_{[2,4]}b_{[1,3]} + \\ &\quad a_{[3,2]}b_{[2,1]} + a_{[3,3]}b_{[2,2]} + a_{[3,4]}b_{[2,3]} + \\ &\quad a_{[4,2]}b_{[3,1]} + a_{[4,3]}b_{[3,2]} + a_{[4,4]}b_{[3,3]} \end{aligned} \tag{6.31}$$

Uplatnění aktivační funkce

Matice C obsahuje konvoluci vstupního signálu. Nyní je třeba uplatnit aktivační funkci. Výhradně se pro tyto aplikace volí $\sigma = ReLU = max\{0, x\}$. Funkce ReLU je velmi výhodná, protože vede u velmi rozsáhlých datových struktur na tzv. „řídké“ matice. [4]

Na řídké matice existuje řada efektivních algoritmů, které výrazně urychlují výpočty.

$$\sigma(x) = \begin{cases} 0, & \forall x \in (-\infty, 0) \\ x, & \forall x \in [0, \infty) \end{cases} \tag{6.32}$$



Obrázek 6.14: *Uplatnění funkce ReLU*

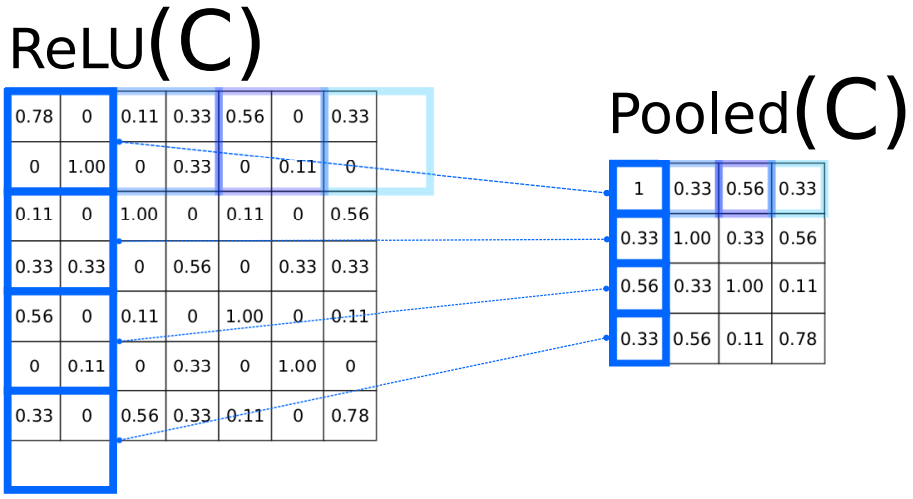
Poolovací vrstva

V této vrstvě dochází k redukci dat. Poolování se provádí po průchodu aktivační vrstvou. K tomu potřebujeme provést čtyři následující kroky:

- Vyberte velikost okna (obvykle 2 nebo 3)

[4] matice u nichž se velké množství prvků rovná nule

- Vyberte krok (obvykle 2)
- Projděte oknem přes filtrovaná data
- Z každého okna vezměte maximální hodnotu



Obrázek 6.15: *Poolovací funkce*

Na obrázku 6.15 je znázorněn princip poolingu.

Výše uvedené kroky, výpočet konvoluce, uplatnění aktivační funkce a pooling tvoří jádro CNN konvolučního algoritmu. Proces učení se skládá z uplatnění více průchodů konvolučním algoritmem za použití více filtrů.

Vypočítejme konvoluci vstupních dat se třemi různými filtry. Pak dostaneme:

pro vstupní funkci, která je reprezentována maticí (a pro nás znamenající písmeno X)

Působením konvoluce filtrem1, uplatněním ReLU a poolingu na vstupní matici reprezentující znak X, (viz obrázek 6.14)

Což se dá symbolicky zapsat jako:

$$\mathrm{X} * filtr_n \cdot \mathrm{ReLU} \cdot \mathrm{pooling} = pool_n$$

dostaneme matici poolu v tomto tvaru:

$$\text{pro vstupní matici (X)} = \begin{bmatrix} -1 & -1 & -1 & -1 & -1 & -1 & -1 \\ -1 & 1 & -1 & -1 & -1 & 1 & -1 \\ -1 & -1 & 1 & -1 & 1 & -1 & -1 \\ -1 & -1 & -1 & 1 & -1 & -1 & -1 \\ -1 & -1 & 1 & -1 & 1 & -1 & -1 \\ -1 & 1 & -1 & -1 & -1 & 1 & -1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 \end{bmatrix}$$

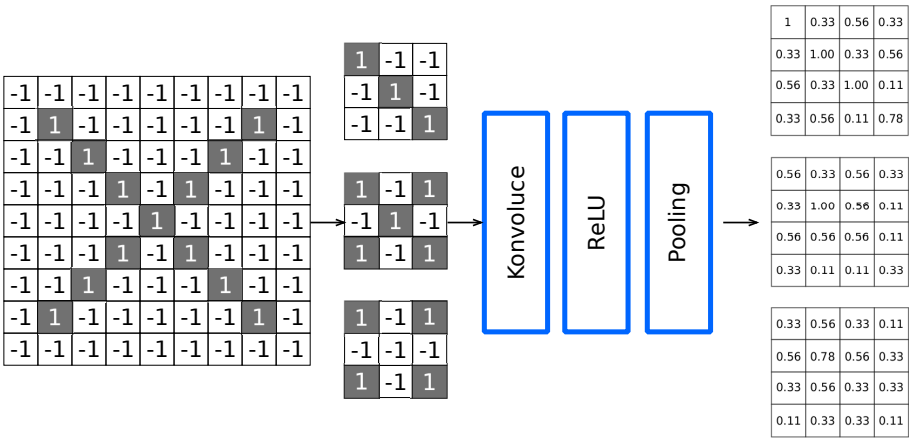
$$\text{a pro filtr1:} \begin{bmatrix} 1 & -1 & -1 \\ -1 & 1 & -1 \\ -1 & -1 & 1 \end{bmatrix} \text{získáme pool1:} \begin{bmatrix} 1.00 & 0.33 & 0.56 & 0.33 \\ 0.33 & 1.00 & 0.33 & 0.56 \\ 0.56 & 0.33 & 1.00 & 0.11 \\ 0.33 & 0.56 & 0.11 & 0.78 \end{bmatrix}$$

$$\text{a pro filtr2:} \begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix} \text{získáme pool2:} \begin{bmatrix} 0.56 & 0.33 & 0.56 & 0.33 \\ 0.33 & 1.00 & 0.56 & 0.11 \\ 0.56 & 0.56 & 0.56 & 0.11 \\ 0.33 & 0.11 & 0.11 & 0.33 \end{bmatrix}$$

$$\text{a pro filtr3:} \begin{bmatrix} 1 & -1 & 1 \\ -1 & 1 & -1 \\ 1 & -1 & 1 \end{bmatrix} \text{získáme pool3:} \begin{bmatrix} 0.33 & 0.56 & 0.33 & 0.11 \\ 0.56 & 0.78 & 0.56 & 0.33 \\ 0.33 & 0.56 & 0.33 & 0.33 \\ 0.11 & 0.33 & 0.33 & 0.11 \end{bmatrix}$$

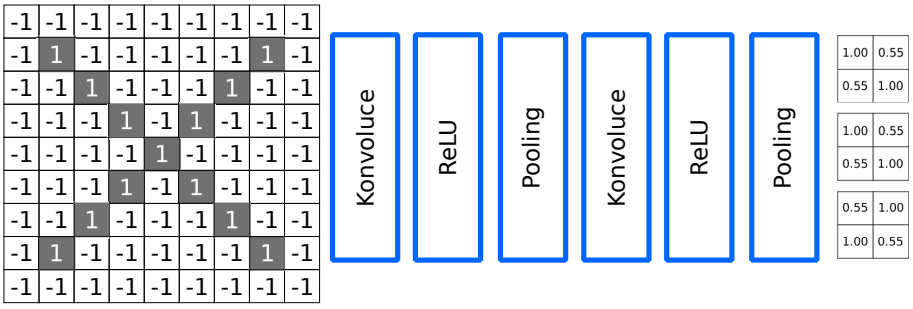
Poznámka: znak „*“ ve výše uvedených rovnicích představuje operaci konvoluce, nikoliv násobení!

Na obrázku 6.16 je znázorněn výpočet poolových matic pro více filtračních matic. V našem případě pro 3 různě zvolené filtry. Samozřejmě čím více použitých filtračních matic, tím získáme přesnější výsledek. Je však třeba mít na paměti poměr ceny a výkonu, kdy příliš velké množství filtrů výrazně zvyšuje výpočetní zátěž a tím i čas potřebný k učení sítě, přičemž kvalita učení se v závislosti na počtu filtrů již dále nezvyšuje a nebo se zvyšuje tak nepatrně, že na kvalitu funkce CNN to již nemá žádný podstatný vliv.



Obrázek 6.16: *Jeden krok výpočtu kovoluce*

Dalším krokem konvolučního algoritmu (viz obrázek 6.17), dále zredukujeme data tak, že výsledně dostaneme 3 poolové matice o rozměrech 2x2. Co s tím? Tyto matice lze triviální operací převést na vektor čísel a prohlásit, že tento vektor reprezentuje písmeno X.



Obrázek 6.17: *Výpočet konvoluce ve více krocích*

Takže námi naučená konvoluční neuronová síť si pamatuje, že znak X je reprezentován vektorem: [5].

$$\left[\ 1.00,\ \ 1.00,\ \ 0.55,\ \ 1.00,\ \ 1.00,\ \ 0.55,\ \ 0.55,\ \ 0.55,\ \ 0.55,\ \ 1.00,\ \ 1.00,\ \ 0.55\ \right]$$

Například písmeno B bude reprezenováno vektorem jiným, a to: [6]

$$\left[\ 1.00,\ \ 0.99,\ \ 0.66,\ \ 1.00,\ \ 1.00,\ \ 0.55,\ \ 0.55,\ \ 0.55,\ \ 0.70,\ \ 0.90,\ \ 1.00,\ \ 0.55\ \right]$$

Jak neuronová síť rozhodne, že písmeno X se rovná našemu naučenému X? Velmi jednoduše. Sečteme hodnoty vektoru reprezentujícího X a vektoru reprezentujícího X a porovnáme.

Pozor!!! Sčítají se jen hodnoty které jsou blízké hodnotě 1.

Příklad:

Znak X je reprezentován tímto vektorem.

$$\mathrm{X} = \left[\ 1.00,\ \ 1.00,\ \ 0.55,\ \ 1.00,\ \ 1.00,\ \ 0.55,\ \ 0.55,\ \ 0.55,\ \ 0.55,\ \ 1.00,\ \ 1.00,\ \ 0.55\ \right]$$

a znak X (v italice) je reprezentován tímto vektorem.

$$X = \left[\ 0.96,\ \ 0.55,\ \ 0.65,\ \ 0.98,\ \ 0.99,\ \ 0.65,\ \ 0.55,\ \ 0.54,\ \ 0.56,\ \ 0.96,\ \ 0.98,\ \ 1.00\ \right]$$

To znamená, že součet významných hodnot pro znak X (blízkých k 1) je:

$$X = 1.00 + 1.00 + 1.00 + 1.00 + 1.00 + 1.00 = 5$$

a pro znak X je:

$$X = 0.96 + 0.98 + 0.99 + 0.96 + 0.98 = 4.87$$

[5] X je poctivě spočítáno (skript v Octave (matlabu) je k dispozici v příloze

[6] Nepočítal jsem to, berte to prosím jen jako příklad

Následně můžeme vypočítat procentuální pravděpodobnost že X je totožné se znakem X.

$$X/x = 4.87/5 = 0.974$$

Takže závěrem lze prohlásit, že znak X je na 97.4 % písmenem X. A to je celé.

6.2.5 Definice sítě CNN v knihovnách TensorFlow a Keras

V Keras lze definovat více typů CNN sítí. Základ však představují sítě Conv1D, Conv2D a Conv3D.

```
tf.keras.layers.Conv2D(
    filters,
    kernel_size,
    strides=(1, 1),
    padding="valid",
    data_format=None,
    dilation_rate=(1, 1),
    groups=1,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Argumenty

filters:	Celé číslo, rozměrnost výstupního prostoru (tj. počet výstupních filtrů v konvoluci).
kernel_size:	Celé číslo nebo n-tice/seznam 2 celých čísel, určující výšku a šířku okna 2D konvoluce. Může to být jedno celé číslo, které určí stejnou hodnotu pro všechny prostorové dimenze.
strides:	Celé číslo nebo n-tice/seznam 2 celých čísel určující kroky konvoluce podél výšky a šířky. Může to být jedno celé číslo, které určí stejnou hodnotu pro všechny prostorové dimenze. Zadání jakékoli hodnoty kroku != 1 je neslučitelné se zadáním jakékoli hodnoty dilation_rate != 1.
padding:	jedno z „platných“ nebo „stejných“ (nerozlišují se malá a velká písmena). ”platný” znamená žádné vyplnění. ”stejné” vede k vyplnění nulami rovnoměrně vlevo/vpravo nebo nahoru/-dolů od vstupu tak, že výstup má stejnou výšku/šířku jako vstup.
data_format:	Řetězec, jeden z channel_last (výchozí) nebo Channels_first. Řazení rozměrů ve vstupech. Channels_last odpovídá vstupům s tvarem (velikost_dávky, výška, šířka, kanály), zatímco kanály_nejprve odpovídají vstupům s tvarem (velikost_dávky, kanály, výška, šířka). Výchozí hodnota je image_data_format, která se nachází ve vašem konfiguračním souboru Keras na adrese /.keras/keras.json. Pokud ji nikdy nenastavíte, bude to channel_last.
dilation_rate:	celé číslo nebo n-tice/seznam 2 celých čísel, určující rychlost dilatace, která se má použít pro dilatovanou konvoluci. Může to být jedno celé číslo, které určí stejnou hodnotu pro všechny prostorové dimenze. V současné době je zadání jakékoli hodnoty dilation_rate != 1 nekompatibilní se zadáním jakékoli hodnoty stride != 1.

groups:	Kladné celé číslo určující počet skupin, ve kterých je vstup rozdělen podél osy kanálu. Každá skupina je konvolvována samostatně pomocí filtrů / skupinových filtrů. Výstupem je zřetězení všech výsledků skupin podél osy kanálu. Vstupní kanály i filtry musí být dělitelné skupinami.
activation:	Aktivační funkce k použití. Pokud nic nezadáte, žádná aktivace se nepoužije (viz keras.activations).
use_bias:	Boolean, zda vrstva používá vektor bias.
kernel_initializer:	Inicializátor pro matici vah jádra (viz keras.initializers). Výchozí hodnota je 'glorot_uniform'.
bias_initializer:	Inicializátor pro vektor zkreslení (viz keras.initializers). Výchozí hodnota je 0.
kernel_regularizer:	Funkce regulátoru použitá na matici vah jádra (viz keras.regularizers).
bias_regularizer:	Funkce regulátoru použitá na vektor zkreslení (viz keras.regularizers).
activity_regularizer:	Funkce regulátoru aplikovaná na výstup vrstvy (její "aktivace") (viz keras.regularizers).
kernel_constraint:	Funkce omezení použitá na matici jádra (viz keras.constraints).
bias_constraint:	Funkce omezení použitá na vektor zkreslení (viz keras.constraints).

Vstupní tvar	4+D tenzor s tvarem: batch_shape + (kanály, řádky, sloupce), pokud data_format='channels_first' nebo 4+D tenzor s tvarem: batch_shape + (řádky, sloupce, kanály), pokud data_format='poslední_kanály'.
Výstupní tvar	4+D tenzor s tvarem: batch_shape + (filtry, nové_řádky, nové_sloupce), pokud data_format='channels_first' nebo 4+D tenzor s tvarem: tvar_dávky + (nové_řádky, nové_sloupce, filtry), pokud data_format='poslední_kanály'. hodnoty rows a cols se mohly změnit kvůli odsazení.
Návratová hodnota	4+D Tenzor představující aktivaci (conv2d(vstupy, jádro) + zkreslení).

Níže uvedených 6 řádků kódu definuje konvoluční základnu pomocí společného vzoru: zásobníku vrstev Conv2D a MaxPooling2D.

Jako vstup přijímá CNN tenzory tvaru (image_height, image_width, color_channels), přičemž ignoruje velikost dávky. Parametr color_channels odkazuje na (R,G,B). V tomto příkladu je nakonfigurována CNN tak, aby zpracovávala vstupy tvaru (32, 32, 3), což je formát obrázků CIFAR. To lze udělat předáním argumentu input_shape vaší první vrstvě.

Listing 6.2: *Inicializace sítě typu CNN*

```
import tensorflow as tf
from tensorflow.keras import datasets, layers, models
import numpy as np


model = models.Sequential()
model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
model.add(layers.MaxPooling2D((2, 2)))
model.add(layers.Conv2D(64, (3, 3), activation='relu'))
.
.
.
```

6.3 RNN - Rekurentní neuronové síťe

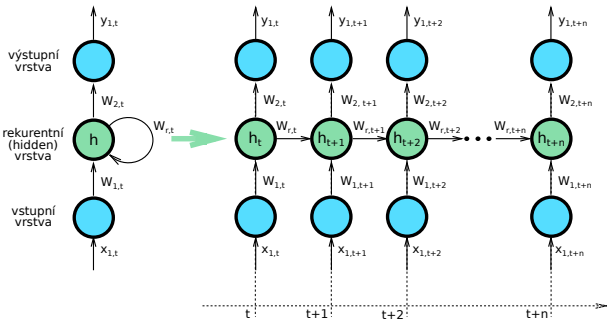
Lidé nezačínají přemýšlet od nuly, ale staví své myšlenky již na dříve vybudovaných zkušenostech. Tradiční neuronové sítě to nedokážou a zdá se, že jde o zásadní nedostatek.

Představte si například, že chcete klasifikovat, jaký druh události se děje v každém bodě filmu. Tradiční neuronová síť nedokáže z předchozího děje dedukovat události následující. „Zabili Bredlyho...“. Naše zkušenosti nám naznačují, že děj filmu se nebude točit kolem vyšetřování vraždy Bredlyho, ale výše vyřčená věta je zdrojem skvělého humoru geniálního Jiřího Menzela. Toto tradiční neuronové síti chybí a klidně může na základě této věty, film ohodnotit jako detektivku.

Rekurentní neuronové sítě tento problém do jisté míry řeší. Jsou to sítě, kde každý z neuronů má zavedenu zpětnou vazbu. To umožňuje informaci uchovávat k pozdějšímu využití.

RNN jsou rodinou neuronových sítí vhodných pro zpracování sekvenčních dat. Zatímco konvoluční neuronová síť je výhodná pro zpracování dat tvaru vícerozměrných matic, pak rekurentní neuronová síť je výhodná pro zpracování posloupnosti hodnot $x(1), ..., x(\tau)$. Síť typu RNN je velmi výhodná pro frekvenční analýzu, například v systémech pro rozpoznávání řeči.

Stejně jako konvoluční, lze rekurentní sítě snadno škálovat na rozsáhlé n dimenzionální datové struktury. Většina rekurentních sítí může také zpracovávat sekvence proměnné délky.

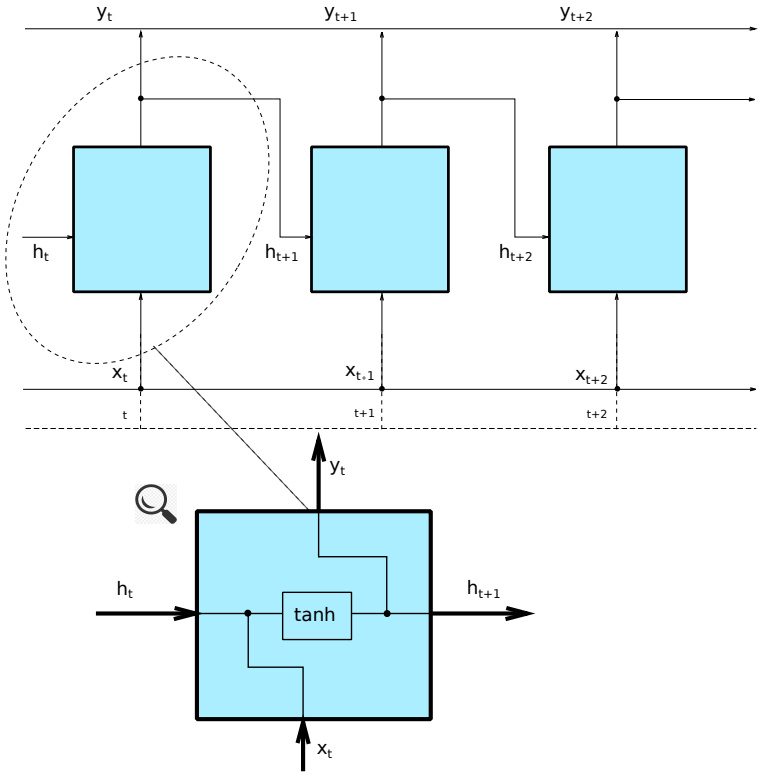


Obrázek 6.18: *Topologie sítě typu SimpleRNN*

Sdílení parametrů umožňuje rozšířit a aplikovat model na příklady různých forem (zde různé délky) a generalizovat je napříč nimi. Sdílení parametrů je velmi užitečné, když se například konkrétní informace objeví na více pozicích v rámci zkoumané sekvence.

Konvoluční přístup je základem neuronových sítí s časovým zpožděním. Konvoluční operace umožňuje síti sdílet parametry v čase. Výstupem konvoluce je posloupnost, kde každý člen výstupu je funkcí malého počtu sousedních členů vstupu. Myšlenka sdílení parametrů se projevuje v aplikaci stejného konvolučního jádra v každém časovém kroku.

Rekurentní sítě však sdílejí parametry odlišným způsobem od sítí konvolučních. Každý člen výstupu je funkcí předchozích členů výstupu. Každý člen výstupu je vytvořen pomocí stejného pravidla aktualizace použitého pro předchozí výstupy.



Obrázek 6.19: *Schéma SimpleRNN*

Mozek RNN tvoří vrstva paměťových buněk typu LTSM (Long Short-Term Memory). Ta udržuje stav buňky a také zajišťuje, že signál (informace ve formě přechodu) se při zpracování sekvence neztratí. V každém časovém kroku LSTM zohledňuje aktuální slovo, přenos a stav buňky.

6.3.1 Problém mizejícího gradientu

U rekurentních neuronových sítí se vyskytuje tzv.„problém mizejícího gradientu“ (vanishing gradient problem), který je historicky jednou z největších překážek úspěchu rekurentních neuronových sítí. Tento problém popsal Sepp Hochreiter, německý počítačový vědec, který se významnou měrou angažoval ve vývoji rekurentních neuronových sítí.[10]

Problém spočívá v tom, že algoritmus Back-propagation, který se používá také u rekurentních sítí, pracuje trochu v odlišném režimu. Zatímco u dopředných sítí, výsledky Back-propagation ve vyšší vrstvě, slouží jako vstup pro optimalizaci vrstvy předchozí, tak u rekurentních sítí výsledek Back-propagation ovlivňuje (díky zpětné vazbě) současně i výsledky stávající skryté neuronové vrstvy. Obecně lze říci, že gradient v hlubokých neuronových sítích je nestabilní a tíhne u hlubších vrstev sítě buď k mizení a nebo neúměrnému růstu gradientu, tzv. explozi, což má za následek, že neuronová síť ztrácí schopnost učit se. [11]

Problém mizejícího gradientu je způsoben multiplikativní povahou algoritmu zpětné propagace. To znamená, že gradienty vypočtené v hlubokých vrstvách rekurentní sítě mají buď příliš malý dopad (problém s mizejícím gradientem), nebo příliš velký dopad (problém s explozivně rostoucím gradientem) na váhy neuronů, které jsou ve vyšších vrstvách neuronové sítě. Zjednodušeně lze říct, že pokud máme číslo menší než 1 a vynásobíme ho několikrát proti sobě, skončíme s velmi malým číslem,(číslem které zmizí). Podobně mnohonásobné násobení čísla většího než 1 vede k velmi velkému číslu.

Problém mizejícího gradientu lze vyřešit řadou strategií, které tento problém vyřeší, nebo alespoň omezí.

Například je možno použít upravenou verzi algoritmu Back-propagation, nazvaný truncated Back-propagation, který omezuje počet kroků, přičemž zastaví algoritmus dříve, než dojde k problému s explozivně rostoucím gradientem.

Druhou možností je například inicializace vah, která zahrnuje umělé vytvoření počáteční hodnoty pro váhy v neuronové síti, aby se zabránilo tomu, že algoritmus Back-propagation přiřazuje nereálně malé váhy.

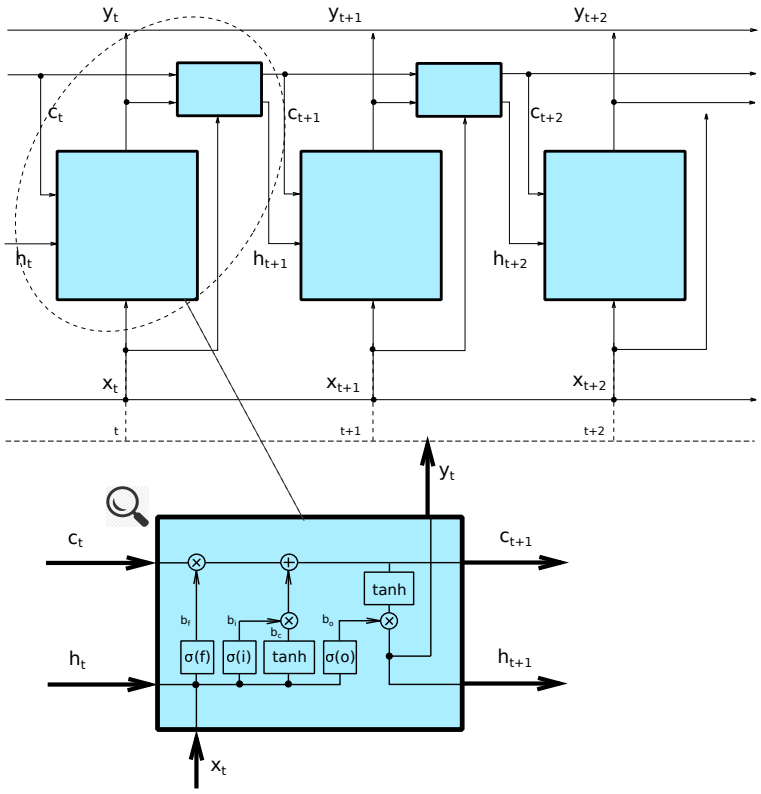
Efektivním řešením problému mizejícího gradientu je specifický typ neuronové sítě s názvem Long Short-Term Memory Networks (LSTM), jehož průkopníky byli Sepp Hochreiter a Jürgen Schmidhuber. LSTM se používají v problémech primárně souvisejících s rozpoznáváním řeči.

6.3.2 Long short-term memory (LSTM)

Sítě LSTM jsou typem rekurentní neuronové sítě používané k řešení problému mizejícího gradientu. Od „běžných“ rekurentních neuronových sítí se liší tím, že mají implementovány paměťové buňky, které výše uvedený problém řeší.

Faktor, který násobí rekurentní neuronovou síť v Back-propagation algoritmu, je obvykle značen W_r. To přináší již dříve zmíněné dva problémy:

- Jestliže se W_r blíží k malému číslu $W_r \to 0$, dochází k problému mizejícího gradientu
- Jestliže se W_r blíží k velkému číslu $W_r \to$ velké číslo, dochází k problému explozivního růstu gradientu



Obrázek 6.20: *Schéma LSTM*

LSTM síť tento problém řeší. Místo neuronů však mají sítě LSTM implementovány paměťové bloky, které jsou propojeny vrstvami. Blok má komponenty, díky nimž má trochu jiné vlastnosti nežli klasický neuron. Především má paměť pro předchozí sekvence výpočtu gradientu. Blok obsahuje brány, které spravují stav a výstup bloku. Blok pracuje na základě vstupní sekvence a každá brána v rámci bloku používá sigmoidní aktivační jednotky k řízení.

V rámci jednotky existují tři typy bran:

- **Forget Gate:** podmíněně rozhoduje, jaké informace vyhodit z bloku.
- **Input Gate:** podmíněně rozhoduje, které hodnoty ze vstupu aktualizují stav paměti.
- **Output Gate:** podmíněně rozhoduje o tom, co má být výstupem, na základě vstupu a paměti bloku.

Každý paměťový blok je chápán jako stavový automat, v němž jednotlivé brány mají váhy, jejichž hodnoty získává stavový automat v procesu učení.

Vzhledem k tomu, že se jedná o poměrně komplikovaný algoritmus (který přesahuje původní myšlenku „Velmi stručného úvodu do neuronových sítí“, rád odkážu na [10], [24], kde je vše brilantně vysvětleno včetně matematického backgroundu. Zde jsou uvedeny jen výsledné matematické vztahy nutné pro definici algoritmu Foward propagation a *Back-propagation Through Time* který umí problém mizejícího gradientu vyřešit.

Notace:

h_t, C_t	...	vektory skryté rekurentní vrstvy
x_t	...	vstupní vektor
b_f	...	bias vektor Forget gate
b_i	...	bias vektor Input gate
b_c	...	bias vektor přenosu (Carry)
b_o	...	bias vektor Output gate
W_f, W_i, W_c, W_o	...	parametrické matice
$\sigma, \tanh$	...	aktivační funkce

Algoritmus Forward Propagation:

$$\begin{aligned} f_t &= \sigma(W_f.[h_{t-1}, x_t] + b_f) \\ i_t &= \sigma(W_i.[h_{t-1}, x_t] + b_i) \\ o_t &= \sigma(W_o.[h_{t-1}, x_t] + b_o) \\ C_t &= \tanh(W_c.[h_{t-1}, x_t] + b_c) \\ C_t &= f_t \times C_{t-1} + i_t \times C_t \\ h_t &= o_t \times \tanh(C_t) \end{aligned} \tag{6.33}$$

Algoritmus Back-propagation Through Time:

Zřetězenými parciálními derivacemi rovnic pro Back propagation se lze dopracovat k tomuto výsledku:

$$\begin{aligned} \frac{\partial C_{t+1}}{\partial h_t} &= \frac{\partial C_{t+1}}{\partial \widetilde{C}_{t+1}} \frac{\partial \widetilde{C}_{t+1}}{\partial h_t} + \frac{\partial C_{t+1}}{\partial f_{t+1}} \frac{\partial f_{t+1}}{\partial h_t} + \frac{\partial C_{t+1}}{\partial i_{t+1}} \frac{\partial i_{t+1}}{\partial h_t} \\ \frac{\partial h_{t+1}}{\partial C_t} &= \frac{\partial h_{t+1}}{\partial C_{t+1}} \frac{\partial C_{t+1}}{\partial C_t} \\ \frac{\partial h_{t+1}}{\partial h_t} &= \frac{\partial h_{t+1}}{\partial C_{t+1}} \frac{\partial C_{t+1}}{\partial h_t} + \frac{\partial h_{t+1}}{\partial o_{t+1}} \frac{\partial o_{t+1}}{\partial h_t} \end{aligned}$$

$$\begin{aligned} \Pi_t &= \frac{\partial E_t}{\partial h_t} + \frac{\partial h_{t+1}}{\partial ht} \Pi_{t+1} + \frac{\partial C_{t+1}}{\partial h_t} T_{t+1} \\ T_t &= \frac{\partial E_t}{\partial h_t} \frac{\partial E_t}{\partial C_t} + \frac{\partial h_{t+1}}{\partial C_t} \Pi_{t+1} + \frac{\partial C_{t+1}}{\partial C_t} T_{t+1} \end{aligned} \tag{6.34}$$

$$\begin{aligned} b_t^f &= b_{t+1}^f + \frac{\partial C_t}{\partial f_t} \frac{\partial f_t}{\partial W_t^f} \left(\frac{\partial h_t}{\partial C_t} \Pi_t + T_t \right) \\ b_t^i &= b_{t+1}^i + \frac{\partial C_t}{\partial i_t} \frac{\partial i_t}{\partial W_t^i} \left(\frac{\partial h_t}{\partial C_t} \Pi_t + T_t \right) \\ b_t^c &= b_{t+1}^c + \frac{\partial C_t}{\partial c_t} \frac{\partial c_t}{\partial W_t^i} \left(\frac{\partial h_t}{\partial C_t} \Pi_t + T_t \right) \\ b_t^o &= b_{t+1}^o + \frac{\partial C_t}{\partial c_t} \frac{\partial c_t}{\partial W_t^i} (\Pi T_t) \end{aligned}$$

Produktem bran Forget Gate, Input Gate a Output gate jsou signály typu bias b_f, b_i, b_o, (viz obrázek 6.20 které spolu s paměťovou buňkou každého bloku neuronové sítě řeší algoritmus LSTM sítě. Matematický popis je zde zmíněn jen okrajově pro představu komplikovaného algoritmu Back-propagation Through Time. Pro podrobnější představu doporučuji [24].

6.3.3 Definice LSTM sítě v knihovnách Keras a TensorFlow

```
tf.keras.layers.LSTM(
    units,
    activation="tanh",
    recurrent_activation="sigmoid",
    use_bias=True,
    kernel_initializer="glorot_uniform",
    recurrent_initializer="orthogonal",
    bias_initializer="zeros",
    unit_forget_bias=True,
    kernel_regularizer=None,
    recurrent_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    recurrent_constraint=None,
    bias_constraint=None,
    dropout=0.0,
    recurrent_dropout=0.0,
    return_sequences=False,
    return_state=False,
    go_backwards=False,
    stateful=False,
    time_major=False,
    unroll=False,
    **kwargs
)
```

Argumenty

units:	Kladné celé číslo, dimenzionalita výstupního prostoru. aktivace: Aktivační funkce k použití. Default: hyperbolický tangens (tanh). Pokud projdete None, nebude použita žádná aktivace (tj. "lineární" aktivace: a(x) = x).
recurrent_activation:	Aktivační funkce, která se má použít pro opakující se krok. Default: sigmoid (sigmoid). Pokud projdete None, nebude použita žádná aktivace (tj. "lineární" aktivace: a(x) = x).
use_bias:	Boolean (výchozí True), zda vrstva používá vektor zkreslení.
kernel_initializer:	Inicializátor pro matici vah jádra, používaný pro lineární transformaci vstupů. Default: glorot_uniform.
recurrent_initializer:	Inicializátor pro matici vah recurrent_kernel, používaný pro lineární transformaci rekurentního stavu. Default: ortogonal.
bias_initializer:	Inicializátor pro vektor zkreslení. Default: nuly.
unit_forget_bias:	Boolean (výchozí True). Je-li True, přidejte 1 ke zkreslení brány zapomenutí při inicializaci. Nastavení na true také vynutí bias_initializer="nuly".
kernel_regularizer:	Funkce regulátoru použitá na matici vah jádra. Default: None.
recurrent_regularizer:	Funkce regulátoru použitá na matici vah recurrent_kernel. Default: None.
bias_regularizer:	Funkce regulátoru použitá na vektor zkreslení. Default: None.
activity_regularizer:	Funkce regulátoru aplikovaná na výstup vrstvy (její "aktivace"). Default: None.
kernel_constraint:	Funkce omezení použitá na matici vah jádra. Default: None.
recurrent_constraint:	Funkce omezení použitá na matici vah recurrent_kernel. Default: None.
bias_constraint:	Funkce omezení použitá na vektor zkreslení. Default: None.
dropout:	Pohybuje se v intervalu $< 0, 1 >$. Zlomek jednotek k poklesu pro lineární transformaci vstupů. Default: 0.
recurrent_dropout:	Pohybuje se v intervalu $< 0, 1 >$. Zlomek jednotek k poklesu pro lineární transformaci rekurentního stavu. Default: 0.

return_sequences: logická hodnota. Zda se má vrátit poslední výstup. ve výstupní sekvenci nebo v celé sekvenci. Default: $log(0)$.

return_state: Boolean. Zda se má kromě výstupu vrátit i poslední stav. Default: $log(0)$.

go_backwards: Boolean (výchozí False).

stateful: Boolean (výchozí $log(0)$). Je-li $log(1)$, poslední stav pro každý vzorek na indexu i v dávce bude použit jako počáteční stav pro vzorek indexu i v následující dávce.

time_major: Formát tvaru vstupů a výstupů tenzorů. Pokud je True, vstupy a výstupy budou mít tvar [časové kroky, dávka, prvek], zatímco v případě False to bude [dávka, časové kroky, prvek]. Použití time_major = True je o něco efektivnější, protože se vyhne transpozici na začátku a na konci výpočtu RNN. Většina dat TensorFlow je však hlavní dávka, takže tato funkce standardně přijímá vstup a vydává výstup ve formě hlavní dávky.

unroll: Boolean (výchozí False). Pokud je True, síť se rozvine, jinak se použije symbolická smyčka. Rozbalení může urychlit RNN, i když má tendenci být náročnější na paměť. Rozvinování je vhodné pouze pro krátké sekvence.

Call Argumenty:

vstupy:	3D tenzor s tvarem [dávka, časové kroky, funkce].
maska:	Binární tenzor tvaru [ukázky, časové kroky] udávající, zda má být daný časový krok maskován (volitelně, výchozí je Žádný). Jednotlivá položka True znamená, že by měl být použit odpovídající časový krok, zatímco položka False znamená, že odpovídající časový krok by měl být ignorován.
training:	Python boolean udávající, zda se má vrstva chovat v trénovacím režimu nebo v inferenčním režimu. Tento argument je předán buňce při jejím volání. To je relevantní pouze v případě, že je použit dropout nebo recurrent_dropout (volitelně, výchozí je None).
initial_state:	Seznam tenzorů počátečního stavu, které mají být předány prvnímu volání buňky (volitelné, výchozí je None, což způsobí vytvoření nulou vyplněných tenzorů počátečního stavu).

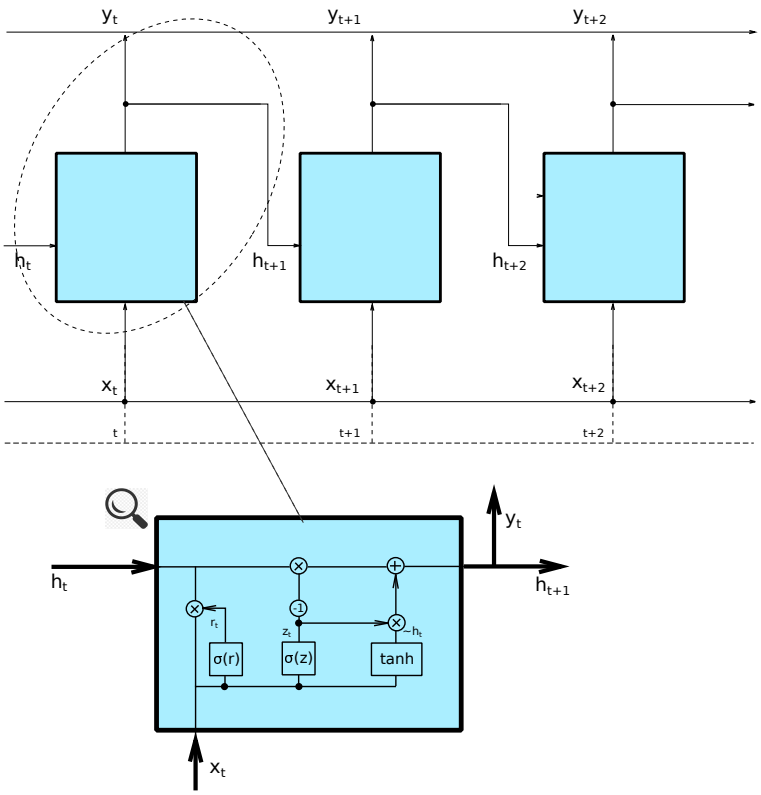
Přestože je matematické pozadí sítí typu LSTM poměrně komplikované, jejich definice s pomocí knihoven TensorFlow a Keras je překvapivě jednoduchá

Listing 6.3: *Inicializace neuronové sítě LSTM*

```
from Keras import layers
import numpy as np
import tensorflow as tf
.
.
.
model = Keras.model.Sequential();
model.add(layers.LSTM(128, input_shape=(maxlen, len(chars))))
model.add(layers.Dense(len(chars), activation('softmax')
.
.
```

6.3.4 Gated recurrent units (GRU)

Podobné vlastnosti jako LSTM sítě mají také tzv. Gated recurrent units (GRU), které jsou dalším typem rekurentních neuronových sítích. Výhodou GRU sítí je menší počet parametrů než u LSTM sítí. Tento typ sítě představili v roce 2014 Kyunghyun Cho et al. Ukázalo se, že GRU vykazují lepší výkon na určitých menších a méně častých souborech dat. Pro hlubší studium doporučuji [**cho2014learning**].



Obrázek 6.21: *Schéma GRU*

Princip GRU sítí, je stejný jako RNN, ale rozdíl je v provozu a hradlech spojených s každou jednotkou GRU. K vyřešení problému, kterému čelí standardní RNN, zahrnuje GRU dva mechanismy ovládání brány nazývané Update gate a Reset gate.

Update gate

Aktualizační brána je zodpovědná za určení množství předchozích informací, které je třeba předat v dalším stavu. To je opravdu mocné, protože model se může rozhodnout zkopírovat všechny informace z minulosti a eliminovat riziko mizejícího gradientu.

Reset gate

Resetovací brána se používá z modelu k rozhodnutí, kolik z minulých informací je potřeba zanedbat; zkrátka rozhoduje, zda je předchozí stav buňky důležitý nebo ne.

Nejprve se aktivuje resetovací brána, která ukládá relevantní informace z minulého časového kroku do nového obsahu paměti. Poté vynásobí vstupní vektor a skrytý stav jejich vahami. Dále vypočítá násobení po prvcích mezi resetovací bránou a dříve skrytým násobkem stavu. Po sečtení výše uvedených kroků se použije nelineární aktivační funkce a vygeneruje se další sekvence.

6.3.5 Definice GRU sítě v knihovnách TensorFlow a Keras

```
tf.keras.layers.GRU(
    units,
    activation="tanh",
    recurrent_activation="sigmoid",
    use_bias=True,
    kernel_initializer="glorot_uniform",
    recurrent_initializer="orthogonal",
    bias_initializer="zeros",
    kernel_regularizer=None,
    recurrent_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
```

```
    kernel_constraint=None,
    recurrent_constraint=None,
    bias_constraint=None,
    dropout=0.0,
    recurrent_dropout=0.0,
    return_sequences=False,
    return_state=False,
    go_backwards=False,
    stateful=False,
    unroll=False,
    time_major=False,
    reset_after=True,
    **kwargs
)
```

Argumenty objektu GRU

units:	Kladné celé číslo, dimenze výstupního prostoru.
activation:	Aktivační funkce. Default: hyperbolický tangens (tanh). Pokud projdete None, nebude použita žádná aktivace (tj. "lineární" aktivace: a(x) = x).
recurrent_activation:	Aktivační funkce, která se má použít pro opakující se krok. Default: sigmoid (sigmoid). Pokud projdete None, nebude použita žádná aktivace (tj. "lineární" aktivace: a(x) = x).
use_bias:	Boolean, (výchozí True), zda vrstva používá vektor bias.
kernel_initializer:	Inicializátor pro matici vah jádra, používaný pro lineární transformaci vstupů. Default: glorot_uniform.
recurrent_initializer:	Inicializátor pro matici vah recurrent_kernel, používaný pro lineární transformaci rekurentního stavu. Default: ortogonální.

bias_initializer:	Inicializátor pro vektor zkreslení. Default: nuly.
kernel_regularizer:	Funkce regulátoru použitá na jádro matice vah. Default: None.
recurrent_regularizer:	Funkce regulátoru použitá na matici vah recurrent_kernel. Default: None.
bias_regularizer:	Funkce regulátoru použitá na vektor zkreslení. Default: None.
activity_regularizer:	Funkce regulátoru aplikovaná na výstup vrstvy (její "aktivace"). Default: None.
kernel_constraint:	Funkce omezení použitá na matici vah jádra. Default: None.
recurrent_constraint:	Funkce omezení použitá na matici vah recurrent_kernel. Default: None.
bias_constraint:	Funkce omezení použitá na vektor zkreslení. Default: None.
dropout:	Pohybuje se v intervalu $< 0,1 >$. Zlomek jednotek k poklesu pro lineární transformaci vstupů. Default: 0.
recurrent_dropout:	Pohybuje se v intervalu $< 0,1 >$. Zlomek jednotek k poklesu pro lineární transformaci rekurentního stavu. Default: 0.
return_sequences:	logická hodnota. Zda vrátit poslední výstup ve výstupní sekvenci nebo celou sekvenci. Default: $log(0)$.
return_state:	Boolean. Zda se má kromě výstupu vrátit i poslední stav. Default: $log(0)$.
go_backwards:	Boolean (výchozí $log(0)$). Pokud je $log(1)$, zpracujte vstupní sekvenci zpět a vraťte obrácenou sekvenci.
statefull:	Boolean (výchozí $log(0)$). Je-li $log(1)$, poslední stav pro každý vzorek na indexu i v dávce bude použit jako počáteční stav pro vzorek indexu i v následující dávce.
unroll:	Boolean (výchozí $log(0)$). Pokud je $log(1)$, síť se rozvine, jinak se použije symbolická smyčka. Rozbalení může urychlit RNN, i když má tendenci být náročnější na paměť. Rozvinování je vhodné pouze pro krátké sekvence.

time_major: Formát tvaru vstupů a výstupů tenzorů. Pokud je True, vstupy a výstupy budou mít tvar [časové kroky, dávka, prvek], zatímco v případě False to bude [dávka, časové kroky, prvek]. Použití time_major = True je o něco efektivnější, protože se vyhne transpozici na začátku a na konci výpočtu RNN. Většina dat TensorFlow je však hlavní dávka, takže tato funkce standardně přijímá vstup a vydává výstup ve formě hlavní dávky.

reset_after: konvence GRU (zda použít resetovací bránu po nebo před násobením matice). False = "before", True = "after" (výchozí a kompatibilní s CuDNN).

Call Argumenty:

vstupy: 3D tenzor s tvarem [dávka, časové kroky, funkce].

maska: Binární tenzor tvaru [ukázky, časové kroky] udávající, zda má být daný časový krok maskován (volitelně, výchozí je Žádný). Jednotlivá položka True znamená, že by měl být použit odpovídající časový krok, zatímco položka False znamená, že odpovídající časový krok by měl být ignorován.

training: Python boolean udávající, zda se má vrstva chovat v trénovacím režimu nebo v inferenčním režimu. Tento argument je předán buňce při jejím volání. To je relevantní pouze v případě, že je použit dropout nebo recurrent_dropout (volitelně, výchozí je None).

initial_state: Seznam tenzorů počátečního stavu, které mají být předány prvnímu volání buňky (volitelné, výchozí je None, což způsobí vytvoření nulou vyplněných tenzorů počátečního stavu).

6.3.6 Porovnání topologie GRU a LSTM sítí.

1. GRU má dvě brány, LSTM má tři brány
2. GRU nemá žádnou vnitřní paměť, nemají výstupní bránu, která je přítomna v LSTM
3. V LSTM jsou vstupní brána a cílová brána propojeny aktualizační bránou a v GRU se resetovací brána aplikuje přímo na předchozí skrytý stav. V LSTM odpovědnost za resetovací bránu přebírají dvě brány, tj. vstup a cíl.

Podle empirického hodnocení není jasný vítěz. Základní myšlenka použití mechanismu získávání k učení dlouhodobých závislostí je stejná jako v LSTM.

6.3.7 Využití rekurentních neuronových sítí

Používají se pro podobné typy úloh jako sítě typu CNN, nicméně jejich hlavní doménou jsou úlohy pro zpracování přirozeného jazyka. Vynikají také v oblastech zpracování signálů, hudebního modelování, generování řečového signálu a rozpoznávání pojmenovaných entit (jméno, příjmení, adresa, datum...), strojový překlad, rozpoznávání řeči, jazykové modelování a generování textu, textové rešerše, analýza v call centech.

6.4 SOM - Self Organizing Map

Samoorganizující se mapa (SOM) nebo samoorganizující se funkce (SOFM) je technika strojového učení bez dozoru, která se používá k vytváření nízko dimenzionální (obvykle dvojrozměrné) reprezentace vyšší dimenzionální sady dat při zachování topologické struktury.

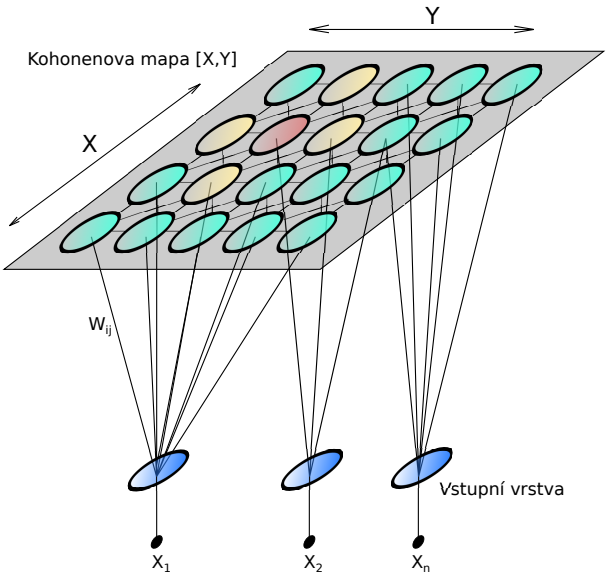
SOM je typ umělé neuronové sítě, která je trénována pomocí konkurenčního učení, nikoli učení závislém na korekci chyb (např. Backpropagation).

SOM zavedl finský profesor Teuvo Kohonen v 80. letech minulého století, a proto se mu někdy říká Kohonenova mapa nebo Kohonenova síť[14] . Kohonenova síť je výpočetně výhodná abstrakce stavící na biologických modelech neuronových systémů ze 70. let [17] a výpočetních modelech sahajících až k Alanu Turingovi v 50. letech 20. století [30].

Kohonenovy mapy jsou typem neuronové sítě, která provádí klastrování, známé jako samoorganizující se mapa. Tento typ sítě lze použít k seskupení datových sad do různých skupin, přestože vlastnosti těchto skupin nejsou známy. Data jsou seskupeny tak, že záznamy v konkrétní skupině nebo klastru jsou vzájemně podobné, přičemž data v různých skupinách jsou odlišná. Je to druh sítě která se s výhodou používá pro vizualizaci dat velkých dimenzí. Komprimuje data do geometrických vztahů s nízko dimenzionální reprezentaci.

Základními jednotkami jsou neurony, které jsou jsou organizovány ve dvou vrstvách. Vstupní vrstva a výstupní vrstva (výstupní mapa), jsou vzájemně propojeny klasickými neuronovými synapsemi, tedy všechny vstupní neurony jsou připojeny ke všem výstupním neuronům. Výstupní vrstva však tvoří matici $N \times M$.

Počet neuronů ve výstupní vrstvě se liší v závislosti na problému. Může se pohybovat od několika do několika tisíc v závislosti na složitosti problému. Během tréninku každý neuron soutěží se všemi ostatními, tak aby „vyhrál“ optimální nastavení.



Obrázek 6.22: *Topologie sítě typu SOM*

SOM zahrnuje tři základní rysy:

- **Konkurence:** Výstupní uzly (neurony) v samoorganizující se mapě mezi sebou soutěží, aby co nejlépe vystihly chování vstupního vzorku dat. Kvalita reprezentace vstupního vzorku dat se měří pomocí diskriminační funkce, která porovnává vstupní vektor s váhovým vektorem každého výstupního uzlu. Hledá se vítěz soutěže, kterého představuje uzel s takovými nastavenými vahami připojení, které se nejlépe shodují se vstupním vzorkem dat. Existuje celá řada funkcí, které určují vítěze. Nejčastěji se používá euklidovská vzdálenost.
- **Spolupráce:** Podobně jako „lidské“ neurony, které se zabývají blízce souvisejícími informacemi, jsou blízko sebe, takže mohou interagovat prostřednictvím krátkých synaptických spojení SOM je topografická organizace, ve které představují blízká umístění ve výstupním prostoru vstupy s podobnými vlastnostmi. To je možné za přítomnosti informace o sousedství. Vítězný uzel určuje prostorové umístění sousedství spolupracujících uzlů. Tyto uzly, sdílející společné funkce, se navzájem aktivují, aby se naučily něco ze stejného vstupu.

- **Adaptace:** Vektory váhy vítěze a jeho sousedních jednotek na mapě jsou upraveny ve prospěch vyšších hodnot jejich diskriminačních funkcí. Prostřednictvím tohoto procesu učení se příslušné uzly více podobají vstupnímu vzorku. Uzly, které mají silnou odezvu na konkrétní část vstupních dat, budou mít tedy v budoucnu zvýšenou šanci reagovat na podobná vstupní data.

Tyto rysy se rekurentně promítají na všechny tréninkové vzorky tak, jak jsou postupně předávány do sítě SOM. Tímto způsobem jsou různá místa na mapě vycvičena tak, aby vytvářela silnou odezvu na jisté charakteristické vlastnosti vstupních informací[2].

Síť SOM se obvykle skládá ze dvou vrstev uzlů, vstupní vrstvy a výstupní vrstvy, viz obrázek 6.22. Liší se od většiny ostatních neuronových sítí a v SOM je vstupní vrstva zdrojových uzlů přímo připojena k výstupní vrstvě výpočetních uzlů bez jakékoli skryté vrstvy [6]. Uzly ve vstupní vrstvě označují atributy (funkce) nebo obecněji proměnné obsažené ve vstupních datech. Každý kus vstupních dat je reprezentován n-rozměrným vstupním vektorem $x = (x_1 \dots x_n)$, jehož prvky označují hodnoty atributů konkrétní datové sady. Pokud existují velké rozdíly mezi hodnotami atributů v datové sadě, je nutná normalizace dat, aby se zabránilo dominanci určitého atributu nebo podmnožiny atributů. To poskytne všem atributům stejné šance a zlepší numerickou přesnost.

Z-Skore

Jedna z nejběžnějších metod která provádí transformaci prvků vektoru x s pomocí předpisu:

$$y_i = \frac{x_i - \mu}{\sigma}$$

kde: μ je střední hodnota (průměr) souboru hodnot a σ je směrodatná odchylka souboru hodnot.

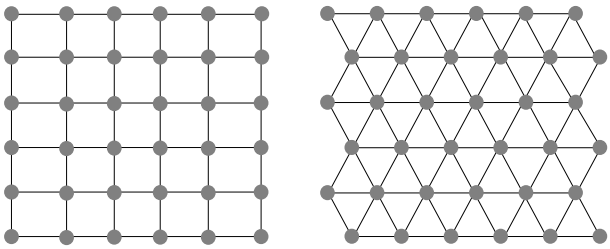
Metoda Min-Max

Min-Max metoda transformuje hodnoty vstupního vektoru do množiny hodnot [0,1] s pomocí triviálního vztahu

$$y_i = \frac{x_i}{max(x)}$$

Pro data jejichž hodnoty mají exponenciální závislost, se s výhodou používá logaritmická transformace.

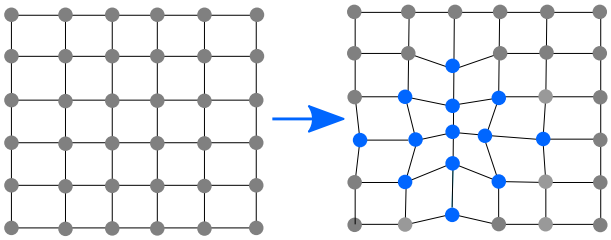
Topologie SOM vrstvy tvořící Kohonenovu síť, může být čtvercová nebo hexagonální. Mezi těmito topologiemi je obvykle preferována hexagonální struktura, protože vykazuje větší rozptyl ve velikosti jednotlivých „sousedů“. V obou topologiích je dovoleno rozlišovat počet řádků a sloupců. Přesto výzkum různých topologií ukazuje, že topologie čtvercového typu s velikostí sítě n × n dosahují nejlepších výsledků [13].



Obrázek 6.23: *Dva typy sítě typu SOM, čtvercová a hexagonální*

Algoritmus sítě typu SOM

Algoritmus vypočítá model, který nejlépe popisuje vlastnosti zkoumané datové domény. Model pozorované domény je uspořádán v dvojrozměrné mřížce, takže vlastnosti podobných shluků dat jsou k sobě blíže než různé.



Obrázek 6.24: *Princip algoritmu sítě typu SOM*

Shoda podobnosti

Proces učení v SOM se nespoléhá na předem definované cílové výsledky, které by sloužily jako vzory. Naopak, proces učení je koncipován tak, že jednotlivé uzly mezi sebou soutěží o aktivaci.

Aktivován a deklarován jako vítěz bude pouze uzel, jehož vektor váhy je nejvíce podobný vstupnímu vektoru. Nalezení minimální vzdálenosti mezi vstupními daty x a všemi váhovými vektory (w_i) v matici SOM se vypočítávají pomocí různých metod měření. Nejčastěji se však využívá vlastností Euklidovské vzdálenosti.

Euklidovská vzdálenost mezi vzorkem x, vybraným náhodně ze vstupní datové sady, a všemi váhovými vektory při iteraci t je definována následovně:

$$d_i(t) =\mid (x(t) - w_i(t)) \mid= \sqrt{\sum_{j=1}^{m} (x_{tj} - w_{tji})^2} \tag{6.35}$$

kde $\mid \dot{\mid}$ označuje euklidovskou normu, w_{ji} je váhový vektor,který tvoří váhová spojení v matici SOM.

Aktualizace vah

Vektory váhy vítěze a jeho sousedních jednotek ve výstupním prostoru jsou upraveny tak, aby se staly reprezentativnějšími pro funkce, které charakterizují vstupní prostor. Tato aktualizace směrem ke vstupnímu vzorku vyžaduje zvážení následujících dvou parametrů: rychlost učení a velikost sousedství. Rychlost učení, (t), řídí rychlost změny váhových vektorů a jako ve všech neuronových sítích nabývá hodnot mezi 0 a 1. U SOM rychlost učení postupně

klesá v závislosti na indexu kroku iterace t. To znamená, že zatímco se krokový index zvyšuje, rychlost učení se může snižovat lineárně, exponenciálně nebo geometricky [6], nebo může být nepřímo úměrná t [4, 17].

Zejména v případech, kdy jsou počátečním váhám přiřazeny náhodné hodnoty, by rychlost učení měla začínat na přiměřeně vysoké hodnotě, která je blízká jednotě, a postupně se snižovat na malé hodnoty. Tento postup odpovídá větším opravám na začátku tréninkového procesu (tj. Fáze řazení) než na konci, kde probíhá jemné doladění mapy (tj. Fáze konvergence) [4, 26]. Fáze řazení může trvat až 1 000 iterací algoritmu SOM a možná i více, zatímco konvergenční fáze adaptivního procesu musí být alespoň 500krát větší než počet neuronů v síti [4, 7].

Pomocí formalismu diskrétního času, vzhledem k váhovému vektoru $w_i(t)$ vítězného neuronu i při iteraci t, je aktualizovaný vektor váhy $w_i(t+1)$ při iteraci $t+1$ definován následovně:

$$w_i(t+1) = w_i(t) + \alpha(t)[x(t) - w_i(t)] \tag{6.36}$$

Jak bylo uvedeno výše, topologické vlastnosti původních dat lze zachovat pouze ve výstupním prostoru s ohledem na informace o sousedství. Na procesu učení se tedy podílejí i referenční vektory uzlů v sousedství. Rychlost přizpůsobení vah klesá směrem od vítězného uzlu, k uzlům ostatním. Neuron, jehož váhový vektor je nejvíce podobný vstupu, se nazývá jednotka s nejlepší shodou (best matching unit BMU). Váhy BMU a neuronů v jeho blízkosti v mřížce SOM jsou upraveny směrem ke vstupnímu vektoru. Velikost změny klesá s časem a se vzdáleností sítě od BMU.

Proces iterace, který zahrnuje Všechny tři zde popsané procesy - soutěž, spolupráce a adaptace - se rekurentně opakuje pro zbývající tréninková data, dokud se váhy nespojí a dokud nebudou pozorovány žádné viditelné změny ve výstupní vrstvě. Celá sada výsledných vektorů vah reflektuje rozložení vstupních dat.

Rychlost učení

Všimněte si, že pokud by byl koeficient rychlosti učení roven nule, váhy by se neměnily, tj. $w_1(1) = w_1(0)$. Alternativně, pokud by byla rychlost učení zvolena rovna jedné, nové váhy by se rovnaly vstupním datům, tj. $w_1(1) = x(0)$.

Kroky Kohonenova algoritmu SOM

1. **Inicializace**
   - Definujte rozměr matice výstupního prostorumxn.
   - Počátečním váhovým vektorům $w_i(0)$ přiřaďte náhodné hodnoty, nebo lze alternativně použít hodnoty vektoru vzorků,odebraného náhodně z tréninkové sady.
   - Zvolte funkci pro výpočet sousedních hodnot. Následně zvolte koeficient rychlosti učení α. Ten může být pro první iteraci velikosti 1, a bude se každým krokem procesu učení rekurentně zmenšovat, například $\alpha_{n+1} = \alpha n/2$.
   - Dále zvolte aktivační funkci (poloměr sousedství) $\sigma(0)$. Přiřaďte počáteční hodnoty pro $\alpha(0)$ a $\sigma(0)$. Normalizujte tréninková data. Definujte počet iterací tréninkového algoritmu
   - Náhodně vyberte ze sady tréninkových dat vstupní vektor x(t), nebo naplňte váhy matice SOM náhodnými hodnotami.
2. **Euklidovské vzdálenosti** Vypočítejte euklidovské vzdálenosti mezi vstupním vektorem a vektorem hmotnosti každého výstupního uzlu a najděte nejlépe odpovídající uzel c (t) při iteraci t použitím kritéria minimální vzdálenosti:

$$c(t) = argmin| \, x(t) - w_i(t) \, |, i = 1, \ldots, n$$

3. **Aktualizace vah** Upravte váhy vítězného uzlu a jeho okolí podle jejich vzdáleností od vítězného uzlu pomocí vzorce pro aktualizaci: Pro vítězný

uzel bude funkce sousedství $h_{ci}(t)$ rovna 1.

$$w_i(t+1) = w_i(t) + \alpha(t) h_{ci}(t) [x(t) - w_i(t)]$$

4. **Přenastavení parametrů** Nastavte t = t + 1. Upravte velikost sousedních uzlů a rychlost učení.
5. **Iterace** Vraťte se ke kroku 2, dokud nebude změna vah menší než předem stanovená prahová hodnota nebo dokud nebude dosaženo maximálního počtu T iterací.

Inicializační část algoritmu SOM

Listing 6.4: *Inicializace neuronové sítě typu SOM*

```
from tqdm import tqdm
import numpy as np
import tensorflow as tf

#----------------------------------------------------------------
#  SOM - implementace Kohonennovy mapy 2-D SOM s Gausovskou Neighbourhood
#     funkci a linearnim snizovanim rychlosti uceni
#  Prerekvizity
#    Python 3.8
#  Knihovny
#    TensorFlow 2.0
#    NumPy 1.19.5
#----------------------------------------------------------------
#  @Author
#    Implementace Kohonenovy mapy pro TensorFlow 1.5 (C) Sachin Joglekar
#    Do verze Tensorflow V2 prispel (C) Dragan Avramovski
#    pro pedagogicke ucely upravil (ponekud zprehlednil kod) Petr Lukasik
#----------------------------------------------------------------
class SOM(object):
    #Pro kontrolu, zda byl SOM natrenovan
    _trained = False

    #----------------------------------------------------------
    #Constructor objektu SOM
    #----------------------------------------------------------
    def __init__(self, m, n, dim, n_iterations=100, alpha=None, sigma=None):
        """
        Inicializace TensorFlow.

         m,n ....................... rozmery SOM.
         n_iterations .............. pocet iteraci
         dim........................ velikost vstupniho vektoru.
         alpha...................... rychlost uceni, vychozi hodnota je 0,3
         sigma...................... pocatecni hodnota sousedstvi, a oznacuje
                                     polomer vlivu BMU pri treninku.
                                     Ve vychozim nastaveni je nastavena na
                                     polovinu maxima (m, n).
        """
```

```
39
40          self._m = m
41          self._n = n
42          if alpha is None:
43              alpha = 0.3
44          else:
45              alpha = float(alpha)
46          if sigma is None:
47              sigma = max(m, n) / 2.0
48          else:
49              sigma = float(sigma)
50
51          self._n_iterations = abs(int(n_iterations))
52
53          #Init TenzorFlow grafu
54          self._graph = tf.Graph()
55
56          #inner objekt grafu
57          with self._graph.as_default():
58
59              #Nahodne inicializovane vahove vektory pro vsechny neurony,
60              #umisteno do vektoru o rozmeru(m x n, dim)
61              self._weightage_vects = tf.Variable(tf.random.normal([m*n, dim]))
62
63              #Matice SOM neuronu o velikosti [m*n, 2]
64              self._location_vects = tf.constant(np.array(list(self.
    _neuron_locations(m, n))))
65
66              #Alokace prostoru pro Treninkovy vektor
67              #placeholder - alokace prostoru typu float a velikosti dim
68              self._vect_input = tf.compat.v1.placeholder("float", [dim])
69              #Cislo iterace
70              self._iter_input = tf.compat.v1.placeholder("float")
71
72              #Vypocet nejlepsi shody s danym vektorem.
73              #Vypocita euklidovskou vzdalenost mezi kazdym
74              #vahovym vektorem a vstupem.
75              #Vrati index neuronu, ktery dava nejmensi hodnotu
76              tf_stack =  tf.stack([self._vect_input for i in range(m*n)])
77              tf_subtract = tf.subtract(self._weightage_vects, tf_stack)
78              tf_pow = tf.pow(tf_subtract, 2)
79              tf_sqrt = tf.sqrt(tf.reduce_sum(tf_pow, 1))
80              bmu_index = tf.argmin(tf_sqrt,0)
81
82              #Nalezeni lokace BMU na zaklade indexu BMU
83              slice_input = tf.pad(tf.reshape(bmu_index, [1]), np.array([[0, 1]]))
84              tf_slice = tf.slice(self._location_vects, slice_input, tf.constant(np.
    array([1, 2])))
85              bmu_loc = tf.reshape(tf_slice, [2])
86
87              #Vypocet konstant: alfa -rychlost uceni
88              #                  : sigma - polomer BMU
89              #v souvislosti s poctem iteraci
90              learning_rate_op = tf.subtract(1.0, tf.compat.v1.div(self._iter_input,
     self._n_iterations))
91              _alpha_op = tf.multiply(alpha, learning_rate_op)
92              _sigma_op = tf.multiply(sigma, learning_rate_op)
93
94
95              #Vektor rychlosti uceni pro vsechny neurony,
96              #na zaklade poctu iteraci a umisteni v BMU.
97              tf_stack = tf.stack([bmu_loc for i in range(m*n)])
```

```
98              tf_subtract = tf.subtract(self._location_vects, tf_stack)
99              tf_pow = tf.pow(tf_subtract, 2)
100             bmu_distance_squares = tf.reduce_sum(tf_pow, 1)
101
102             tf_compat_div = tf.compat.v1.div(tf.cast(bmu_distance_squares, "
    float32"), tf.pow(_sigma_op, 2))
103             tf_negative = tf.negative(tf_compat_div)
104             neighbourhood_func = tf.exp(tf_negative)
105             learning_rate_op = tf.multiply(_alpha_op, neighbourhood_func)
106
107             #Operace, ktera bude pouzivat learning_rate_op
108             #k aktualizaci vahovych vektoru vsech neuronu
109             #na zaklade konkretniho vstupu
110             tf_stack = [tf.tile(tf.slice(learning_rate_op, np.array([i]), np.array
    ([1])), [dim]) for i in range(m*n)]
111             learning_rate_multiplier = tf.stack(tf_stack)
112
113             tf_stack = tf.stack([self._vect_input for i in range(m*n)])
114             weightage_delta = tf.multiply(learning_rate_multiplier, tf.subtract(
    tf_stack, self._weightage_vects))
115
116             new_weightages_op = tf.add(self._weightage_vects, weightage_delta)
117             self._training_op = tf.compat.v1.assign(self._weightage_vects,
    new_weightages_op)
118
119             #init sesssion
120             self._sess = tf.compat.v1.Session()
121
122             #init promennych
123             init_op = tf.compat.v1.global_variables_initializer()
124             self._sess.run(init_op)
```

Rekurentní část algoritmu SOM - tréninkový cyklus

Listing 6.5: *Treninkový algoritmus neuronové sítě typu SOM*

```
    #--------------------------------------------------------------
    #metoda train
    #--------------------------------------------------------------
    def train(self, input_vects):

        #iterace
        for iter_no in tqdm(range(self._n_iterations)):
            #trenink vsech vektoru
            for input_vect in input_vects:
                feed_dict={self._vect_input: input_vect, self._iter_input: iter_no
    }
                self._sess.run(self._training_op, feed_dict)

        #ulozeni centroid_grid pro dalsi pouziti
        centroid_grid = [[] for i in range(self._m)]
        self._weightages = list(self._sess.run(self._weightage_vects))
        self._locations = list(self._sess.run(self._location_vects))

        for i, loc in enumerate(self._locations):
            centroid_grid[loc[0]].append(self._weightages[i])

        self._centroid_grid = centroid_grid

        #vytrenovano...
        self._trained = True
```

6.4.1 Kvalita SOM

SOM má schopnost interpretovat rozsáhlé datové struktury, v podobě nízko dimenzionálních struktur (provádí ztrátovou kompresi) při zachování topologie rozsáhlých struktur. Výstupy SOM se mohou lišit v závislosti na počátečním nastavení parametrů, jako je počet výstupních uzlů, rychlost učení a rychlost aktualizace. Proto je nutné strukturu SOM vyladit tak, aby interpretovala jisté typy dat co nejvěrněji. Neexistují však žádná obecně přijímaná pravidla pro parametrů. K řešení těchto problémů byly v literatuře vyvinuty různé metody měření pro hodnocení a porovnávání kvality výstupů SOM.

V tomto ohledu jsou nejvíce studovanými vlastnostmi SOM kvalita učení a kvalita interpretace výsledků (projekce).

Kvalita učení je označována jako vektorová kvantizace a kvalita interpretace jako zachování topologie. Obecně platí, že s rostoucí přesností interpretace

výsledků, klesá kvalita zachování topologie. To v praxi znamená, že snaha o zpřesnění výsledků jde obvykle proti kvalitě zachování topologie dat.

Kvantizační chyba

Quantization Error (QE) poskytuje prostředek k posouzení kvality učení a ukazuje, jak dobře mapa odpovídá datům. Vypočítává se určením průměrné vzdálenosti vzorkových vektorů k jednotce nejlepší shody [4]. Vzorec průměrné chyby kvantování je následující:

$$QE = \frac{\sum_{t=1}^{T} \mid x(t) - w_c(t) \mid}{T} \tag{6.37}$$

kde $w_c(t)$ je referenční vektor vah a $x(t)$ je vstupní vektor.

Topografická chyba

Zachování topologie, tj. Kvalita projekce, je vlastnost, jejíž definování je složitější a těžko měřitelné. Nejběžnějším měřítkem kvality pro zachování topologie je topografická chyba (TE), která může být vyjádřena jako procento datových vektorů, pro které první a druhý neuron, odpovídající jednotce BMU nepatří k sousedním jednotkám. Topografická chyba se vypočítá následovně:

$$TE = \frac{1}{T} \sum_{t=1}^{T} u(x(t)) \tag{6.38}$$

kde T je počet vstupních vzorků, a $u(x(t)) = 1$, pokud jednotky nejlepší a druhé nejlepší shody $x(t)$ spolu nesousedí, jinak $u(x(t)) = 0$. Jinými slovy, čím méně jsou narušeny sousedské vztahy v síti, tím lépe SOM zachovává topologii. Pro malé sítě je však tento výpočet velmi nepřesný a nespolehlivý, protože výpočet TE je příliš zjednodušen a tím pádem trpí diskrétní povahou výstupního prostoru.

6.4.2 Aplikace SOM

Problém obchodního cestujícího (TSP).

K seznamu měst a jejich vzdáleností, vyhledává algoritmus SOM nejkratší (nejlevnější) možnou trasu. Princip TSP spočívá v nalezení optimální trasy, které znamená, že každé město je navštíveno přesně jednou. V posledním kroku se algoritmus vrací do výchozího města.

Vlastnosti hledání sousedních optim, činí ze SOM vhodný prostředek pro nalezení optimální trasy. Když je poloha každého města reprezentována bodem v dvourozměrném vstupním prostoru a prohlídka sekvencí N výstupních uzlů (tj. Uzavřený řetězec), pak lze na řešení TSP pohlížet jako na mapování vstupu (prostor) na uzly řetězce. Po konečném počtu tréninkových cyklů se očekává, že se váhy výstupních uzlů shodují s umístěními měst.[28]

Problém návrhu výrobní buňky.

Ve výrobě jsou stroje seskupeny podle typů vyráběných dílů. Problém tvorby buněk se primárně týká shlukování strojních celků. Jako jednu z metod optimalizace problému může SOM úspěšně řešit seskupení strojů a dílů na základě sledu operací. Části produktu jsou definovány jako vstupní data, zatímco stroje jako výstupní uzly. Algoritmus SOM nejen navrhuje řešení klastrování, ale také poskytuje vizualizaci řešení.

Problém s optimálním umístěním produktu.

Problém, který zkoumá optimální zacílení produktu na zákazníka. Na základě vstupů typu konkurence, nasycenost trhu, preference zákazníka lze pomocí sítí SOM nalézt optima. SOM aproximuje neznámé rozdělení vícerozměrných pravděpodobností přizpůsobením topologicky seřazených jednotek (pozice produktu a zákazníka) do předem definovaného nízko dimenzionálního vjemového prostoru a propojuje tyto pozice s uvedenými nebo odhalenými údaji o výběru značky. Umožňuje provádět analýzu segmentace (klastrování) a polohování (mapování) současně. Ve srovnání s technikami s více proměnnými se postupy

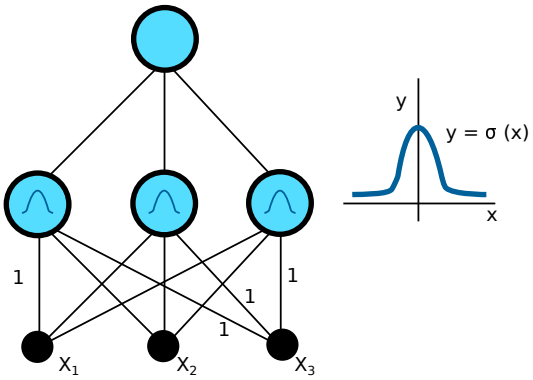
SOM snadno implementují, nevyžadují přísné distribuční předpoklady a ani nevyžadují zvláštní vlastnosti škálování nezpracovaných dat. Účinně si poradí s velmi malou rozměrností a vzorky neomezené velikosti. Podrobnosti o segmentaci trhu lze získat v [15],[12].

Problém sekvenování JIT (Multiple Object Just-In-Time).

Tento problém zvažuje systém plánování výroby JIT, kde jsou přítomny dva nepřímo související cíle minimalizace nastavení mezi různými produkty a optimalizace flexibility plánu. [19]

6.5 Sítě typu RBF - Radial Basis Function

Síť RBF je topologicky totožná se sítěmi typu Perceptron. Jediným rozdílem je aktivační funkce, která má jiný průběh, než standardní aktivační funkce používané v perceptronech. RBF sítě mimochodem nemají ani svou definici v knihovně TensorFlow. Proto je třeba definovat v rámci projektu, svoji vlastní vrstvu.



Obrázek 6.25: *Topologie sítě typu RBF*

Na rozdíl od standardní aktivační funkce (funkce s lichým průběhem) se zde uplatňují funkce radiální báze. Do množiny radiálních bází patří jakákoli funkce, která je definována jako funkce Euklidovské vzdálenosti od určitého centrálního bodu (poloměru). Zjednodušeně lze říct, že pravidlu funkce radiální báze vyhovuje sudá funkce, která vykazuje jistou symetrii, označovanou jako parita, pro kterou platí:

$$f(x) = f(|\, x \,|) \tag{6.39}$$

K tomuto typu funkcí patří například velmi často používaná funkce normálního (Gaussova) rozdělení, které se v sítích RBF používá velmi často.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma}} \tag{6.40}$$

Funkce aktivace neuronů RBF se mírně liší a obvykle se zapisuje jako:

$$\varphi(x) = e^{-\beta(\chi-\mu)^2} \tag{6.41}$$

6.5.1 Rozdíly v topologii RBF sítí v porovnání s MLP sítěmi

1. Síť RBF (ve své přirozené podobě) má jednu skrytou vrstvu, MLP mohou mít libovolný počet skrytých vrstev
2. Sítě RBF jsou obvykle plně propojeny, zatímco u MLP je běžné, že jsou propojeny pouze částečně.
3. V MLP sdílejí výpočetní uzly (procesní jednotky) v různých vrstvách společný neuronální model, i když ne nutně stejnou aktivační funkci. V sítích RBF fungují skryté uzly (základní funkce) velmi odlišně a mají velmi odlišný účel, než výstupní uzly.
4. V sítích RBF je argumentem každé aktivační funkce skryté jednotky vzdálenost mezi vstupem a „váhami“ (středy RBF), zatímco v MLP je to vnitřní součin vstupu a vah.
5. MLP jsou obvykle trénovány pomocí jediného globálního dohlíženého algoritmu, zatímco RBF sítě jsou obvykle trénovány po jedné vrstvě s první vrstvou bez dozoru.
6. MLP konstruují globální aproximace k nelineárním vstup-výstupním mapování s distribuovanými skrytými reprezentacemi, zatímco RBF sítě mají tendenci používat lokalizované nelinearity (Gaussovy) na skryté vrstvě ke konstrukci lokálních aproximací.

6.5.2 Použití RBF sítí

Velmi často se používají k vyhodnocování aktivity v oblasti elektroencefalografie (EEG).

7 Příklady nasazení ANN v Pythonu

7.1 Simulace logického obvodu s pomocí neuronové sítě

V následujícím příkladu je ukázka triviální neuronové sítě v roli simulátoru logického obvodu. Na základě vstupní sekvence logických nul a jedniček přepne výstup do požadovaného stavu. Je zde prezentován back-propagation algoritmus, a je také možno vyzkoušet chování sítě s různými počty skrytých vrstev. Je také možno testovat vliv počtu iterací - kroků učení a jejich závislost na kvalitě výstupní informace.

7.1.1 Triviální simulátor logického obvodu v Pythonu

Níže uvedený příklad demonstruje použití algoritmu Back-Propagation pro neuronovou síť, jejímž úkolem je rozeznat kombinace nul a jedniček na vstupu a interpretovat výsledek na výstupu. Příklad je napsán tak, že je možno simulovat různé parametry neuronové sítě.

Využívá pythonovské knihovny Numpy pro práci s vektory a maticemi.

Příkladu lze využít jako simulace velikosti a rozsahu neuronové sítě, s tím, že lze velmi snadno ověřit řadu provozních stavů a doporučení o velikosti vrstev a počtu vrstev, počtu iterací v procesu učení a přesnosti výsledku. Při jisté trpělivosti v procesu učení, lze docílit toho, že tato triviální síť rozpozná vstupní kombinaci [0,0,1; 0,1,0; 1,0,0] z jen jediné učební sekvence, například [0,0,1]. Tedy dokáže do jisté míry problém zobecnit. Vzhledem k tomu, že se jedná

z „pedagogických důvodů“ o velmi triviální příklad, v němž chybí například zapojení biasů, je vlastnost zobecňovat u této sítě velmi nestabilní.

Příklad 1.

Listing 7.1: *Triviální logický automat*

```
    #-------------------------------------------------------------------------
    # Simulace PLC
    # jednoducha neuronova sit pro simulaci PLC automatu
    # v zavislosti na vstupnim vektoru [x1,x2,x3,...,xn],  xi in [0,1]
    # navrhne vystupni vektor [y1,y2,y3,...yn], yi in [0,1] na zaklade
    # uciciho se algoritmu backpropagation.
    #-------------------------------------------------------------------------
    # definice neuronove site je dana ctyrmi hodnotami:
    # 1. vstupni vektor x-inputs,             min(x) = 2
    # 2. vystupnivektor y-outputs,            min(y) = 1
    # 3. pocet neuronu ve vrstve z-neurons,   min(z) >= min(x)
    # 4. pocet skrytych vrstev site a-hidden, min(a) >= 0
    #
    # Neuronova sit muze, ale nemusi obsahovat skryte vrstvy. Musi vsak
    # obsahovat vstupni a vystupni vrstvu.
    #
    # velikost neuronove site se definuje na radku:
    # neural_network = NeuralNetwork(inputs=x,
    #                                outputs=y,
    #                                hidden-layers=a,
    #                                neurons=z);
    #
    #-------------------------------------------------------------------------
    #  max. hodnoty jsou dany vypocetni mohutnosti daneho hardwaru,
    #  nedoporucuji vsak experimentovat s hodnotami > 128
    #  pamatujte ze se jedna o maticove operace, jejichz slozitost
    #  roste s N(^2) x pocet iteraci!!!. Pokud prezenete doporucenou
    #  velikost, narazite na velikost pameti, pripadne velkou casovou
    #  narocnost.
    #-------------------------------------------------------------------------

import numpy as np
#import matplotlib.pyplot as plt # graf zmeny chyby v prubehu treninku
#from pandas.core.arrays import integer

    #-------------------------------------------------------------------------
    # Class.NeuronLayer
    #-------------------------------------------------------------------------
class NeuronLayer():
    def __init__(self, neurons, inputs):
        self.weights = 2 * np.random.random((inputs, neurons)) - 1
        self.biases  = 2 * np.random.random((neurons)) - 1


    #-------------------------------------------------------------------------
    # Class.NeuralNetwork
    #-------------------------------------------------------------------------
class NeuralNetwork():
    def __init__(self, inputs, outputs, hidden_layers, neurons):

        if inputs < 2:
            print("pocet vstupu:", inputs, " je chybny-doplnena impl.hodnota");
            input = 2;
```

```
54          if hidden_layers < 0:
55              print("pocet vrstev:", layers, " je chybny-doplnena impl.hodnota");
56              hidden_layres = 0;
57          if outputs < 1:
58              print("pocet vrstev:", layers, " je chybny-doplnena impl.hodnota");
59              outputs = 1;
60          if neurons < inputs:
61              print("pocet neuronu:", neurons, " je chybny-doplnena impl.hodnota");
62              neurons = inputs;
63
64          self.layers  = hidden_layers + 2;   #vstupni, hiddden, vystupni
65          self.inputs  = inputs;
66          self.outputs = outputs;
67          self.neurons = neurons;
68          self.layer = [];
69          self.a = [];
70          self.error_history = []
71          self.epoch_list = []
72
73          for i in range (self.layers):
74              # vstupni layer -> pocet vstupu = pocet neuronu
75              if i == 0:
76                  self.layer.insert(i,NeuronLayer(self.neurons, self.inputs));
77              #vystupni layer -> pocet vystupu = pocet neuronu
78              elif i == self.layers - 1:
79                  self.layer.insert(i, NeuronLayer(self.outputs, self.neurons));
80              #mezivrstva -> pocet neuronu = self.neurons
81              else:
82                  self.layer.insert(i, NeuronLayer(self.neurons, self.neurons));
83
84
85
86      #-------------------------------------------------------------------------
87      #aktivacni  funkce ==> S(x) = 1/1+e^(-x)
88      #-------------------------------------------------------------------------
89      def sigmoid(self, x, deriv=False):
90      #   derivace sigmoidy
91          if deriv == True:
92              return x * (1 - x)
93      #   sigmoida
94          else:
95              return 1 / (1 + np.exp(-x))
96
97      #-------------------------------------------------------------------------
98      # treninkovy algoritmus - optimalizace hodnot vah a biasu.
99      #-------------------------------------------------------------------------
100     def train(self, training_input, training_output, it):
101         for i in range(it):
102             # forward propagation
103             self.forward_propagation(training_input);
104             # backward propagation
105             self.backward_propagation(training_input, training_output);
106
107
108         print ("pocet iteraci treninkoveho algoritmu", i+1);
109
110     #-------------------------------------------------------------------------
111     # forward propagation
112     #-------------------------------------------------------------------------
113     def forward_propagation(self, input_layer):
114         layer = input_layer;
115         for i in range(self.layers):
```

```
116             layer = self.sigmoid(np.dot(layer, self.layer[i].weights), deriv=False
    );
117             self.a.insert(i, layer);
118 
119 
120         return(layer);
121 
122 
123     #------------------------------------------------------------------------
124     # backward propagation
125     #------------------------------------------------------------------------
126     def backward_propagation(self, input_train, output_train):
127         # backward propagation
128         # vypocet chyby- rozdil mezi pozadovanym a predikovanym vystupem
129         for i in reversed(range(self.layers)):
130             if i == self.layers - 1:
131                 a_error = output_train - self.a[i].T;            #vstupni vrstva
132                 a_delta = a_error.T * self.sigmoid(self.a[i], deriv=True);
133             else:
134                 a_error = a_delta.dot(self.layer[i+1].weights.T);#vrstva i + 1
135                 a_delta = a_error * self.sigmoid(self.a[i], True);
136 
137             if i == 0:
138                 a_adjust = input_train.T.dot(a_delta);           #vrstva 0
139             else:
140                 a_adjust = self.a[i - 1].T.dot(a_delta);         #vrstva i - 1
141             # uprav hodnotu vah ve vrstve2.
142             self.layer[i].weights += a_adjust;
143 
144 
145     #------------------------------------------------------------------------
146     # main
147     #------------------------------------------------------------------------
148 
149 if __name__ == "__main__":
150 
151     #init random number generator
152     np.random.seed(1);
153 
154     #pocet pruchodu ucicim se algoritmem
155     pocet_iteraci_bp = 5000
156 
157     # Treninkova sada: 6 prikladu se ctyrmi vstupnimi hodnotami
158     # a dvema vystupnimi hodnotami
159 
160     training_input =  np.array([[0, 0, 1, 1],
161                                 [1, 0, 0, 0],
162                                 [0, 1, 0, 1],
163                                 [0, 1, 1, 1],
164                                 [1, 1, 1, 1],
165                                 [0, 0, 0, 0]]);
166 
167     training_output = np.array([[0,1],
168                                 [1,1],
169                                 [0,0],
170                                 [1,1],
171                                 [1,0],
172                                 [0,1]]).T;
173 
174     # Definice neuronove site
175 
176     input_size = training_input[0].size;
```

```
177     output_size = training_output.T[0].size;
178
179     neural_network = NeuralNetwork(inputs = input_size,
180                                    outputs = output_size,
181                                    hidden_layers = 3,
182                                    neurons = 8);
183
184     # Trenink neuronove site...
185     neural_network.train(training_input,
186                          training_output,
187                          pocet_iteraci_bp);
188
189     # vypocet prikladu...
190     print("Naucene priklady\n");
191     output = np.rint(neural_network.forward_propagation(np.array([0, 0, 1, 1])));
192     print ("Vstup [0, 0, 1, 1]; pozad. vystup:[0,1]:-> skut. vystup:", output);
193     output = np.rint(neural_network.forward_propagation(np.array([1, 1, 1, 1])));
194     print ("Vstup [1, 1, 1, 1]; pozad. vystup:[1,0]:-> skut. vystup:", output);
195     output = np.rint(neural_network.forward_propagation(np.array([0, 1, 0, 1])));
196     print ("Vstup [0, 1, 0, 1]; pozad. vystup:[0,0]:-> skut. vystup:", output);
197     output = np.rint(neural_network.forward_propagation(np.array([0, 1, 1, 1])));
198     print ("Vstup [0, 1, 1, 1]; pozad. vystup:[1,1]:-> skut. vystup:", output);
199
200     print();
201     print("Nenaucene priklady\n");
202     output = np.rint(neural_network.forward_propagation(np.array([1, 1, 0, 1])));
203     print ("Vstup [1, 1, 0, 1] -> ?: skut. vystup: ", output);
204     output = np.rint(neural_network.forward_propagation(np.array([1, 1, 0, 1])));
205     print ("Vstup [1, 1, 1, 0] -> ?: skut. vystup: ", output);
206     output = np.rint(neural_network.forward_propagation(np.array([1, 1, 0, 1])));
207     print ("Vstup [1, 0, 1, 1] -> ?: skut. vystup: ", output);
208     output = np.rint(neural_network.forward_propagation(np.array([0, 0, 0, 1])));
209     print ("Vstup [0, 0, 0, 1] -> ?: skut. vystup: ", output);
```

7.1.2 Simulátor logického obvodu v Pythonu s pomocí TensorFlow

TensorFlow je bezplatná a otevřená softwarová knihovna od GOOGLE pro strojové učení. Může být použit v celé řadě úkolů, ale má zvláštní zaměření na výcvik a odvozování hlubokých neuronových sítí. V oblasti výzkumu a nasazení metod umělé inteligence je využívána více jak v 70% všech aplikací.

Primárním programátorským prostředím TensorFlow je Python. Výpočetní uzly a tenzory jsou objekty Pythonu. TensorFlow však vlastní výpočty v pythonu neprovádí. Knihovny TensorFlow, jsou psány jako vysoce výkonné binární soubory v C++ nebo Cuda C. Python pouze směruje provoz mezi jednotlivými částmi a poskytuje programátorské abstrakce. Následující ukázka řeší stejný úkol jako příklad[1], využívá však vlastnosti profesionální knihovny TensorFlow.

Jinými slovy, TensorFlow je správná cesta pro projekty v oblastech umělé inteligence.

Příklad 2.

Listing 7.2: *Logický automat s využitím AI-Tensorflow*

```
    #-------------------------------------------------------------------------
    # Priklad PLC s pomoci neuronove site tensorflow
    # Tensorflow je sit poskytovana v Open Licenci Googlem.
    # Prerekvizity:
    #    vyvojove prostredi CUDA verze 11.4 minimalne
    # Pythonovske knihovny
    #    tensorflow (vypocty na CPU)
    #    tensorflow-gpu (vypocty na GPU) (nepovinne0
    #    keras - nadstavba nad tensorflow
    #-------------------------------------------------------------------------

import numpy as np
from keras import optimizers
from keras.models import Sequential
from keras.layers import Dense

    #-------------------------------------------------------------------------
    # Class.NeuralNetwork tensorflow
    #-------------------------------------------------------------------------
class NeuralNetwork():
    def __init__(self, inputs, outputs, neurons):

        self.model = Sequential();
        self.sgd = optimizers.SGD(lr=1);
        self.model.compile(loss='mean_squared_error', optimizer=self.sgd);
        # vstupni vrstva
```

```
27          self.model.add(Dense(units=neurons, activation='sigmoid', input_dim=inputs
    ))
28          # vystupni vrstva
29          self.model.add(Dense(units=outputs, activation='sigmoid'))
30
31          print(self.model.summary())
32
33  # Fixing a random seed ensures reproducible results
34  if __name__ == "__main__":
35      #init random number generator
36      np.random.seed(1);
37      #pocet pruchodu ucicim se algoritmem
38      pocet_iteraci_bp = 1500
39
40      # Treninkova sada: 6 prikladu se ctyrmi vstupnimi hodnotami
41      # a dvema vystupnimi hodnotami
42
43      training_input =  np.array([[0, 0, 1, 1],
44                                  [1, 0, 0, 0],
45                                  [0, 1, 0, 1],
46                                  [0, 1, 1, 1],
47                                  [1, 1, 1, 1],
48                                  [0, 0, 0, 0]]);
49
50      training_output = np.array([[0,1],
51                                  [1,1],
52                                  [0,0],
53                                  [1,1],
54                                  [1,0],
55                                  [0,1]]);
56
57      # Definice neuronove site
58
59      input_size = training_input[0].size;
60      output_size = training_output[0].size;
61
62      neural_network = NeuralNetwork(inputs = input_size,
63                                     outputs = output_size,
64                                     neurons = 32);
65
66
67      neural_network.model.fit(training_input, training_output, epochs=
        pocet_iteraci_bp, verbose=False);
68      print(neural_network.model.predict(training_input));
```

8 Vibrodiagnostika - síť typu SOM

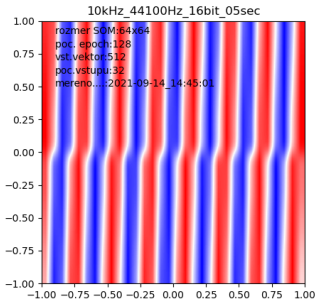
8.1 V časové oblasti

Vibrodiagnostika je velmi účinná metoda k diagnostice a prevenci poruchových stavů všech typů zařízení, kde se uplatňuje rotační nebo posuvný pohyb. Ve strojírenství má výsadní místo v oblasti udržitelnosti dlouhodobé jakosti obraběcích strojů. Úkolem je zavčasu odhalit vznikající poruchu, která sice ještě nemá vliv na kvalitu a přesnost obrábění, ale mohla by v budoucnu představovat závažný problém. Popíši zde možnost využití neuronové sítě, pro zobrazení změny frekvenčních spekter v čase, v souvislosti s dlouhodobým opotřebením a vznikem poruchy kuličkových ložisek. Síť typu SOM může generovat spektrální obrazy konkrétního zařízení a síť typu CNN tyto obrazy vyhodnocovat. V okamžiku, kdy spektrální obrazy začnou vykazovat anomálie proti původnímu stavu, může síť typu CNN dopručit preventivní údržbu.

Síť typu SOM je velmi užitečná k vizuálnímu zobrazení vazeb a vztahů nad zkoumanými daty. Této vlastnosti se pokusíme využít pro posouzení různých změřených spekter. Jedná se konkrétně o třífázový elektromotor, který měl prokazatelně vadná ložiska a tentýž motor, který byl následně generálkován. V praxi tuto úlohu snadno vyřešil zkušený údržbář, který s pomocí šroubováku přiloženému k uchu a vadnému domečku ložiska dokázal velmi spolehlivě diagnostikovat stav tohoto zařízení. Zkušených údržbářů je dnes jako šafránu, takže zkusíme tento úkol svěřit neuronové síti.

Jako druhé zařízení zkusíme analyzovat chování čerpadla, které má sice ložiska v pořádku, avšak jeho poruchou je ucpaná výstupní trubka.

Signály představují standardní zvukové soubory typu „WAV“ se vzorkovací frekvencí 44 kHz. Signál naměřených hodnot čerpadla a motoru v režimu porucha a bez poruchy byl předložen k analýze neuronové síti SOM. Bylo zvoleno více kritérií pro nastavení sítě (změna velikosti vstupních dat, změna velikosti matice SOM, změna počtu vstupů, mezi něž byl signál rozložen.



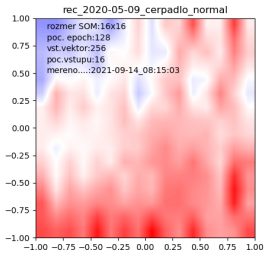
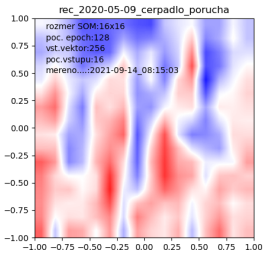
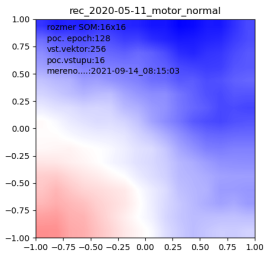
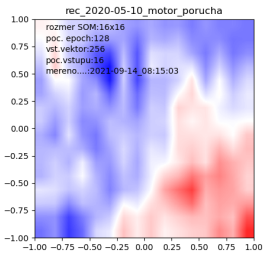
Obrázek 8.1: *Etalon 10 kHz v časové oblasti*

Na obrázku 8.1 je znázorněno, jak si neuronová síť poradí se signálem 10 kHz. Zde je patrná závislost sinusového průběhu signálu znázorněného pomocí Kohonenovy mapy. Na dalších obrázcích jsou již konkrétní signály získané z různých režimů jednoduchých pracovních strojů. Na nich je patrná souvislost mezi poruchou a normálním režimem práce. Porucha se vyznačuje výrazně „neklidným“ znázorněním kmitočtového spektra.

8.1.1 Databáze výstupů SOM v časové oblasti

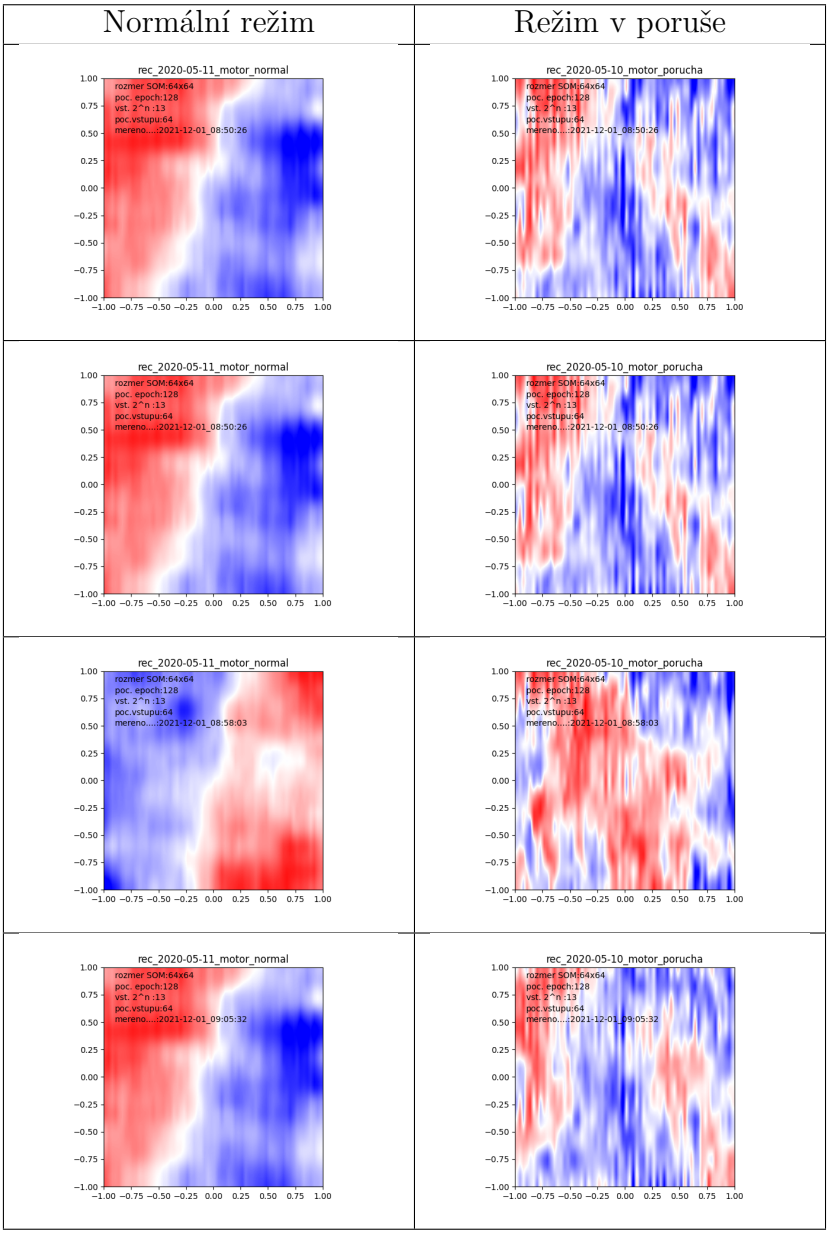
V tabulce 8.2 jsou znázorněny charakteristiky naměřených signálů v časové ose. V levém sloupci jsou charakteristiky motoru po generální opravě a v pravém sloupci jsou carakteristiky motoru v poruše. Z výše uvedeného vyplývá, že lze poměrně snadno detekovat společné vlastnosti zobrazení jak pro zařízení v poruše tak i po opravě.

Tato úloha může úspěšně pokračovat například tak, že tyto charakteristiky naučíme rozeznávat některou z konvolučních neuronových sítí, (které jsou pro rozpoznávání obrazu zvláště vhodné) a definovat mezní hodnoty, kdy neurono-

Normální režim	Režim v poruše
Čerpadlo bez poruchy	Čerpadlo v poruše
	
Motor po generální opravě	Motor s vadnými ložisky
	

Tabulka 8.1: *Porovnání charakteristik signálů v časové oblasti pomocí SOM sítě*

vá síť sama upozorní například na začínající poruchu a doporučí preventivní údržbu.



Tabulka 8.2: *Databáze výstupních 2D obrazů signálu v časové oblasti pomocí SOM sítě*

8.2 Ve frekvenčním spektru

Předchozí příklad byl interpretován nad čistými daty akustického souboru typu WAV. Z výše uvedeného (viz tabulka 8.2) vyplývá, že samoorganizovanou síť lze využít pro navržené účely.

Zde se však pokusíme nad standardním WAV souborem (souborem v časové oblasti) vyrobit spektrální analýzu, na kterou nasadíme úplně stejný typ sítě typu SOMa následně porovnáme výsledky z čistých datový souborů a souborů podrobených spektrální analýze.

Obecně je nutno poznamenat, že aparátu Fourierových transformací nelze použít pro náhodné procesy přímo a to z těchto důvodů [29][25].

- Realizace náhodného signálu není periodická, tudíž nelze využít Fourierovu transformaci.
- Fourierův integrál z realizace nekonečné řady nekonverguje. Teoretickou podmínkou náhodného signálu je, že představuje nekonečnou řadu. V praxi však tato podmínka nemůže být splněna, proto například u technického bílého šumu, se vlastnostem náhodného signálu jen do jisté máry přibližujeme.
- Pojem spektra u technicky realizovaného šumu není správný, protože spektrum nese informaci o zcela determinovaných vztazích mezi amplitudami a fázemi. U náhodného procesu však nemohou být tyto vztahy determinovány - nejednalo by se o náhodný signál.

Přesto se aparátu Fourierovy transformace používá i u náhodných procesů. Zavádí se spektrální výkonová hustota $S(\omega)$, přičemž vztah $S(\omega)d\omega$ představuje průměrný výkon za velmi dlouhou dobu v kmitočtovém pásmu šířky $d\omega$.

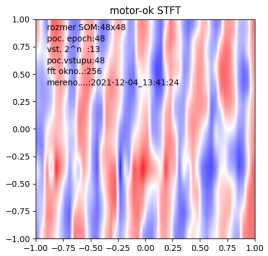
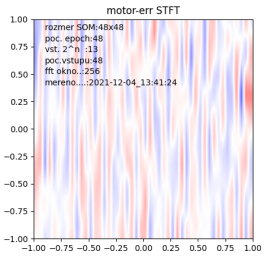
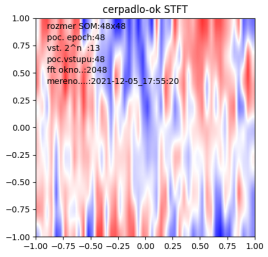
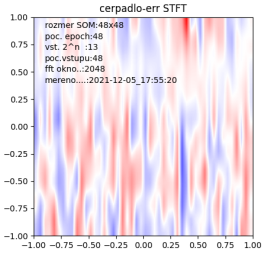
K tomu, aby se na takovýto (neperiodický) signál mohla uplatnit Fourierova transformace, musí splňovat dvě základní kritéria:

Signál musí být:

- **stacionární** střední hodnota měřeného signálu je stejná po celou dobu realizace. Jinými slovy, intenzita signálu nesmí kolísat.

- **ergodický** - existuje nulová pravděpodobnost, že střední hodnota v čase se neliší od střední hodnoty měřeného signálu, tj. $P(|\tilde{x} - \bar{x}| > \epsilon) = 0$. Zjednodušeně lze říct, že pokud je signál ergodický, splňuje podmínku, že střední hodnota změřená v jistém časovém intervalu se neliší od střední hodnoty nad realizací celého signálu.

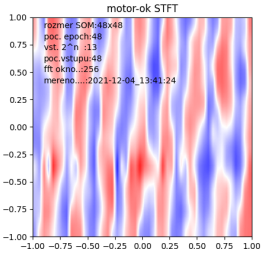
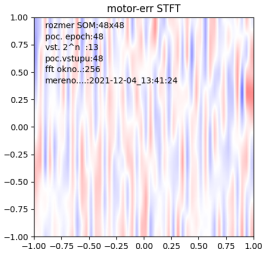
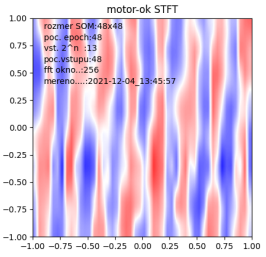
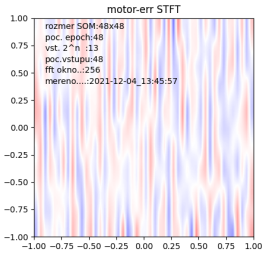
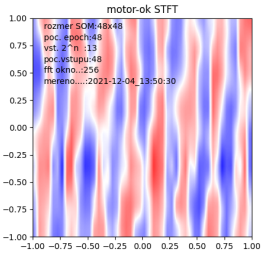
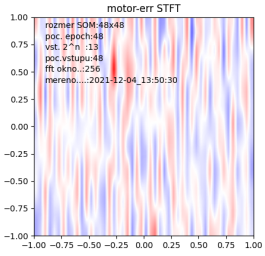
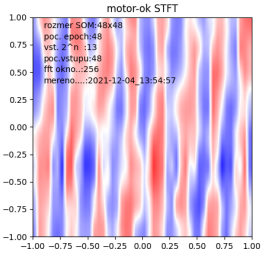
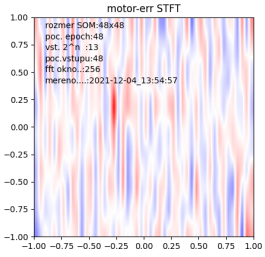
V tabulce 8.3 jsou vidět již konkrétní signály získané z různých režimů jednoduchých pracovních strojů. Na nich je patrná souvislost mezi poruchou a normálním režimem práce. Porucha se vyznačuje výrazně „neklidným“ znázorněním kmitočtového spektra. Následně byly tyto spektrogramy poslány do neuronové sítě SOM[1], přičemž výsledky byly následující.

Normální režim	Režim v poruše
	
	

Tabulka 8.3: *Znázornění spektrogramů pomocí sítě SOM*

[1] Byla to stejná síť se stejnými parametry, jako u příkladu výše

8.2.1 Databáze výstupů SOM ve spektrální oblasti

Normální režim	Režim v poruše
	
	
	
	

Tabulka 8.4: *Databáze výstupních 2D obrazů signálu ve spektrální oblasti pomocí SOM sítě*

Naměřené výsledky ukazují, že i ve spektrální oblasti lze rozhodnout do jaké kategorie jednolivá měření spadají. Nutno podotknout, že v této oblasti

není tak výrazný rozdíl v zobrazení poruchy a správné funkce. Nutno však podotknout, že lze dosáhnout lepších výsledků odladěním vlastností SOM sítě. Nastavení optimálních vlastností je vždy uplaňována metoda pokusů a omylů, protože nelze dopředu určit, které z parametrů sítě budou pro danou úlohu zásadní.

9 Digitální dvojče - příklad sítě typu DENSE

9.1 Digitální dvojče

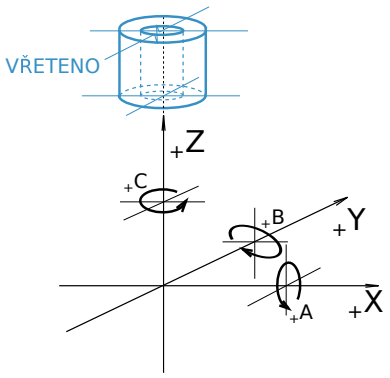
Digitální dvojče je digitální (počítačový) model reálného, například výrobního, měřicího apod. automatizovaného zařízení, na němž lze simulovat jeho fungování, komunikaci mezi jeho složkami atd. Může se také učit z různých zdrojů a adaptovat na měnící se podmínky. Taková virtuální replika reálných zařízení pomáhá odhalit různé chyby a nesrovnalosti ještě předtím než se dané zařízení uvede do provozu. Pojem se vyskytuje například v souvislosti s momentálním nástupem programu Průmysl 4.0. Využívá se hlavně ve výrobních závodech, kde dokáže zkrátit dobu zprovoznění nových linek či závodů a umožňuje zvyšovat jejich efektivitu. [30].

V našem příkladu bude model digitálního dvojčete aplikován na úlohu kompenzace teplotních dilatací obráběcího stroje. Teplotní roztažnost materiálů má výrazný vliv na přesnost výrobku. Při požadavku přesnosti $0,01[mm]/1[m]$, mají tepelné dilatace již velmi významný negativní vliv na proces obrábění. Snahou je proto umísťovat ty nejpřesnější stroje do klimatizovaných místností.

To však v mnoha případech není možné a nebo je velmi nákladné. Existuje však možnost, díky vlastnostem moderních řídicích systémů, teplotní dilatace kompenzovat korekcemi v jednotlivých lineárních osách (X, Y, Z)[1] a rotačních

[1] Platí, že osa Z je rovnoběžná s osou pracovního vřetena, přičemž kladný smysl probíhá od obrobku k nástroji.

osách (A,B,C). Pro obsažení kompletní 3D geometrie prostoru, jsou nutné tři lineární osy (X, Y, Z) a dvě osy rotační, např (A, B).[2]



Obrázek 9.1: *Osy vertikálního centra - pravotočivý souřadný systém*

V případě, že stroj má řízeny pouze tři lineární (X, Y, Z) osy, nelze v praktických aplikacích obsáhnout obraběcím nástrojem všechny body 3D prostoru.

Nutno podotknout, že ve většině případů obrábění postačí řízení tří (X, Y, Z) os. Pětiosé obrábění se používá pro obrábění tvarově komplikovaných tvarů (lopatky turbín, lodní šrouby a podobně).

V našem příkladu se pokusíme kompenzovat tepelné dilatace tříosého obráběcího centra MCFV 1260i[3], které je ve výrobním programu firmy TAJMAC-ZPS, a.s.

V současné době, je teplotní kompenzace počítána empirickými vztahy, které však nepostihují dynamiku ani nelinearitu teplotních roztažností konstrukce v požadované přesnosti.

Jako elegantní řešení se nabízí vytvoření digitálního dvojčete, které obsahuje informace o vývoji teplot a deformací rámu stroje v závislosti na pracovním zatížení a čase. Toto digitální dvojče je možno následně použít jako vstupní informaci pro trénink neuronové sítě, která by na základě naučených souvislostí, poskytovala přesnější údaje o tepelné roztažnosti a jejího vlivu na geometrii stroje.

[2] V závislosti na konstrukci stroje
[3] publikováno s laskavým svolením TAJMAC-ZPS, a.s.

Obrázek 9.2: *stroj MCFV 1260i z produkce TAJMAC-ZPS, a.s.*

Navržené hodnoty následně poslat řídicímu systému, jako kompenzační parametry pro eliminaci geometrické odchylky stroje. Cílem je, aby neuronová síť dokázala predikovat kompenzační data s přesností optimálně (± 5) [μm].

9.2 Struktura a návrh aplikace

Pro trénink sítě bylo nezbytné získat data z reálného provozu, která obsahují obraz chování stroje v závislosti na teplotách a geometrických odchylkách v časové ose. Na stroji byla umístěna sada teplotních čidel pt100 a spolu s měřením odchylek (s pomocí měřicích artefaktů, které jsou umístěny na pracovní ploše stolu), byla sbírána data (ve vzorkovacím intervalu cca 5 minut). Tato data, která můžeme prohlásit za model digitálního dvojčete, obsahovala závislosti teplot na geometrickém tvaru obraběcího prostoru, do něhož byla zahrnuta tepelná roztažnost os X, Y a Z. Jednalo se o cca 80 000 vzorků, které obsahovaly informace o vlivu teplot, vlhkosti a slunečního osvitu na celkovou geometrii stroje.

Úkolem procesu učení je následně získat co nejmenší ztrátovou funkci, která určuje minimální rozdíl predikčních a skutečných vlastností sledovaného objektu[5].

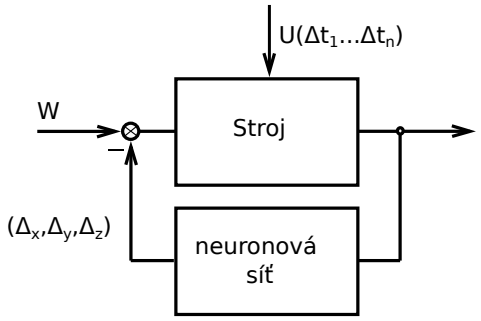
$$J(y) = min(f((t_1, ..., t_i), (\Phi_1, ..., \Phi_j), (H_1, ..., H_k), \Delta(x, y, z))$$

Po tréninku, jsou následně síti v režimu predict předkládána data v podobě vývoje teplotních poměrů. Úkolem je co nejpřesněji interpretovat geometrické odchylky.

$$\Delta(x, y, z) = f((t_1, ..., t_i), (\Phi_1, ..., \Phi_j), (H_1, ..., H_k))$$

Situaci lze popsat klasickým regulačním schématem, kdy žádanou hodnotou W je co nejvyšší přesnost soustavy, přičemž do soustavy vstupuje porucha v podobě změny teplot. Kompenzace poruchového stavu je prováděna ve zpětné vazbě neuronovou sítí. V tomto případě stačí kompenzace pruchové veličiny U, přičemž žádaná kodnota W vstupuje do soustavy jenom jako konstanta (tedy $\Delta(x, y, z) = 0$).

Vlastní kompenzace na žádanou hodnotu, na základě výsledků z neuronové sítě, je svěřena řídicímu systému stroje, který se postará o vyrovnání odchylek tak, aby se regulovaná soustava co nejvíce přiblížila žádané hodnotě. To znamená docílit geometrickou odchylku maximálně v intervalu (± 5) [μm].



Obrázek 9.3: *Regulační schéma*

9.3 Vliv teplot na změnu geometrie stroje

Pro optimální návrh tréninku sítě bylo nutné určit, která měřená místa mají na geometrickou soustavu největši vliv. Na základě srovnání grafických závislostí a jejich vlivu na co nejlepší výsledek bylo zjištěno, že nejvíce významné na celkové chování systému jsou teploty ve vřetenu stroje a také teploty jednotlivých pravítek os X, Y, Z.

V praxi se ukázalo, že veličiny vlhkost $\Phi_1 ... \Phi_j$ a osvit $H_1 ... H_j$ trénink sítě ovlivňují velmi málo, proto byly z množiny predikčních dat vyloučeny. Takže do výpočtu ztrátové funkce vstupují jen hodnoty naměřených geometrických odchylek a teplot.

$$J(y) = min(f(t_1, ..., t_i), \Delta(x, y, z))$$

Takže výsledná interpretace geometrických odchylek je závislá jen na teplotách.

$$\Delta(x, y, z) = f(t_1, ..., t_i)$$

Směry teplotních dilatací jsou znázorněny na obrázku 9.4. Minimální vliv na nepřesnosti obrábění v důsledku teplotních dilatací má osa X, jejíž deformace jsou osově souměrné spolu s obrobkem. Teplotní roztažnost obrobku a upínacího stolu se kompenzuje sama a není třeba se jí zabývat. Přesto byla do tréninku sítě zařazena, ač na celkový výsledek kompenzace stroje má minimální vliv.

Poněkud komplikovanější je situace v ose Y, kdy teplotní roztažnost nepůsobí osově souměrně, takže nepřesnosti vlivem této poruchy jsou již patrné a jsou předmětem teplotní kompenzace.

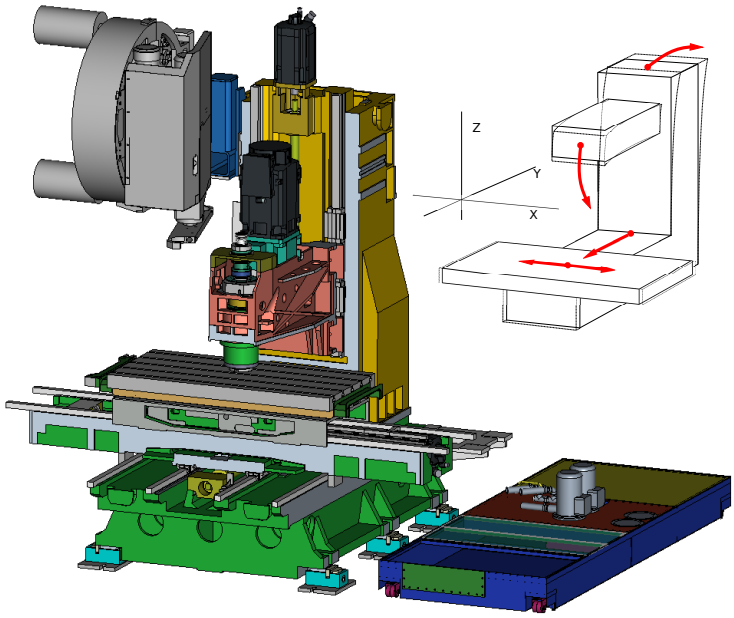
Poměry dilatací v ose Z jsou relativně komplikované, protože na vzniku geometrických odchylek vlivem teplot se podílí jak osa Y (díky osově nesouměrné roztažnosti), ale také deformace hlavního stojanu a vlastního vřeteníku.

Teplotní závislost zde výrazně ovlivňuje pohon posuvu osy Z a pohon vřetene, který konstrukci stroje zahřívá až na 70 $°C$. Navíc zde vznikají nepřesnosti,

které jsou způsobeny odklonem stojanu dozadu (teplotní vliv pohonu osy Z) a vřeteníku, který se deformuje směrem dolů (vliv hlavního pohonu vřetene).

Ač jdou tyto deformace proti sobě, (částečně se vzájemně kompenzují), přesto dochází k osové nesouměrnosti hlavního vřetene vůči stolu, cca 0.03[mm]/1[m]. I když to není mnoho, tato nepřesnost se projevuje zejména při obrábění dlouhých součástí. Vyosení vřetene, které takto vzniká, nelze u tříosého obráběcího centra dostatečně přesně kompenzovat.

Toho lze docílit až u pětiosých center. U tříosého centra lze kompenzovat pouze směryh X, Y a Z. Vzniká tak úhlové vyosení, které, nelze odstranit. V praxi to znamená, že například velmi dlouhý vrtaný otvor má tvar deformované elipsy. Dochází tak ke ztrátě kruhovitosti otvoru). Jedná se však o zanedbatelné nepřesnosti v řádu jednotek mikronů $[\mu m]$, takže je v kompezační úloze zanedbán.



Obrázek 9.4: *Směry teplotních dilatací stroje MCFV 1260i*

9.4 Vliv měřicích prvků a čidel na přesnost predikce

Stroj byl vybaven řadou měřicích čidel. Podrobný popis je v [26].

Při testování jednotlivých fyzikálních veličin (teplota, vlhkost, osvit) bylo zjištěno, že zásadní vliv na přesnost predikce, mají čidla umístěná ve vřetenu stroje *temp_vr01,...,temp_vr07*, čidla teploty pravítek *X,Y,Z (temp_pr01, temp_pr02 a temp_pr03)* a čidlo teploty hlavního motoru,*temp_S1*. Toto čidlo mělo vliv dominantní. Všechny ostatní měřené veličiny (teploty mimo vřeteno, vlhkost, osvit), měly minimální vliv na přesnost predikce a byly z tréninkových dat vyloučeny.

9.5 Návrh neuronové sítě

Prvotním záměrem bylo naučit síť predikovat odchylky ve všech třech osách naráz (tedy simulovat sítí celý 3D prostor). V tomto případě však výsledky nebyly příliš průkazné, proto bylo rozhodnuto, že budou natrénovány tři sítě, pro každou osu zvlášť s tím, že každé ze sítí (X,Y,Z) se předloží teploty ve vřeteni stejnou měrou. Rozdíl však bude jen v pravítku osy. To znamená síti X se předloží teplota prvítka x, atd...

Je jasné, že trénink tří sítí je časově mnohem více náročná úloha. Po optimalizaci a návrhu parametrů sítí pro každou osu zvlášť jsme se vrátili k původnímu návrhu 3D sítě, která za použití optimalizovaných parametrů vykazuje stejné vlastnosti (MAE MSE) jako sítě, pro každou osu zvlášť.

Pro testy byly zvoleny tři typy sítí, klasická DENSE, která posloužila pro prvotní náhledy do problematiky, díky své nenáročnosti na systémové zdroje. Následně byly testovány rekurentní sítě typu LSTM a GRU s těmito poznatky:

- Síť typu DENSE klade nejnižší nároky na systémové zdroje, proces učení je řádově v minutách. Nejlepších výsledků bylo dosaženo v konfiguraci

2 hidden (DENSE, 71, activation=elu, initializer=he_normal, optimizer=adam). Počet ladicích epoch byl zkoušen v intervalu (64, 128). Díky typu síte bylo samozřejmě dosaženo nejhorších výsledků, co se týče parametru MAE a MSE. Co se týká snahy o navýšení přesnosti, bylo při navyšování počtu učebních epoch, shledáno maximum v podobě $epoch = 128$. Pak již síť měla tendenci k přeučování. (Začala se zhoršovat ztrátová funkce $J(y)$.

Ukazuje se však, že pokud má být systém nasazen na embedded zařízení, bude patrně síť typu DENSE jediná přijatelná možnost. Zlepšení bychom mohli docílit změnou parametrů aktivační funkce a optimálního počtu neuronů v jednotlivých vrstvách modelu sítě.

- Síť typu LSTM byla zkoušena v konfiguraci 3 hidden (LSTM, 512), přičemž aktivační funkce nebyla definována. Byla ponechána implicitní, která je součástí konfigurace LSTM. Každá z vrstev byla proložena vrstvou $Dropout = 0.2$, pro zvýšení propustnosti modelu. Jedná se o síť při níž bylo dosaženo překvapivě nejhorších výsledků hodnot MAE a MSE. V souvislosti s nároky na hardware ve srovnání se sítí DENSE, bylo prozatím konstatováno, že tento typ sítě není perspektivní. Špatný výsledek lze samozřejmě přičíst n nedostatečnému testování a nastavení jednotlivých parametrů, díky časové náročnosti. Po získání GPU se k testování této sítě ještě vrátíme, abychom vylepšili jak parametry tak i nastavení LSTM sítě, od níž se předpokládá, že bude dávat nejlepší výsledky.
- Síť typu GRU byla zkoušena opět v konfiguraci 3 hidden (GRU, 512), a opět byla aktivační funkce ponechána implicitně. Každá z vrstev byla proložena vrstvou $Dropout = 0.2$.. Z hlediska MAE a MSE přinesla nejlepší výsledky a v podstatě splnila požadavek na přesnost predikce v řádu (± 5) [μm].

9.6 Optimální parametry sítě

Při provádění zkoušek byla zvolena síť typu DENSE, která vykazovala parametry, které se více méně shodovaly s požadovanými vlastnostmi predikovaných a naměřených hodnot. Ú sítí typu LSTM nebo GRU by mohlo být dosaženo ještě lepších výsledků, ale vzhledem k potřebě rychlého doučováni sítě za provozu stroje, nebyly z důvosu výpočetní náročnosti použity.

V tabulkách 9.1, 9.2 a 9.3 jsou uvedeny parametry neuronové sítě, které se podílejí na celkovém nasavení a optimalizaci tréninkového procesu.

9.6.1 Vstupní parametry pro proces predikce sítě

Tyto naměřené hodnoty jsou předkládány síti v procesu predikce. Na základě těchto parametrů predikuje neuronová síť teplotní dilatace v osách (X, Y, Z), které jsou následně předkládány řídicímu systému ke korekci.

Měřená veličina $[°C]$	**Název**
temp_lo03	Teplota lože 3
temp_st02	Teplota stojanu 2
temp_st06	Teplota stojanu 6
temp_st07	Teplota stojanu 7
temp_S1	Teplota motoru Z
temp_pr01	Teplota pravítka X
temp_pr02	Teplota pravítka Y
temp_pr03	Teplota pravítka Z
temp_vr01	Teplota vřetene 1
temp_vr02	Teplota vřetene 2
temp_vr03	Teplota vřetene 3
temp_vr04	Teplota vřetene 4
temp_vr05	Teplota vřetene 5
temp_vr06	Teplota vřetene 6
temp_vr07	Teplota vřetene 7

Tabulka 9.1: *Vstupní proměnné pro predikci*

9.6.2 Vstupní parametry pro proces učení sítě

Tyto naměřené hodnoty jsou předkládány síti v procesu učení. Na základě těchto parametrů se snaží neuronová síť minimalizovat ztrátovou funkci $J(y)$. Proces učení nastaví vnitřní parametry sítě tak, aby co nejlépe predikovala teplotní vlivy na přesnost procesu obrábění. Vstupní parametry v oblasti teplotních vlivů v procesu učení jsou totožné s parametry v procesu predikce.

Přidány jsou měřené parametry odchylek přesnosti v osách X, Y, Z. Tyto odchylky jsou měřeny s vysokou přesností (řádově $\pm 1\mu m$) a jsou síti předkládány jen v procesu učení. V normálním provozu již tyto odchylky nejsou měřeny (časově velmi náročné a navíc jsou použity velmi drahé sondy, které by v běžném provozu vemi trpěly). Úkolem neuronové sítě je predikce těchto hodnot. Výsledkem je tedy výrazná úspora nákladů a zvýšení produktivity, vyloučením neproduktivního času nutného k měření odchylek. Výsledky z výstupu neuronové sítě (predikované dilatace v osách X, Y, Z) jsou následně předkládány řídicímu systému ke korekci.

Měřená veličina $[°C]$, $[\mu m]$	**Název**
dev_x4	Artefakt X 4 (osa X čtvrtá poloha)
dev_y4	Artefakt Y 4 (osa Y čtvrtá poloha)
dev_z4	Artefakt Z 4 (osa Z čtvrtá poloha)
temp_lo03	Teplota lože 3
temp_st02	Teplota stojanu 2
temp_st06	Teplota stojanu 6
temp_st07	Teplota stojanu 7
temp_S1	Teplota motoru Z
temp_pr01	Teplota pravítka X
temp_pr02	Teplota pravítka Y
temp_pr03	Teplota pravítka Z
temp_vr01	Teplota vřetene 1
temp_vr02	Teplota vřetene 2
temp_vr03	Teplota vřetene 3
temp_vr04	Teplota vřetene 4
temp_vr05	Teplota vřetene 5
temp_vr06	Teplota vřetene 6
temp_vr07	Teplota vřetene 7

Tabulka 9.2: *Vstupní proměnné pro učení sítě*

9.6.3 Hyperparametry sítě

Hyperparametry, jsou proměnné, kterými se nastavují vnitřní vlastnosti neuronové sítě. Neexistuje žádná kuchařka, která by podala návod, jak které parametry nastavit.

Jediná cesta, jak optimálně nastavit fungující neuronovou síť je metoda pokusů a omylů a následné iterace k lepším výsledkům. Jedná se o poměrně zdlouhavou činnost, která je přímo závislá na zkušenostech toho, kdo tuto síť ladí.

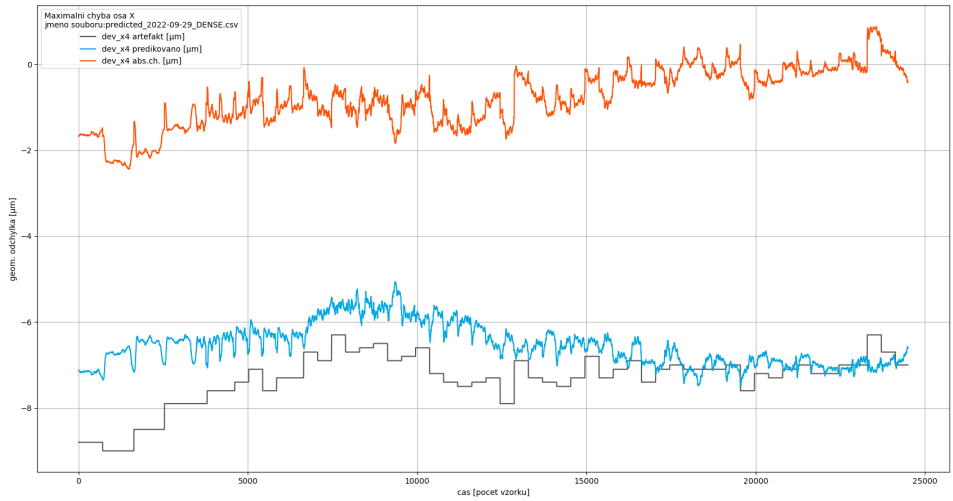
Název hyperparametru	**Hodnota**
parm : **"type:"**	val : **"DENSE"**
parm : **"units"**	val : **"71"**
parm : **"act_function"**	val : **"ELU"**
parm : **"kernel_initializer:"**	val : **"RandomUniform object"**
parm : "use_bias"	val : "False"
parm : "bias_initializer"	val : "zeros"
parm : "kernel_regularizer"	val : "None"
parm : "bias_regularizer"	val : "None"
parm : "activity_regularizer"	val : "None"
parm : "kernel_constraint"	val : "None"
parm : "bias_constraint"	val : "None"
parm : "layers_count"	val : "2"
parm : "loss"	val : "mse"
parm : **"optimizer"**	val : **"adam"**
parm : **"metrics"**	val : **"['mse', 'acc']"**
parm : "loss_weights"	val : "None"
parm : "sample_weight_mode"	val : "None"
parm : "weighted_metrics"	val : "None"
parm : "target_tensors"	val : "None"
parm : **"batch_size"**	val : **"128"**
parm : **"epochs"**	val : **"52"**
parm : **"verbose"**	val : **"0"**
parm : "callbacks"	val : "None"
parm : "validation_split"	val : "0.0"
parm : **"shuffle"**	val : **"True"**
parm : "class_weight"	val : "None"
parm : "sample_weight"	val : "None"
parm : "initial_epoch"	val : "0"
parm : "steps_per_epoch"	val : "None"
parm : "validation_steps"	val : "None"
parm : "validation_batch_size"	val : "None"
parm : "validation_freq"	val : "1"
parm : "max_queue_size"	val : "1"
parm : "workers"	val : "1"
parm : "use_multiprocessing"	val : "False"

Tabulka 9.3: *Hyperparametry sítě typu DENSE*

9.7 Závislosti predikcí na naměřených hodnotách.

Na obrázcích 9.5,9.6 a 9.7 jsou znázorněny průběhy skutečně naměřených hodnot (černá barva grafu), predikce (modrá barva) a absolutní chyby (oranžová barva). Zde je vidět poměrně velmi dobrá přesnost predikce vůči reálně naměřeným hodnotám, která u každé z predikovaných os má relatvně dobrou odchylku od skutečnosti v řádu jednotek mikrometrů.

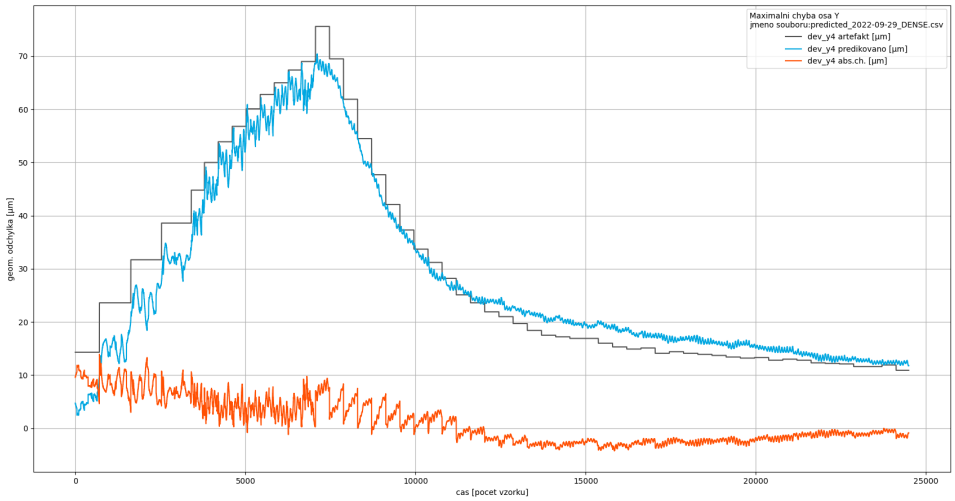
Na obrázku 9.8 je znázorněna maximální chyba, která se vyskytla při predikci v rámci 128 vzorků. Jinými slovy, pro každých 128 vzorků predikovaných dat, byla prezentována maximální chyba predikce vůči změřeným hodnotám.



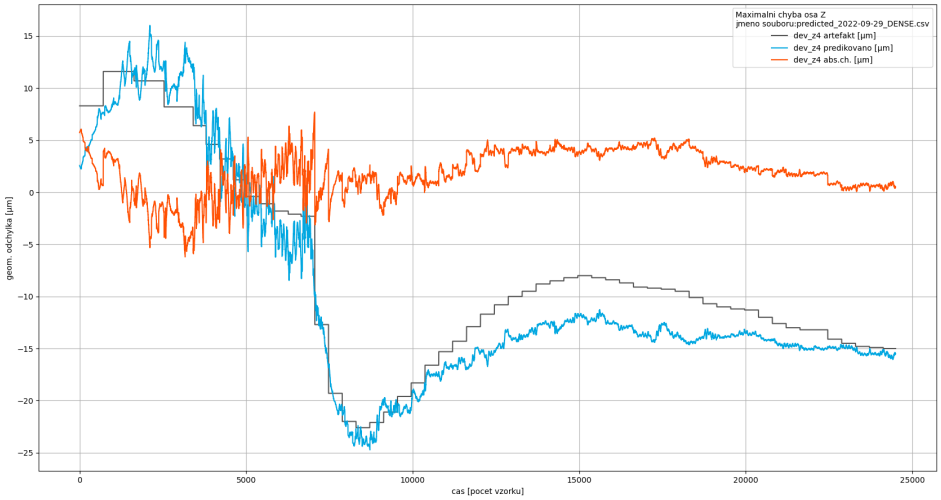
Obrázek 9.5: *Výsledek predikce pro osu X (měřeno v µm) - síť typu DENSE*

9.7.1 Vliv objemu dat na přesnost predikce

Kvalita učení sítě závisí nejen na množství dat, předkládaných k učebnímu procesu (minimalizaci nákladové funkce $J(y)$, ale i jejich kvalita. Data by měla obsahovat pokud možno všechny obvyklé provozní stavy sledované soustavy. Obecně se ukazuje, že kvalita dat je významnější ukazatel, než jejich objem.

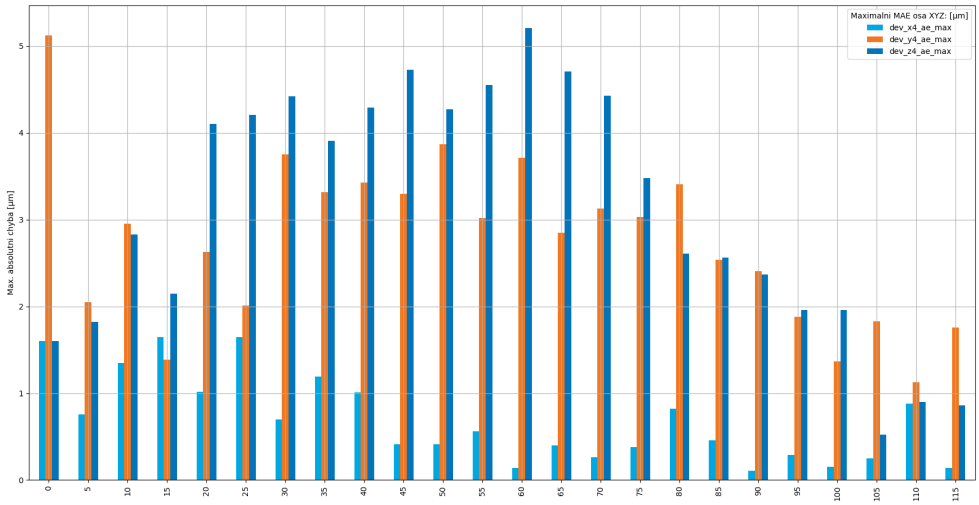


Obrázek 9.6: *Výsledek predikce pro osu Y (měřeno v µm) - síť typu DENSE*



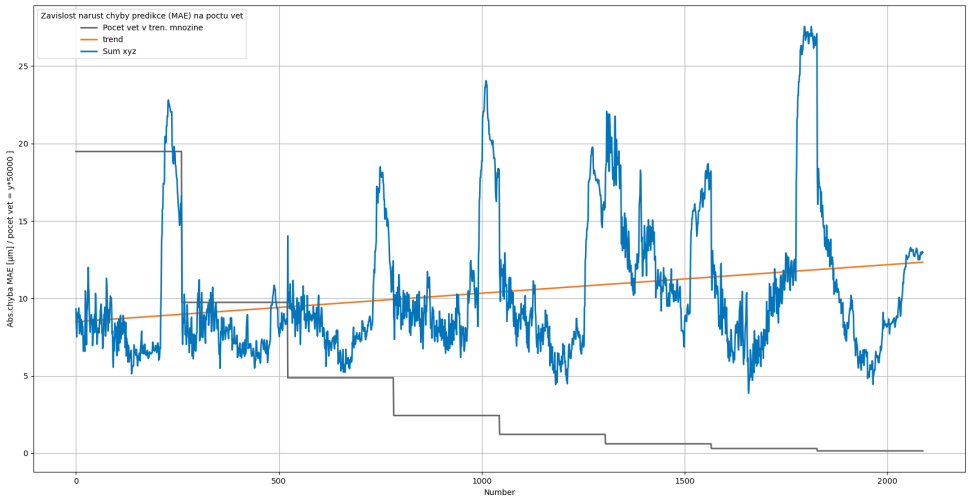
Obrázek 9.7: *Výsledek predikce pro osu Z (měřeno v µm) - síť typu DENSE*

Bylo provedeno měření přesnosti optimalizace v závislosti na objemu měřených dat. Z celkového množství dat byla vždy vybrána každá n-tá věta, kdy $n = (1, 2, 4, 8, 16, 32, 64, 128)$. To znamená, že do procesu učení byla připravena data v celém objemu (cca 1 000 000 vět), následně jejich polovina, čtvrtina atd... dle předpisu $n = (1, 1/2, 1/4, ..., 1/128)$. Byl zkoumán trend nárůstu ne-



Obrázek 9.8: *Vyjádření maximální chyby (MAE) pro osy X,Y a Z (měřeno v µm) - síť typu DENSE*

přesnosti predikce v souvislosti na snižujícím se počtu vět. Touto metodou byla vybírána data z celého objemu dat tak, že jednotlivé vzorky dat byly pravidelně rozprostřeny v časové ose. To znamená, že data obsahovala většinu provozních stavů, které vznikly v průběhu celého časového úseku provozu stroje.9.9



Obrázek 9.9: *Trend vývoje chyby predikce na objemu dat, předkládaných k učení*

Z obrázku 9.9 plyne, že nepřesnost predikce roste poměrně málo. To potvrzuje domněnku, že kvalita dat má zásadní vliv na přesnosti učení. Z grafu lze vyčíst, že data redukovaná na 1/8 původního objemu zvýší nepřesnost predikce o cca. 3 μm.

9.8 Popis digitálního dvojčete

Program pro digitální dvojče byl napsán v Pythonu s využitím knihoven Tensorflow (C) Google a Keras (C) Francois Chollet, které poskytují základní prostředí pro definici neuronových sítí. Jedná se o knihovny uvolněné v Open Source licenci, takže jejich použití není vázáno na uzavřenou licenční politiku. V současné době jsou tyto knihovny využívány ke strojovému učení ve více jak 80% všech aplikací. Jejich výhodou je skvělá dokumentace a spousta příkladů, které mohou snáze prošlápnout cestu v této poměrně komplikované disciplíně.

Níže jsou uvedeny a komentovány nejdůležitější části programu, jehož úplná verze je umístěna na GITHUBu: `https://github.com/lukasik-petr/ai-neuro`.

9.8.1 Konektivita s PLC stroje

Pro konekt do PLC stroje je využita knihovna *opcua* `https://open62541.org/doc/current/`, která je publikována v open source licenci a zastřešována Fraunhoferovým Institutem. V níže uvedené části je znázorněn princip komunikace stroje s digitálním dvojčetem.

Ze stroje jsou (s pomocí PLC a opcua) vyčítány hodnoty teplot a hodnoty změřených geometrických odchylek způsobených teplotními dilatacemi v rámci konstrukce stroje. Ty jsou následně předkládány k tréninku sítě.

V okamžiku, kdy je neuronová síť dostatečně natrénována, jsou z PCL vyčítány jen teploty, které jsou následně předloženy síti, pro predikci teplotních dilatací.

Příklad 1.

Listing 9.1: *ai-daemon*

```
1
2  #------------------------------------------------------------------------
3  # opcCollectorBR_PLC - opc server BR
4  #------------------------------------------------------------------------
5      def opcCollectorBR_PLC(self):
6
7          plc_isRunning = True;
8          # tabulka nodu v br plc
9          plc_br_table        = np.array([["temp_ch01",     "ns=6;s=::AsGlobalPV:
       teplota_ch01"],
10                                         ["temp_lo01",     "ns=6;s=::AsGlobalPV:
       teplota_lo01"],
11                                         ["temp_lo03",     "ns=6;s=::AsGlobalPV:
       teplota_lo03"],
12                                         ["temp_po01",     "ns=6;s=::AsGlobalPV:
       teplota_po01"],
13                                         ["temp_pr01",     "ns=6;s=::AsGlobalPV:
       teplota_pr01"],
14                                         ["temp_pr02",     "ns=6;s=::AsGlobalPV:
       teplota_pr02"],
15                                         ["temp_pr03",     "ns=6;s=::AsGlobalPV:
       teplota_pr03"],
16                                         ["temp_sl01",     "ns=6;s=::AsGlobalPV:
       teplota_sl01"],
17                                         ["temp_sl02",     "ns=6;s=::AsGlobalPV:
       teplota_sl02"],
18                                         ["temp_sl03",     "ns=6;s=::AsGlobalPV:
       teplota_sl03"],
19                                         ["temp_sl04",     "ns=6;s=::AsGlobalPV:
       teplota_sl04"],
20                                         ["temp_st01",     "ns=6;s=::AsGlobalPV:
       teplota_st01"],
21                                         ["temp_st02",     "ns=6;s=::AsGlobalPV:
       teplota_st02"],
```

```
22                                          ["temp_st03",     "ns=6;s=::AsGlobalPV:
    teplota_st03"],
23                                          ["temp_st04",     "ns=6;s=::AsGlobalPV:
    teplota_st04"],
24                                          ["temp_st05",     "ns=6;s=::AsGlobalPV:
    teplota_st05"],
25                                          ["temp_st06",     "ns=6;s=::AsGlobalPV:
    teplota_st06"],
26                                          ["temp_st07",     "ns=6;s=::AsGlobalPV:
    teplota_st07"],
27                                          ["temp_st08",     "ns=6;s=::AsGlobalPV:
    teplota_st08"],
28                                          ["temp_vr01",     "ns=6;s=::AsGlobalPV:
    teplota_vr01"],
29                                          ["temp_vr02",     "ns=6;s=::AsGlobalPV:
    teplota_vr02"],
30                                          ["temp_vr03",     "ns=6;s=::AsGlobalPV:
    teplota_vr03"],
31                                          ["temp_vr04",     "ns=6;s=::AsGlobalPV:
    teplota_vr04"],
32                                          ["temp_vr05",     "ns=6;s=::AsGlobalPV:
    teplota_vr05"],
33                                          ["temp_vr06",     "ns=6;s=::AsGlobalPV:
    teplota_vr06"],
34                                          ["temp_vr07",     "ns=6;s=::AsGlobalPV:
    teplota_vr07"],
35                                          ["temp_vz02",     "ns=6;s=::AsGlobalPV:
    teplota_vz02"],
36                                          ["temp_vz03",     "ns=6;s=::AsGlobalPV:
    teplota_vz03"],
37                                          ["light_ambient", "ns=6;s=::AsGlobalPV:
    vstup_osvit"],
38                                          ["temp_ambient",  "ns=6;s=::AsGlobalPV:
    vstup_teplota"],
39                                          ["humid_ambient", "ns=6;s=::AsGlobalPV:
    vstup_vlhkost"]]);
40
41          if not self.pingSocket(self.host1, self.port1):
42              plc_isRunning = False;
43              return(plc_br_table, plc_isRunning);
44
45          client = Client(self.uri1)
46          try:
47              client.connect();
48              self.logger.debug("Client: " + self.uri1 + " connect.....")
49              plc_br_table = np.c_[plc_br_table, np.zeros(len(plc_br_table))];
50
51              for i in range(len(plc_br_table)):
52                  node = client.get_node(str(plc_br_table[i, 1]));
53                  typ  = type(node.get_value());
54                  val = float(self.myFloatFormat(node.get_value())) if typ is float
    else node.get_value();
55                  #val = node.get_value() if typ is float else node.get_value();
56                  plc_br_table[i, 2] = val;
57
58          except Exception as ex:
59              traceback.print_exc();
60              self.logger.error(traceback.print_exc());
61              plc_isRunning = False;
62
63          finally:
64              client.disconnect();
```

```
65              return(plc_br_table, plc_isRunning);
```

V cyklu jsou zde načítány jednotlivé hodnoty OPC uzlů a vkládány do tabulky /textitplc_br_table. Takto je vytvořena jedna věta, která se zapisuje do matice o rozměrech $len(br\_plc\_table)x128$ která je po naplnění 128 vzorky, následně posána síti k predikci.

9.8.2 Příprava dat

Předkládaná data se musí před tím, než jsou předložena síti, neprve upravit. U sítě DENSE je situace snažší v tom, že DENSE nevyžaduje definici časových oken, takže matice měřených dat může být síti předložena přímo. U sítě GRU nebo LSTM je situace poněkud složitější, protože tyto sítě vyžadují, aby byl k predikci přidán ještě tzv časový vývoj řady. Časový vývoj nikoliv ve smyslu času ale ve smyslu směru vývoje datových struktur. Proto se musí definovat min. 3D tenzor (pro 2D) data, který je o časový vývoj obohacen. V praxi se jedná o definici tzv. časového okna, které se v datech posouvá a vytváří tak prostorová data. Nevýhodou je, že objem dat takto enormně naroste. To násleně také odpovídá spolu s komplikovanějšími algoritmy GRU a LSTM, výrazně delšímu času potřebnému k natrénování sítě.

Příklad 2.

Listing 9.2: *ai-daemon*

```
1  #------------------------------------------------------------------------
2  # toTensorLSTM(self, dataset, window = 64):
3  #------------------------------------------------------------------------
4  # Pracujeme - li s rekurentnimi sitemi (LSTM GRU...), pak
5  # musíme vygenerovat dataset ve specifickém formátu.
6  # Vystupem je 3D tenzor ve forme 'window' casovych kroku.
7  #
8  # Jakmile jsou data řvytvoena ve ěform 'window' časových úkrok,
9  # jsou nasledne prevedena do pole NumPy a reshapovana na
10 # pole 3D X_dataset.
11 #
12 # Funkce take vyrobi pole y_dataset, ktere muze byt pouzito pro
13 # simulaci modelu vstupnich dat, pokud tato data nejsou k dispozici.
14 # y_dataset predstavuje "window" časových úrámc krat prvni prvek casoveho
15 # ramce pole X_dataset
16 #
17 # funkce vraci: X_dataset - 3D tenzor dat pro uceni site
18 #               y_dataset - vektor vstupnich dat (model)
19 #               dataset_cols - pocet sloupcu v datove sade.
```

```
#
# poznamka: na konec tenzoru se pripoji libovolne 'okno' aby se velikost
#           o toto okno zvetsila - vyresi se tim chybejici okno pri predikci
#
#----------------------------------------------------------------------

    def toTensorLSTM(dataset, window):

        X_dataset = []  #data pro tf.fit(x - data pro uceni
        y_dataset = []  #data pro tf.fit(y - vstupni data
                            #jen v pripade ze vst. data nejsou definovana

        values = dataset[0 : window, ];
        dataset = np.append(dataset, values, axis=0) #pridej delku okna
        dataset_rows, dataset_cols = dataset.shape;


        if window >= dataset_rows:
            self.logger.error("prilis maly  vektor dat k uceni!!! \nparametr
    window je vetsi nez delka vst. vektoru ");
            return(None);

        for i in range(window, dataset_rows):
            X_dataset.append(dataset[i - window : i, ]);
            y_dataset.append(dataset[i, ]);

        #doplnek pro append chybejicich window vzorku pri predikci
        X_dataset.append(dataset[0 : window, ]);

        X_dataset = np.array(X_dataset);
        y_dataset = np.array(y_dataset);

        X_dataset = np.reshape(X_dataset, (X_dataset.shape[0], X_dataset.shape[1],
     dataset_cols));

        return NeuronLayerLSTM.DataSet(X_dataset, y_dataset, dataset_cols);
```

Následně se musí tato data po predikci znovu překódovat do srozumitelného tvaru, to znamená, že je třeba časový vývoj po predikci z dat odstranit. to je patrné z jednoduché pythonovské funkce, která sníží řád 3D tenzoru na námi obvyklou 2D matici, která je předmětem výstupu z predikce.

Příklad 3.

Listing 9.3: *ai-daemon*

```
řěčůčů
#----------------------------------------------------------------------
# fromTensorLSTM(self, dataset, window = 64):
#----------------------------------------------------------------------
# Poskladej vysledek vzdy z posledniho behu treninkove sady
# a vrat vysledek o rozmeru [0: (dataset.shape[0] - 1)] krat [0 : dataset.shape
    [2]]
# priklad: ma li tenzor rozmer 100 x 64 x 16, pak vrat vysledek [0:100-1], 64,
    [0,16-1]
# funkce vraci: y_result - 2D array vysledku predikce
#----------------------------------------------------------------------
```

```
10      def fromTensorLSTM(dataset):
11          return(dataset[0 : (dataset.shape[0]),  (dataset.shape[1] - 1) , 0 :
        dataset.shape[2]]);
```

9.8.3 Definice sítě LSTM

V režimu "train" je volána treninková metoda neuralNetworkLSTMtrain, která načte data do vektorů y_train_data, x_train_data, y_valid_data a x_valid_data. Následně jsou poslána k normalizaci do MinMaxScaleru, který nedělá nic jiného, že všechny vstupní hodnoty normalizuje do množiny o rozsahu hodnot $< -1, +1 >$ a to nezávisle na velikosti vstupních veličin. Jinými slovy měřené teploty v rozsahu cca $(20, 80)°C$ jsou normalizovány do tvaru $< -1, +1 >$ a také veličiny geometrických odchylek, které jsou měřeny v řádu μm, jsou taktéž převedeny do tvaru $< -1, +1 >$. Zdá se to být podivné, ale je třeba si uvědomit, že neuronovou síť zajímají pouze gradienty, nikoliv vlastní naměřené hodnoty. Objekty MinMaxScaleru jsou následně archivovány a zapsány do *./temp/y_train_scaler"+ thread_name+".pkl", 'wb'))*, pro následné renormalizaci predikovaných dat - převedení normalizovaných predikcí do reálného tvaru.

Následuje definice neuronové sítě, která se skládá ze vstupní vrstvy (řádek 49), ze dvou skrytých vrstev LSTM, které jsou proloženy vrstvami Dropout (řádky 51,52,53) a následující výstupní vrstvou typu Dense, která definuje velikost výstupního tenzoru predikovaných dat.

Vrstva Dropout náhodně vyselektuje a vyřadí z výpočtu určité množství dat. To se dělá proto, aby se zabránilo přeučení sítě.

Následuje kompilace definovaného modelu sítě a spustí se trénink (řádky 62-67). Po ukončení tréninku, je kompletní naučený objekt sítě zapsán (řádek 71) k pozdějšímu použití v režimu predict.

Příklad 4.

Listing 9.4: *ai-daemon*

```
1  řěčůčů
2  #------------------------------------------------------------------------
3  # Neuronova Vrstava LSTM
```

```
#-----------------------------------------------------------------------
    def neuralNetworkLSTMtrain(self, DataTrain, thread_name):
        window_X = self.window;
        window_Y =  1;

        try:
            y_train_data = np.array(DataTrain.train[DataTrain.df_parm_y]);
            x_train_data = np.array(DataTrain.train[DataTrain.df_parm_x]);
            y_valid_data = np.array(DataTrain.valid[DataTrain.df_parm_y]);
            x_valid_data = np.array(DataTrain.valid[DataTrain.df_parm_x]);

            inp_size = len(x_train_data[0]);
            out_size = len(y_train_data[0]);

        # normalizace dat k uceni a vstupnich treninkovych dat
            self.x_train_scaler = MinMaxScaler(feature_range=(0, 1))
            x_train_data = self.x_train_scaler.fit_transform(x_train_data)
            pickle.dump(self.x_train_scaler, open("./temp/x_train_scaler"+
    thread_name+".pkl", 'wb'))

            self.y_train_scaler = MinMaxScaler(feature_range=(0, 1))
            y_train_data = self.y_train_scaler.fit_transform(y_train_data)
            pickle.dump(self.y_train_scaler, open("./temp/y_train_scaler"+
    thread_name+".pkl", 'wb'))

        # normalizace dat k uceni a vstupnich treninkovych dat
            self.x_valid_scaler = MinMaxScaler(feature_range=(0, 1))
            x_valid_data = self.x_valid_scaler.fit_transform(x_valid_data)
            pickle.dump(self.x_train_scaler, open("./temp/x_valid_scaler"+
    thread_name+".pkl", 'wb'))

            self.y_valid_scaler = MinMaxScaler(feature_range=(0, 1))
            y_valid_data = self.y_valid_scaler.fit_transform(y_valid_data)
            pickle.dump(self.y_train_scaler, open("./temp/y_valid_scaler"+
    thread_name+".pkl", 'wb'))

        #data pro trenink -3D tenzor
            X_train =  DataFactory.toTensorLSTM(x_train_data, window=window_X);
        #vstupni data train
            Y_train = DataFactory.toTensorLSTM(y_train_data, window=window_Y);
            Y_train.X_dataset = Y_train.X_dataset[0 : X_train.X_dataset.shape[0]];
        #data pro validaci -3D tenzor
            X_valid = DataFactory.toTensorLSTM(x_valid_data, window=window_X);
        #vystupni data pro trenink -3D tenzor
            Y_valid = DataFactory.toTensorLSTM(y_valid_data, window=window_Y);
            Y_valid.X_dataset = Y_valid.X_dataset[0 : X_valid.X_dataset.shape[0]];

        # neuronova sit
            neural_model = Sequential();
            neural_model.add(Input(shape=(X_train.X_dataset.shape[1], X_train.cols
    ,)));
            neural_model.add(LSTM(units = self.units, return_sequences=True));
            neural_model.add(Dropout(0.2));
            neural_model.add(LSTM(units = self.units, return_sequences=True));
        #   neural_model.add(Dropout(0.2));
        #   neural_model.add(LSTM(units = self.units, return_sequences=True));
            neural_model.add(layers.Dense(Y_train.cols, activation='relu'));

        # definice ztratove funkce a optimalizacniho algoritmu
            neural_model.compile(loss='mse', optimizer='adam', metrics=['mse', '
    acc']);
        # natrenuj neural_model na vstupni dataset
```

```
60              history = neural_model.fit(X_train.X_dataset,
61                                         Y_train.X_dataset,
62                                         epochs=self.epochs,
63                                         batch_size=self.batch,
64                                         verbose=2,
65                                         validation_data=(X_valid.X_dataset,
66                                                          Y_valid.X_dataset)
67                                 );
68
69
70          # zapis neural_modelu
71              neural_model.save("./models/model_"+thread_name+"_"+DataTrain.axis,
    overwrite=True, include_optimizer=True)
72
73              # make predictions for the input data
74              self.neural_model = neural_model;
75              return ();
76
77          except Exception as ex:
78              self.logger.error(traceback.print_exc());
```

9.8.4 Definice sítě DENSE

U sítě DENSE je situace prakticky totožná s definicí sítě LSTM. V režimu "train" je načtena metoda neuralNetworkDENSEtrain, která se skládá ze stejných kroků jako neuralNetworkLSTMtrain. Nejsou však použity vrstvy Dropout, které jsou u tohoto jednoduchého typu nevýznamné.

Příklad 5.

Listing 9.5: *ai-daemon*

```
1   řěčůčů
2   #------------------------------------------------------------------------
3   # Neuronova Vrstava DENSE
4   #------------------------------------------------------------------------
5       def neuralNetworkDENSEtrain(self, DataTrain, thread_name):
6
7           try:
8
9               y_train_data = np.array(DataTrain.train[DataTrain.df_parm_y]);
10              x_train_data = np.array(DataTrain.train[DataTrain.df_parm_x]);
11              y_valid_data = np.array(DataTrain.valid[DataTrain.df_parm_y]);
12              x_valid_data = np.array(DataTrain.valid[DataTrain.df_parm_x]);
13
14              if (x_train_data.size == 0 or y_train_data.size == 0):
15                  return();
16
17              inp_size = len(x_train_data[0])
18              out_size = len(y_train_data[0])
19
20          # normalizace dat k uceni a vstupnich treninkovych dat
21              self.x_train_scaler = MinMaxScaler(feature_range=(0, 1))
22              x_train_data = self.x_train_scaler.fit_transform(x_train_data)
```

```
23              pickle.dump(self.x_train_scaler, open("./temp/x_train_scaler"+
    thread_name+".pkl", 'wb'))
24
25              self.y_train_scaler = MinMaxScaler(feature_range=(0, 1))
26              y_train_data = self.y_train_scaler.fit_transform(y_train_data)
27              pickle.dump(self.y_train_scaler, open("./temp/y_train_scaler"+
    thread_name+".pkl", 'wb'))
28
29          # normalizace dat k uceni a vstupnich treninkovych dat
30              self.x_valid_scaler = MinMaxScaler(feature_range=(0, 1))
31              x_valid_data = self.x_valid_scaler.fit_transform(x_valid_data)
32              pickle.dump(self.x_train_scaler, open("./temp/x_valid_scaler"+
    thread_name+".pkl", 'wb'))
33
34              self.y_valid_scaler = MinMaxScaler(feature_range=(0, 1))
35              y_valid_data = self.y_valid_scaler.fit_transform(y_valid_data)
36              pickle.dump(self.y_train_scaler, open("./temp/y_valid_scaler"+
    thread_name+".pkl", 'wb'))
37
38
39          # neuronova sit
40              neural_model = Sequential();
41              initializer = tf.keras.initializers.RandomUniform(minval=-0.05, maxval
    =0.05, seed=None)
42              neural_model.add(tf.keras.Input(shape=(inp_size,)));
43              neural_model.add(layers.Dense(units=inp_size,       activation=self.
    actf, kernel_initializer=initializer));
44              neural_model.add(layers.Dense(units=self.units,     activation=self.
    actf, kernel_initializer=initializer));
45          #   neural_model.add(Dropout(0.2));
46              neural_model.add(layers.Dense(units=self.units,     activation=self.
    actf, kernel_initializer=initializer));
47          #   neural_model.add(Dropout(0.2));
48              neural_model.add(layers.Dense(out_size));
49
50          # definice ztratove funkce a optimalizacniho algoritmu
51              neural_model.compile(loss='mse', optimizer='adam', metrics=['mse', '
    acc'])
52
53          # natrenuj neural_model na vstupni dataset
54              history = neural_model.fit(x_train_data,
55                                         y_train_data,
56                                         epochs=self.epochs,
57                                         batch_size=self.batch,
58                                         verbose=2,
59                                         validation_data=(x_valid_data, y_valid_data
    )
60                              );
61
62              neural_model.save("./models/model_"+thread_name+"_"+DataTrain.axis,
    overwrite=True, include_optimizer=True)
63
64              self.neural_model = neural_model;
65
66          # make predictions for the input data
67              return ();
68
69
70          except Exception as ex:
71              traceback.print_exc();
72              self.logger.error(traceback.print_exc());
```

9.8.5 Predikce

V režimu "predict" je načtena metoda neuralNetworkLSTMpredict, která má za úkol dodat síti predikční množinu dat (v našem případě změřené teploty) a poslat je k predikci natrénované síti (viz řádek 20). Po predikci dochází k denormalizaci dat (řádek 25) s pomocí objektů scaleru, které byly archivovány (a následně načteny, viz řádky 12 a 13) při tréninkovém běhu sítě.

Příklad 6.

Listing 9.6: *ai-daemon*

```
1  řěčůčů
2  #------------------------------------------------------------------------
3  # Neuronova Vrstava LSTM predict
4  #------------------------------------------------------------------------
5      def neuralNetworkLSTMpredict(self, DataTrain, thread_name):
6
7          try:
8              axis     = DataTrain.axis;
9              x_test   = np.array(DataTrain.test[DataTrain.df_parm_x]);
10             y_test   = np.array(DataTrain.test[DataTrain.df_parm_y]);
11
12             self.x_train_scaler =  pickle.load(open("./temp/x_valid_scaler"+
       thread_name+".pkl", 'rb'))
13             self.y_train_scaler =  pickle.load(open("./temp/y_valid_scaler"+
       thread_name+".pkl", 'rb'))
14
15             x_test        = self.x_train_scaler.transform(x_test);
16
17             x_object      = DataFactory.toTensorLSTM(x_test, window=self.window);
18             dataset_rows, dataset_cols = x_test.shape;
19         # predict
20             y_result      = self.neural_model.predict(x_object.X_dataset);
21
22         # reshape 3d na 2d
23         # vezmi (y_result.shape[1] - 1) - posledni ramec vysledku - nejlepsi mse i
        mae
24             y_result      = y_result[0 : (y_result.shape[0] - 1),  (y_result.shape
       [1] - 1) , 0 : y_result.shape[2]];
25             y_result      = self.y_train_scaler.inverse_transform(y_result);
26
27         # plot grafu compare...
28             #model.summary()
29
30             return DataFactory.DataResult(x_test, y_test, y_result, axis)
31
32         except Exception as ex:
33             self.logger.error("POZOR !!! patrne se neshoduji predkladana data s
       natrenovanym modelem");
34             self.logger.error("          zkuste nejdrive --typ == train !!!");
35             self.logger.error(traceback.print_exc());
```

Vše ostatní jsou běžné metody a objekty pro manipulaci s datovými strukturami úlohy. Úplné zdrojové kódy je možno získat na `https://github.com/lukasik-petr/ai-neuro`

10 Prognóza časových řad - zdroj učení pod dohledem

10.1 Strojové učení pod dohledem

Většina strojového učení využívá principu učení pod dohledem. V praxi to znamená, že neuronové síti je předložena sada vzorových dat, které představují množinu informací, které jsou předmětem tréninku sítě. To znamená, že na základě této množiny se neuronová síť získá požadované vlastnosti. Síti je tedy předložena vstupní množina dat x a algoritmus *f(x)*, který představuje předpis zobrazení vstupu na výstup y (mapovací funkci). Proces učení lze popsat tímto jednoduchým předpisem:

$$y = f(X)$$

Cílem učení je aproximovat požadovanou funkční závislost tak, že pro vstupní množinu dat, lze předpovědět vlastnosti výstupní proměnné. Výhodou učení neuronové sítě je aproximace souvislostí k nimž je velmi obtížné získat přesnou matematickou formulaci, případně matematickou formulaci odvodit až na základě výsledků dat předložených neuronovou sítí. Příkladem mohou být například úlohy předpovědi počasí, předpovědi vývoje společenosti, chování společnosti v krizových situacích a podobně.

Příkladem datového souboru pro učení pod dohledem, může být soubor , kde každý řádek je výsledkem pozorování (měření) závislostí vstupní proměnné x na hodnotě proměnné y, která má být predikována.

čas vzorek	teplota[℃]	rozměr[µm]
1	21,10	1001
2	21,10	1001
3	21,11	1002
4	21,12	1002
5	21,11	1002
6	21,10	1001
7	21,15	1003
8	21,20	1004
9	21,18	1003

Tabulka 10.1: *Příklad časové řady - závislost roztažnosti na teplotě ok olí*

Časová řada je posloupnost čísel, která jsou uspořádána podle časového indexu. To si lze představit jako seznam nebo sloupec uspořádaných hodnot.

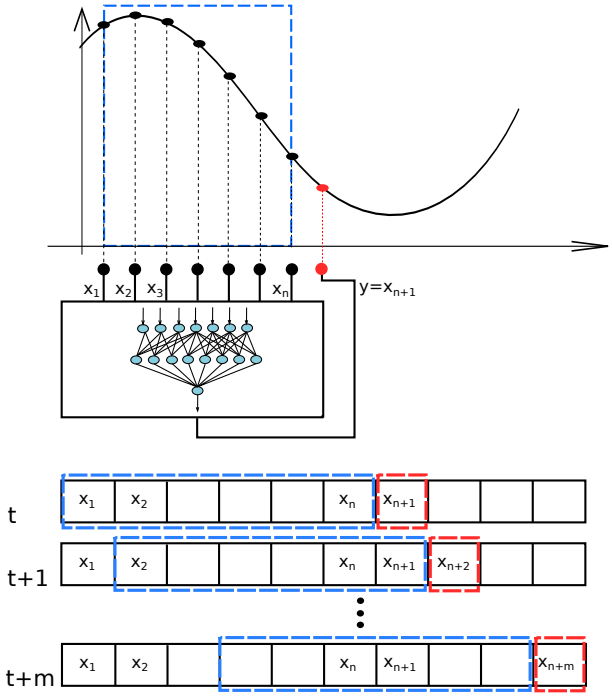
Tento princip se nazývá učení pod dohledem, protože proces učení algoritmu z trénovací datové sady lze považovat za učitele dohlížejícího na proces učení.

Známe správné odpovědi; algoritmus iterativně vytváří předpovědi na trénovacích datech a je opravován prováděním aktualizací. Učení se zastaví, když algoritmus dosáhne přijatelné úrovně výkonu.

Problémy s kontrolovaným učením lze dále seskupit do regresních a klasifikačních problémů.

- **Klasifikační problém:** nastává, když výstupní proměnnou je kategorie, například „červená“ a „modrá“ nebo „vypnuto“, a „zapnuto“. Typickým klasifikačním problémem může být například vyhodnocení recenzí, nebo test plagiátorství a podobné. Velmi důležitým mezikrokem, je tzv. vektorizace dat, což znamená převod slovních souvislostí do řeči čísel. V praxi se může jednat o velmi komplikovaný problém.
- **Regresní problém:** nastává, když výstupní proměnná je skutečná, například změřená hodnota, „rychlost“, „hmotnost“. Výše uvedený vymyšlený příklad závislosti roztažnosti materiálu na teplotě 10.1 je typický regresní problém.

10.2 Posuvné okno (Sliding Window)



Obrázek 10.1: *Posuvné okno - princip*

Vzhledem k posloupnosti čísel pro datovou sadu časových řad můžeme data restrukturalizovat tak, aby vypadala jako problém učení pod dohledem. Můžeme to udělat tak, že použijeme předchozí časové kroky jako vstupní proměnné a další časový krok použijeme jako výstupní proměnnou.

Příklad: máme časovou řadu, kterou jsme získali například měřením:

čas	naměřená hodnota
1	100
2	110
3	108
4	115
5	120

Tabulka 10.2: *Časová řada*

Tuto datovou sadu lze restrukturalizovat jako řízený učební problém použitím hodnoty v předchozím časovém kroku (nebo předchozích časových krocíc)

k předpovědi hodnoty v dalším časovém kroku viz obr 10.1. Pokud by se datový soubor časových řad přeorganizoval tímto způsobem, data by vypadala takto:

X	Y
?	100
100	110
110	108
108	115
115	120
120	?

Tabulka 10.3: *Jednorozměrné posuvné okno*

Podívejme se na výše transformovanou datovou sadu a porovnejme ji s původní časovou řadou. Zde jsou některé postřehy:

- Předchozí časový krok je vstup (X) a další časový krok je výstup (Y) na němž je založen princip problému učení pod dohledem.
- Pořadí mezi pozorováními je zachováno a musí být zachováno i nadále při použití této datové sady k trénování modelu pod dohledem.
- Nemáme žádnou předchozí hodnotu, kterou bychom mohli použít k predikci první hodnoty v sekvenci. Tudíž, tento řádek smažeme, protože jej nemůžeme použít.
- Nemáme známou další hodnotu, kterou bychom předpověděli pro poslední hodnotu v sekvenci. Tudíž, tento řádek smažeme, protože jej nemůžeme použít.

Použití předchozích časových kroků k predikci dalšího časového kroku se nazývá metoda posuvného okna. V některé literatuře může být zkráceně nazývána metodou okna. Ve statistice a analýze časových řad se tomu říká lag nebo lag metoda.

Počet předchozích časových kroků se nazývá šířka okna nebo velikost zpoždění.

Toto posuvné okno je základem toho, jak můžeme z libovolné datové sady časových řad udělat řízený učební problém. Z tohoto jednoduchého příkladu si můžeme všimnout několika věcí:

- princip změny časové řady na regresní nebo klasifikační problém.
- Jakmile je datová sada časové řady připravena tímto způsobem, lze použít jakýkoli ze standardních lineárních a nelineárních algoritmů strojového učení, pokud je zachováno pořadí řádků.
- princip posuvného okna lze použít na časové řadě, která má více než jednu hodnotu, nebo i na tzv. vícerozměrné časové řadě.

10.2.1 Posuvné okno s vícerozměrnými daty časové řady

Rozhodující je počet pozorování zaznamenaných za daný čas v souboru dat časové řady. Teoreticky je tato metoda rozpracována v [**Dietterich-2002**]

- **Jednorozměrné časové řady:** Datové sady, kde je v každém okamžiku pozorována pouze jedna proměnná, měnící se v čase. Například spotřeba elektřiny.
- **Vícerozměrné časové řady:** Datové sady, kde jsou v každém okamžiku pozorovány dvě nebo více proměnných.

Analýza vícerozměrných časových řad bere v úvahu současně více časových řad. Jedná se obecně o mnohem složitější problém než analýza jednorozměrných časových řad a mnoho klasických metod často nefunguje dobře.

Správné místo pro použití strojového učení pro časové řady je tam, kde klasické metody upadají. Níže je uveden další zpracovaný příklad, aby byla metoda posuvného okna konkrétní pro vícerozměrné časové řady.

Předpokládejme, že máme níže vymyšlenou vícerozměrnou datovou sadu časových řad se dvěma pozorováními v každém časovém kroku. Předpokládejme také, že se zabýváme pouze předpovídáním y1(rozměr).

Tuto datovou sadu můžeme přeformulovat jako řízený učební problém s šířkou okna jedna.

čas vzorek	teplota[°C]	rozměr[µm]
1	20.4	1001
2	20.5	1001
3	20.2	1002
4	21.0	1002
5	21.3	1002
6	21.2	1001

Tabulka 10.4: *Vícerozměrná datová řada*

To znamená, že použijeme předchozí hodnoty časového kroku opatření X1 a X2. Budeme mít také k dispozici hodnotu dalšího časového kroku X3. Na základě těchto vstupních hodnot předpovíme další hodnotu časového kroku y1.

Získáme tak 3 vstupní funkce a jednu výstupní hodnotu, kterou budeme předpovídat pro každý tréninkový vzor. Zde je zvolena šířka okna 1 přes tři vstupní hodnoty.[1]

X1	X2	X3	y1
?	?	20.4	1001
20.4	1001	20.5	1001
20.5	1001	20.2	1002
20.2	1002	21.0	1002
21.0	1002	21.3	1002
21.3	1002	21.2	1001
21.2	1001	?	?

Tabulka 10.5: *Vícerozměrná datová řada - předpověď jedné hodnoty y1 ze tří vstupních X1, X2 a X3.*

Vidíme, že stejně jako ve výše uvedeném příkladu jednorozměrné časové řady, možná budeme muset odstranit první a poslední řádek, abychom mohli trénovat náš model učení pod dohledem.

Tento příklad vyvolává otázku, co kdybychom chtěli předpovědět jak y1, tak y2 pro další časový krok?

V tomto případě lze také použít přístup posuvným oknem.

S použitím stejné datové sady časových řad výše to můžeme formulovat jako řízený učební problém, kde předpovídáme jak hodnotu y1, tak hodnotu

[1] Pokud bychom chtěli zvolit šířku okna 2, museli bychom v tomto případě jako vstup vzít šest hodnot. Tabulku hodnot bychom však museli definovat jako sedmisloupcovou.

y2 se stejnou šířkou okna jedna (šířka 1 přes dvě vstupní hodnoty), následovně, přičemž vstupní hodnoty jsou X1 a X2.

X1	X2	y1	y2
?	?	20.4	1001
20.4	1001	20.5	1001
20.5	1001	20.2	1002
20.2	1002	21.0	1002
21.0	1002	21.3	1002
21.3	1002	21.2	1001
21.2	1001	?	?

Tabulka 10.6: *Vícerozměrná datová řada - předpověď dvou hodnot y1 a y2*

Předpovídání více než jedné hodnoty můžeme považovat za predikci sekvence. V tomto případě jsme předpovídali dvě různé výstupní proměnné, ale možná budeme chtít předpovědět více časových kroků před jednou výstupní proměnnou.

Pak použijeme metodu vícekrokové prognózování, které je popsáno v následující části.

10.2.2 Posuvné okno s vícestupňovou předpovědí

Důležitý je počet časových kroků, které je třeba předpovídat.

Opět je tradiční používat pro problém různé názvy v závislosti na počtu časových kroků k předpovědi:

One-Step Forecast: Zde se předpovídá další časový krok (t+1).

Vícekroková předpověď: Zde je třeba předpovědět dva nebo více budoucích časových kroků.

Existuje řada způsobů, jak modelovat vícekrokové prognózování jako problém učení pod dohledem. Prozatím se zaměřujeme na rámování vícekrokové prognózy pomocí metody posuvného okna.

Příklad vícekrokové anakýzy jednorozměrné časové řady z prvního příkladu posuvného okna, viz 10.2.

Tuto časovou řadu můžeme zarámovat jako dvoukrokovou předpovědní datovou sadu pro učení pod dohledem s šířkou okna jedna, následovně:

čas	var1
1	100
2	110
3	108
4	115
5	120

X1	X2	y1
?	100	110
100	110	108
110	108	115
108	115	120
115	120	?
120	?	?

Tabulka 10.7: *Jednorozměrná datová řada - předpověď výstupu y1 ze dvou vstupních hodnot X1 a X2*

Vidíme, že první řadu a poslední dvě řady nelze použít k výcviku modelu pod dohledem. Je to také dobrý příklad, který ukazuje zatížení vstupních proměnných. Konkrétně to, že kontrolovaný model má k práci pouze X1, aby mohl předpovídat jak y1, tak y2. Každý problém vyžaduje pečlivé přemýšlení a experimentování, aby byla nalezena optimální šířka okna, která povede k přijatelnému výkonu modelu.

10.3 Převod časové řady na problém s řízeným učením v Pythonu

Zde bude popsáno:

- Jak vyvinout funkci pro transformaci datové sady časové řady na řízenou učební datovou sadu.
- Jak transformovat jednorozměrná data časové řady pro strojové učení.
- Jak transformovat vícerozměrná data časových řad pro strojové učení.

Časová řada je posloupnost čísel, která jsou uspořádána podle časového indexu. To si lze představit jako seznam nebo sloupec uspořádaných hodnot.

Problém učení pod dohledem se skládá ze vstupních vzorů (X) a výstupních vzorů (y), takže algoritmus se může naučit předpovídat výstupní vzory ze vstupních vzorů.

Například:

X	y
1	2
2	3
3	4
4	5
5	6
6	7
7	8
8	9

Tabulka 10.8: *Jednorozměrná datová řada - předpověď výstupních vzorů ze vstupních hodnot*

10.3.1 Transformace datových řad pythonovskou funkcí *pandas.shift()*

Klíčovou funkcí, která pomáhá transformovat data časových řad na řízený učební problém, je funkce Pandas shift().

Vzhledem k DataFrame lze funkci shift() použít k vytvoření kopií sloupců, které jsou posunuty dopředu (řádky hodnot NaN přidány dopředu) nebo staženy zpět (řádky hodnot NaN přidané na konec).

Toto je chování požadované k vytvoření sloupců pozorování zpoždění a také sloupců předpovědních pozorování pro datovou sadu časových řad ve formátu učení pod dohledem.

Podívejme se na některé příklady funkce shift v akci.

Falešnou datovou sadu časové řady můžeme definovat jako sekvenci 10 čísel, v tomto případě jeden sloupec v DataFrame takto:

Příklad 1.

Listing 10.1: *Použití funkce Shift*

```
from pandas import DataFrame

df = DataFrame()
df['t'] = [x for x in range(10)]
print(df)
```

Spuštění příkladu vytiskne data časové řady s řádkovými indexy pro každé pozorování.

	t
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9

Tabulka 10.9: *Výstup z příkladu 1*

Všechna pozorování můžeme posunout o jeden časový krok dolů vložením jednoho nového řádku nahoře. Protože nový řádek neobsahuje žádná data, můžeme použít NaN k vyjádření „žádná data“.

Funkce shift to může udělat za nás a tento posunutý sloupec můžeme vložit vedle naší původní řady.

Příklad 2.

Listing 10.2: *Použití funkce Shift*

```
from pandas import DataFrame
df = DataFrame()
df['t'] = [x for x in range(10)]
df['t-1'] = df['t'].shift(1)
print(df)
```

Spuštění příkladu nám poskytne dva sloupce v datové sadě. První s původními pozorováními a novou posunutou kolonou.

Vidíme, že posunutím řady dopředu o jeden časový krok nám vznikne primitivní problém učení pod dohledem, i když s X a y ve špatném pořadí. Ignorujte

sloupec popisků řádků. První řádek by musel být vyřazen kvůli hodnotě NaN. Druhý řádek zobrazuje vstupní hodnotu 0,0 ve druhém sloupci (vstup nebo X) a hodnotu 1 v prvním sloupci (výstup nebo y).

	t	t-1
0	0	NaN
1	1	0.0
2	2	1.0
3	3	2.0
4	4	3.0
5	5	4.0
6	6	5.0
7	7	6.0
8	8	7.0
9	9	8.0

Tabulka 10.10: *Výstup z příkladu 2*

Můžeme vidět, že pokud můžeme opakovat tento proces s posuny 2, 3 a více, jak bychom mohli vytvořit dlouhé vstupní sekvence (X), které lze použít k předpovědi výstupní hodnoty (y).

Operátor směny může také přijmout zápornou celočíselnou hodnotu. To má za následek vytažení pozorování nahoru vložením nových řádků na konec. Níže je uveden příklad:

Příklad 3.

Listing 10.3: *Použití funkce Shift*

```
from pandas import DataFrame
df = DataFrame()
df['t'] = [x for x in range(10)]
df['t+1'] = df['t'].shift(-1)
print(df)
```

Spuštění příkladu ukazuje nový sloupec s hodnotou NaN jako poslední hodnotou.

Vidíme, že sloupec prognózy lze brát jako vstupní (X) a druhý jako výstupní hodnotu (y). To znamená, že vstupní hodnotu 0 lze použít k předpovědi výstupní hodnoty 1.

t	t+1
0	1.0
1	2.0
2	3.0
3	4.0
4	5.0
5	6.0
6	7.0
7	8.0
8	9.0
9	NaN

Tabulka 10.11: *Výstup z příkladu 3*

Technicky, v terminologii předpovědí časových řad, aktuální čas (t) a budoucí časy (t+1, t+n) jsou předpovědní časy a minulá pozorování (t-1, t-n) se používají k vytváření předpovědí.

Můžeme vidět, jak lze pozitivní a negativní posuny použít k vytvoření nového DataFrame z časové řady se sekvencemi vstupních a výstupních vzorců pro problém s kontrolovaným učením.

To umožňuje nejen klasickou predikci $X \rightarrow y$, ale také $X \rightarrow Y$, kde vstupem i výstupem mohou být různé sekvence. [2]

Dále funkce posunu pracuje také na takzvaných problémech s více proměnnými časových řad. To je místo, kde místo jedné sady pozorování pro časovou řadu máme více (např. teplotu a tlak). Všechny proměnné v časové řadě lze posunout dopředu nebo dozadu a vytvořit tak vícerozměrné vstupní a výstupní sekvence. To prozkoumáme více později v tutoriálu.

10.3.2 Použití funkce *series_to_supervised()*

Můžeme použít funkci shift() v Pandas k automatickému vytvoření nových rámců problémů časových řad s ohledem na požadovanou délku vstupních a výstupních sekvencí.

[2] Příklad: $X \rightarrow y$ znamená že, z předchozích sekvencí teploty, predikujme následnou sekvenci teploty, $X \rightarrow Y$ znamená že, z předchozích sekvencí teploty, predikujeme následnou sekvenci roztažnosti mateiálu

To by byl užitečný nástroj, protože by nám umožnil prozkoumat různé rámce problému časové řady s algoritmy strojového učení, abychom zjistili, které by mohly vést k lepším modelům.

V této části definujeme novou funkci Pythonu s názvem series_to_supervised(), která vezme jednorozměrnou nebo vícerozměrnou časovou řadu a zarámuje ji jako supervidovanou učební datovou sadu.

Funkce má čtyři argumenty:

- *data:* Sekvence pozorování jako seznam nebo pole 2D NumPy - povinný parametr.
- *n_in:* Počet pozorování (zpoždění) jako vstup (X). Hodnoty mohou být mezi [1..len(data)] - nepovinný parametr. Implicitní hodnota je 1.
- *n_out:* Počet pozorování jako výstup (y). Hodnoty mohou být mezi [0..len(data)-1] - nepovinný parametr. Implicitní hodnota je 1.
- *dropnan:* Boolean, zda se mají vypustit řádky s hodnotami NaN - nepovinný parametr. Výchozí hodnota je True.

Funkce vrací jednu hodnotu - datový rámec:

- *return:* Pandas DataFrame série zarámovanou pro učení pod dohledem.

Nová datová sada je definována jako objekt DataFrame, přičemž každý sloupec je vhodně pojmenován jak číslem proměnné, tak časovým krokem. To umožňuje navrhnout řadu různých problémů s prognózou typu časové krokové sekvence z dané jednorozměrné nebo vícerozměrné časové řady. Jakmile je DataFrame vrácen, můžeme se rozhodnout, jak rozdělit řádky vráceného DataFrame do komponent X a y pro řízené učení libovolným způsobem.

Funkce je definována s výchozími parametry, takže pokud ji zavoláte pouze s vašimi daty, vytvoří DataFrame s t-1 jako X a t jako y.

Kompletní funkce je uvedena níže, včetně komentářů k funkcím.

Příklad 4.

Listing 10.4: *Použití funkce Series_to_supervised*

```
from pandas import DataFrame
from pandas import concat

def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
        """
        Frame a time series as a supervised learning dataset.
        Arguments:
                data: Sequence of observations as a list or NumPy array.
                n_in: Number of lag observations as input (X).
                n_out: Number of observations as output (y).
                dropnan: Boolean whether or not to drop rows with NaN values.
        Returns:
                Pandas DataFrame of series framed for supervised learning.
        """
        n_vars = 1 if type(data) is list else data.shape[1]
        df = DataFrame(data)
        cols, names = list(), list()
        # input sequence (t-n, ... t-1)
        for i in range(n_in, 0, -1):
                cols.append(df.shift(i))
                names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
        # forecast sequence (t, t+1, ... t+n)
        for i in range(0, n_out):
                cols.append(df.shift(-i))
                if i == 0:
                        names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
                else:
                        names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars
    )]
        # put it all together
        agg = concat(cols, axis=1)
        agg.columns = names
        # drop rows with NaN values
        if dropnan:
                agg.dropna(inplace=True)
        return agg
```

10.3.3 Jednokrokové jednorozměrné předpovědi časových řad

V prognózování časových řad je standardní praxí používat zpožděná pozorování (např. t-1) jako vstupní proměnné pro předpověď aktuálního časového kroku (t). Tomu se říká jednokroková prognóza.

Níže uvedený příklad demonstruje jeden časový krok zpoždění (t-1) pro predikci aktuálního časového kroku (t).

Příklad 5.

Listing 10.5: *Použití funkce Series_to_supervised*

```
from pandas import DataFrame
```

```
from pandas import concat

def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
        """
        Frame a time series as a supervised learning dataset.
        Arguments:
                data: Sequence of observations as a list or NumPy array.
                n_in: Number of lag observations as input (X).
                n_out: Number of observations as output (y).
                dropnan: Boolean whether or not to drop rows with NaN values.
        Returns:
                Pandas DataFrame of series framed for supervised learning.
        """
        n_vars = 1 if type(data) is list else data.shape[1]
        df = DataFrame(data)
        cols, names = list(), list()
        # input sequence (t-n, ... t-1)
        for i in range(n_in, 0, -1):
                cols.append(df.shift(i))
                names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
        # forecast sequence (t, t+1, ... t+n)
        for i in range(0, n_out):
                cols.append(df.shift(-i))
                if i == 0:
                        names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
                else:
                        names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars
    )]
        # put it all together
        agg = concat(cols, axis=1)
        agg.columns = names
        # drop rows with NaN values
        if dropnan:
                agg.dropna(inplace=True)
        return agg


values = [x for x in range(10)]
data = series_to_supervised(values)
print(data)
```

Výstupem tohoto příkladu je tato „transformovaná“ časová řada.

	var1(t-1)	var1(t)
1	0.0	1
2	1.0	2
3	2.0	3
4	3.0	4
5	4.0	5
6	5.0	6
7	6.0	7
8	7.0	8
9	8.0	9

Tabulka 10.12: *Výstup z příkladu 5*

Vidíme, že pozorování jsou pojmenována „var1“ a že vstupní pozorování je vhodně pojmenováno (t-1) a výstupní časový krok je pojmenován (t). Můžeme také vidět, že řádky s hodnotami NaN byly z DataFrame automaticky odstraněny.

Tento příklad lze zopakovat se vstupní posloupností libovolné délky čísel, například 3. To lze provést zadáním délky vstupní posloupnosti jako argumentu; například:

Příklad 6.

Listing 10.6: *Použití funkce Series_to_supervised*

```
#data = series_to_supervised(values, 3)

from pandas import DataFrame
from pandas import concat

def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
        """
        Frame a time series as a supervised learning dataset.
        Arguments:
                data: Sequence of observations as a list or NumPy array.
                n_in: Number of lag observations as input (X).
                n_out: Number of observations as output (y).
                dropnan: Boolean whether or not to drop rows with NaN values.
        Returns:
                Pandas DataFrame of series framed for supervised learning.
        """
        n_vars = 1 if type(data) is list else data.shape[1]
        df = DataFrame(data)
        cols, names = list(), list()
        # input sequence (t-n, ... t-1)
        for i in range(n_in, 0, -1):
                cols.append(df.shift(i))
                names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
        # forecast sequence (t, t+1, ... t+n)
        for i in range(0, n_out):
                cols.append(df.shift(-i))
                if i == 0:
                        names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
                else:
                        names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars
    )]
        # put it all together
        agg = concat(cols, axis=1)
        agg.columns = names
        # drop rows with NaN values
        if dropnan:
                agg.dropna(inplace=True)
        return agg


values = [x for x in range(10)]
data = series_to_supervised(values, 3)
print(data)
\end{python}
```

Opět platí, že spuštění příkladu vytiskne přeformátovanou sérii. Můžeme vidět, že vstupní sekvence je ve správném pořadí zleva doprava, přičemž výstupní proměnná má být predikována zcela vpravo.

	var1(t-3)	var1(t-2)	var1(t-1)	var1(t)
3	0.0	1.0	2.0	3
4	1.0	2.0	3.0	4
5	2.0	3.0	4.0	5
6	3.0	4.0	5.0	6
7	4.0	5.0	6.0	7
8	5.0	6.0	7.0	8
9	6.0	7.0	8.0	9

Tabulka 10.13: *Výstup z příkladu 6*

10.3.4 Vícekrokové nebo sekvenční předpovědi časových řad

Jiným typem prognostického problému je použití minulých pozorování k předpovědi sledu budoucích pozorování. To může být nazýváno sekvenční prognózou nebo vícekrokovou prognózou.

Můžeme zarámovat časovou řadu pro sekvenční předpověď zadáním dalšího argumentu. Například bychom mohli sestavit předpovědní problém se vstupní sekvencí dvou minulých pozorování a předpovědět dvě budoucí pozorování následovně:

Příklad 7.

Listing 10.7: *Použití funkce Series_to_supervised*

```
#  data = series_to_supervised(values, 2, 2)

from pandas import DataFrame
from pandas import concat

def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
        """
        Frame a time series as a supervised learning dataset.
        Arguments:
                data: Sequence of observations as a list or NumPy array.
```

```
11                  n_in: Number of lag observations as input (X).
12                  n_out: Number of observations as output (y).
13                  dropnan: Boolean whether or not to drop rows with NaN values.
14          Returns:
15                  Pandas DataFrame of series framed for supervised learning.
16          """
17          n_vars = 1 if type(data) is list else data.shape[1]
18          df = DataFrame(data)
19          cols, names = list(), list()
20          # input sequence (t-n, ... t-1)
21          for i in range(n_in, 0, -1):
22                  cols.append(df.shift(i))
23                  names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
24          # forecast sequence (t, t+1, ... t+n)
25          for i in range(0, n_out):
26                  cols.append(df.shift(-i))
27                  if i == 0:
28                          names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
29                  else:
30                          names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars
      )]
31          # put it all together
32          agg = concat(cols, axis=1)
33          agg.columns = names
34          # drop rows with NaN values
35          if dropnan:
36                  agg.dropna(inplace=True)
37          return agg
38
39
40  values = [x for x in range(10)]
41  data = series_to_supervised(values, 2, 2)
42  print(data)
```

Spuštění příkladu ukazuje diferenciaci vstupních (t-n) a výstupních (t+n) proměnných s aktuálním pozorováním (t) považovaným za výstup.

	var1(t-2)	var1(t-1)	var1(t)	var1(t+1)
2	0.0	1.0	2	3.0
3	1.0	2.0	3	4.0
4	2.0	3.0	4	5.0
5	3.0	4.0	5	6.0
6	4.0	5.0	6	7.0
7	5.0	6.0	7	8.0
8	6.0	7.0	8	9.0

Tabulka 10.14: *Výstup z příkladu 7*

10.3.5 Vícerozměrné předpovědi časových řad

Dalším důležitým typem časových řad jsou vícerozměrné časové řady.

Zde můžeme pozorovat několik různých vstupů a požadovat předpovědi jednoho nebo více z nich. Například můžeme mít dvě sady pozorování časových řad var1 a var2 a chceme předpovídat jednu nebo obě z nich.

Příklad 8.

Listing 10.8: *Použití funkce Series_to_supervised*

```
1
2  from pandas import DataFrame
3  from pandas import concat
4
5  def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
6          """
7          Frame a time series as a supervised learning dataset.
8          Arguments:
9                  data: Sequence of observations as a list or NumPy array.
10                 n_in: Number of lag observations as input (X).
11                 n_out: Number of observations as output (y).
12                 dropnan: Boolean whether or not to drop rows with NaN values.
13         Returns:
14                 Pandas DataFrame of series framed for supervised learning.
15         """
16         n_vars = 1 if type(data) is list else data.shape[1]
17         df = DataFrame(data)
18         cols, names = list(), list()
19         # input sequence (t-n, ... t-1)
20         for i in range(n_in, 0, -1):
21                 cols.append(df.shift(i))
22                 names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
23         # forecast sequence (t, t+1, ... t+n)
24         for i in range(0, n_out):
25                 cols.append(df.shift(-i))
26                 if i == 0:
27                         names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
28                 else:
29                         names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars
       )]
30         # put it all together
31         agg = concat(cols, axis=1)
32         agg.columns = names
33         # drop rows with NaN values
34         if dropnan:
35                 agg.dropna(inplace=True)
36         return agg
37
38
39 raw = DataFrame()
40 raw['ob1'] = [x for x in range(10)]
41 raw['ob2'] = [x for x in range(50, 60)]
42 values = raw.values
43 data = series_to_supervised(values)
44 print(data)
```

Spuštění příkladu vytiskne nové uspořádání dat, které ukazuje vstupní vzor s jedním časovým krokem pro obě proměnné a výstupní vzor s jedním časovým krokem pro obě proměnné.

Opět platí, že v závislosti na specifikách problému lze rozdělení sloupců na složky X a Y zvolit libovolně, například kdyby jako vstup bylo poskytnuto také aktuální pozorování var1 a předpovídat pouze var2.

	var1(t-1)	var2(t-1)	var1(t)	var2(t)
1	0.0	50.0	1	51
2	1.0	51.0	2	52
3	2.0	52.0	3	53
4	3.0	53.0	4	54
5	4.0	54.0	5	55
6	5.0	55.0	6	56
7	6.0	56.0	7	57
8	7.0	57.0	8	58
9	8.0	58.0	9	59

Tabulka 10.15: *Výstup z příkladu 8*

Můžeme vidět, jak to lze použít pro předpovídání sekvencí s vícerozměrnými časovými řadami zadáním délky vstupní a výstupní sekvence, jak je uvedeno výše.

Níže je například uveden příklad přerámování s jedním časovým krokem jako vstupem a dvěma časovými kroky jako výstupní předpovědní sekvence.

Příklad 9.

Listing 10.9: *Použití funkce Series_to_supervised*

```
from pandas import DataFrame
from pandas import concat

def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
        """
        Frame a time series as a supervised learning dataset.
        Arguments:
                data: Sequence of observations as a list or NumPy array.
                n_in: Number of lag observations as input (X).
                n_out: Number of observations as output (y).
                dropnan: Boolean whether or not to drop rows with NaN values.
        Returns:
                Pandas DataFrame of series framed for supervised learning.
        """
        n_vars = 1 if type(data) is list else data.shape[1]
        df = DataFrame(data)
```

```
17          cols, names = list(), list()
18          # input sequence (t-n, ... t-1)
19          for i in range(n_in, 0, -1):
20                  cols.append(df.shift(i))
21                  names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
22          # forecast sequence (t, t+1, ... t+n)
23          for i in range(0, n_out):
24                  cols.append(df.shift(-i))
25                  if i == 0:
26                          names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
27                  else:
28                          names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars
      )]
29          # put it all together
30          agg = concat(cols, axis=1)
31          agg.columns = names
32          # drop rows with NaN values
33          if dropnan:
34                  agg.dropna(inplace=True)
35          return agg
36
37
38  raw = DataFrame()
39  raw['ob1'] = [x for x in range(10)]
40  raw['ob2'] = [x for x in range(50, 60)]
41  values = raw.values
42  data = series_to_supervised(values, 1, 2)
43  print(data)
```

Výsledkem příkladu je tento výpis:

	var1(t-1)	var2(t-1)	var1(t)	var2(t)	var1(t+1)	var2(t+1)
1	0.0	50.0	1	51	2.0	52.0
2	1.0	51.0	2	52	3.0	53.0
3	2.0	52.0	3	53	4.0	54.0
4	3.0	53.0	4	54	5.0	55.0
5	4.0	54.0	5	55	6.0	56.0
6	5.0	55.0	6	56	7.0	57.0
7	6.0	56.0	7	57	8.0	58.0
8	7.0	57.0	8	58	9.0	59.0

Tabulka 10.16: *Výstup z příkladu 9*

Experimentujte se svou vlastní datovou sadou a vyzkoušejte několik různých rámců, abyste zjistili, co funguje nejlépe.
Účelem v této sekci bylo , jak přerámovat datové sady časových řad jako problémy s učením pod dohledem s pomocí Pythonu.

Bylo zde vysvětleno:

- Funkce Pandas shift(), jak ji lze použít k automatickému definování řízených učebních datových sad z dat časových řad.
- Jak přerámovat jednorozměrnou časovou řadu na jednokrokové a vícekrokové řízené učební problémy.
- Jak přerámovat vícerozměrné časové řady na jednokrokové a vícekrokové řízené učební problémy.

11 Základy matematiky pro neuronové sítě

11.1 Pravidlo derivace složených funkcí

Pravidlo derivace složených funkcí (řetězové pravidlo) se používá k výpočtu derivací funkcí vytvořených složením dalších funkcí, jejichž derivace jsou známy.

Nechť x je reálné číslo a obě f a g jsou funkce $f \in R$. Předpokládejme, že $y = g(x)$ a $z = f(g(x)) = f(y)$.

Pak lze napsat pravidlo derivace složených funkcí následovně:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx} \tag{11.1}$$

To lze rozšířit nad rámec skalární proměnné takto:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i} \tag{11.2}$$

11.1.1 Hledání extrémů funkce výpočtem gradientu

Naznačme řešení výpočtu gradientu funkce. Vezměme rovnice 6.15, 6.16,6.17,6.18 a spočítejme derivaci složené funkce. Použijeme pravidla o derivaci složené funkce. 6.19

$$a^{(k)} = \sigma(w^{(k)}a^{(k-1)} + b^{(k)})$$

$$z^k = w^{(k)} a^{(k-1)} + b^{(k)}$$

$$a^{(k)} = \sigma z^{(k)}$$

$$J = (a^{(k)} - \overline{y})^2$$

Řešme parciální derivace složené funkce:

$$\frac{\partial J}{\partial w^{(k)}} = \frac{\partial J}{\partial a^{(k)}} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial w^{(k)}}$$

přičemž:

$$\frac{\partial J}{\partial a^{(k)}} = \frac{\partial (a^{(k)} - \overline{y})^2}{\partial w^{(k)}} = 2\left(a^{(k)} - \overline{y})\right)$$

$$\frac{\partial a^{(k)}}{\partial z^{(k)}} = \frac{\partial\left(\sigma z^{(k)}\right)}{\partial z^{(k)}} = \sigma^{'}\left(z^{(k)}\right)$$

$$\frac{\partial z^{(k)}}{\partial w^{(k)}} = \frac{\partial\left(w^{(k)} a^{(k-1)} + b^{(k)}\right)}{\partial w^{(k)}} = a^{(k-1)}$$

Takže, dosazením dílčích výsledků derivace dostaneme optimalizační funkci:

$$\frac{\partial J}{\partial w^{(k)}} = \frac{\partial J}{\partial a^{(k)}} \frac{\partial a^{(k)}}{\partial z^{(k)}} \frac{\partial z^{(k)}}{\partial w^{(k)}} = 2\left(a^{(k)} - \overline{y})\right) \sigma^{'}\left(z^{(k)}\right) a^{(k-1)}$$

11.2 Konvoluce

Konvoluce je matematická operace dvou funkcí $f(t)$ a $g(t)$ téhož argumentu, kde funkce $f(t)$ je zpracovávaná funkce a funkce $g(t)$ představuje konvoluční jádro (konvoluční filtr). Funkce $g(t)$ působí na $f(t)$ operací "konvoluce", která je definována případě spojitých funkcí integrálem:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau$$

a v případě diskrétních funkcí součinem matic:

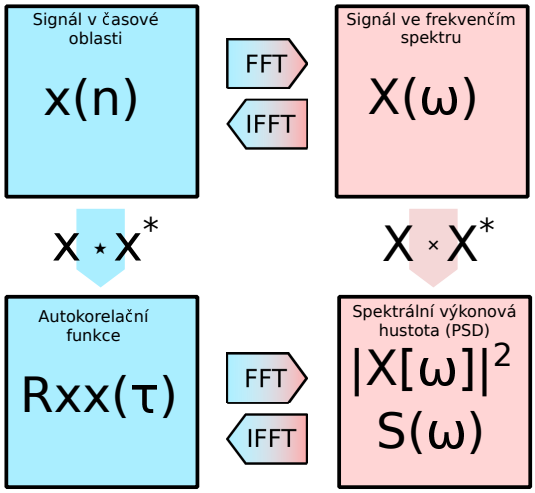
$$(f * g)[n] = \sum_{m=-\infty}^{\infty} f[m]g[n-m]$$

Například výsledkem operace konvoluce diskrétních signálů je „násobení" jednotlivých prvků zdrojového signálu s odpovídajícími prvky konvolučního jádra a následná sumace všech takto vzniklých dílčích výsledků.

Konvoluce má zásadní význam v oblasti zpracování signálů, teorii pravděpodobnosti, statistice, počítačovém vidění a mnoha jiných technických oborech. Například v oblasti neuronových sítí a počítačového vidění zvýrazňuje (filtruje) významné části obrazu od částí nevýznamných. Obecně tedy představuje speciální typ filtru, který umožňuje výrazně snížit objem zpracovávaných dat tak, že oddělí nevýznamné části a propustí jen části významné. V oblasti obrazových informací se například používá k detekci hran.

11.3 DSP čtyřúhelník

DSP (Digital Signal Processing) čtyřúhelník je pomúckou pro pochopení souvislostí při zpracování digitálního signálu a jeho jednotlivých matematických vazeb. Výhodou zpracování signálů ve frekvenční oblasti je nižší náročnost algoritmů na výpočetní výkon $n * log(n)$, zatímco v časové oblasti je složitost vyjádřena $n * n$.



Obrázek 11.1: *DSP čtyřúhelník*

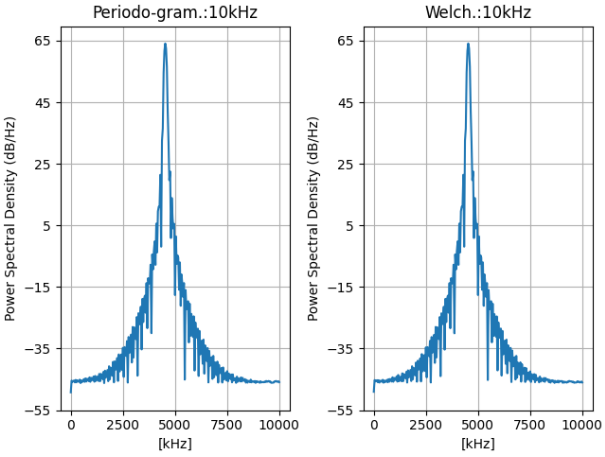
kde:[1]

$x(n)$	–	vzorkovaný signál v časové oblasti
$x \star x^*$	–	konvoluce signálu
$Rxx(\tau)$	–	autokorelační funkce signálu
$X(\omega)$	–	spektrum signálu
$X \times X^*$	–	komplexní sdružení signálu
$\|X[\omega]\|^2 = S(\omega)$	–	spektrální výkonnová hustota signálu

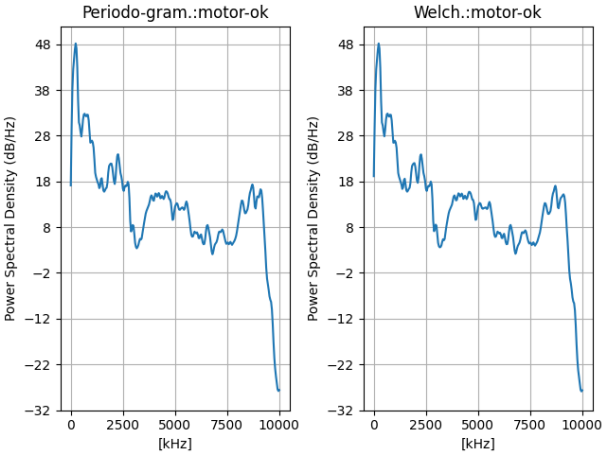
[1] operátorem $\star$ značíme operaci konvoluce a operátorem $\times$ komplexní sdružení

11.4 Spektrální výkonová hustota (PSD)

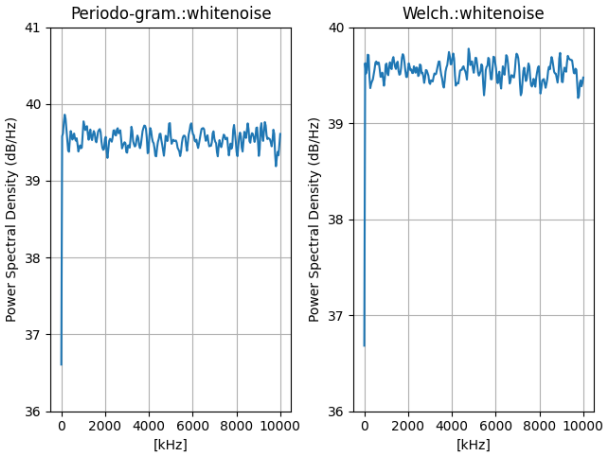
Fourierova transformace autokorelační funkce u periodických, neperiodických a náhodných signálů se nazývá Spektrální výkonová, resp. energetická hustota signálu $S(\omega)$ [25] [4]. Zjednodušeně si lze spektrální výkonovou hustotu představit jako funkci, která zobrazí, kolik je akumulováno energie na jednotlivých frekvencích, spektra sledovaného signálu. Tím samozřejmě lze získat i informaci o tom, zda práce sledovaného objektu se blíží jeho vlastní (rezonanční) frekvenci. To je například u obráběcího stroje velmi nežádoucí stav.



Obrázek 11.2: *Spektrální výkonová hustota signálu 10 kHz*



Obrázek 11.3: *Spektrální výkonová hustota motoru*



Obrázek 11.4: *Spektrální výkonová hustota náhodného procesu*

Vlastní spektrální výkonovou hustotu signálu $S(\omega)$ definuje Wiener-Chinčinův teorem:[2]

$$\begin{aligned} S(\omega) &= \int_{-\infty}^{+\infty} R(\tau) e^{-j\omega\tau} d\tau \\ &= F[R_T(\tau)] \end{aligned}$$

Autokorelační funkce:

$$\begin{aligned} R(\tau) &= \frac{1}{2\pi} \int_{-\infty}^{+\infty} S(\omega) e^{j\omega\tau} d\omega \\ &= F^{-1}[S\omega] \end{aligned}$$

kde:

$S(\omega)$	–	spektrální výkonnová hustota signálu
$R(\tau)$	–	autokorelační funkce signálu

Úkolem vibrodiagnostiky je detekovat signály nacházející se v rozsahu, které znázorňují obrázky 11.2 a 11.4 Spektrální výkonová hustota je velmi dobré

[2] Podrobný matematický význam autokorelační funkce a spektrální hustoty signálu je uveden v [29].

měřítko pro detekci signálů v okolí rezonančních kmitočtů stroje, které jsou pro přesnost a bezporuchovost stroje významné. Primárním úkolem konstrukce stroje je pokud možno posunout rezonanční kmitočet co nejvýše (tuhost konstrukce), aby nezasahoval do kmitočtů vznikajících při obrábění.

Z obrázků 11.2, 11.3 a 11.4 vyplývá, že Spektrální výkonová hustota reálného měřeného signálu se bude vždy pohybobat mezi dvěma ideálními stavy a to determinovaným procesem který je popsán harmonickým kmitočtem viz obr. 11.2 a bílým šumem na obr. 11.4

Tato veličina však nemá pro nasazení neuronové sítě pro rozpoznávání vlastností takový význam jako krátkodobá Fourierova transformace, která díky 2D tenzoru datové struktury (časová osa a osa frekvencí) viz obr 11.1. Na výše uvedených obrázcích 11.2, 11.3, 11.4 jsou znázorněny dva typy výpočtu Výkonové spektrální hustoty, klasická a Welchova, která má poněkud jiný algoritmus jako klasický výpočet. Welchova metoda je vylepšením standardní metody odhadu periodogramového spektra v tom, že snižuje šum v odhadovaných výkonových spektrech ovšem za cenu snížení frekvenčního rozlišení. Kvůli šumu způsobenému nedokonalými a konečnými daty je však použití Welchovy metody v mnoha případech výhodnější. DSP čtyřúhelník nám však napoví, že spektrální výkonovou hustotu lze vypočítat (výhodněji) ze spektra signálu.

11.5 Short Term Fourier Transform - STFT

STFT - (Short Time Fourier Transform) je transformace používaná k určení kmitočtů a fázového obsahu jistých přesně definovaných úseků signálu (okna) , a jejich změn v čase. Algoritmus STFT spočívá v rozdělení signálu na krátké segmenty stejné délky nad nimiž se počítá klasická Fourierova transformace. To odhalí Fourierovo spektrum na každém kratším segmentu. Následně se jednotlivá spektra znázorňují v časové rovině. Tím získáme spektrogram který znázorňuje jednotlivá spektra v čase.

Definice STFT

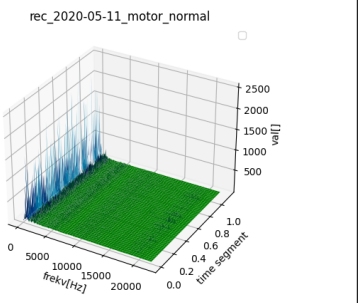
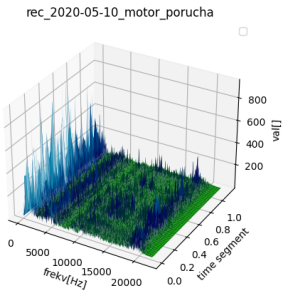
$$X_m(\omega) = \sum_{n=-\infty}^{n=\infty} x(n)w(n-mR)e^{-j\omega n}$$
$$= DFTF_\omega(x.shift_{mR}(w))$$

kde:

$x(n)$	–	vstupní signál v *t(n)*
$w(n)$	–	délka window funkce
$X_m(\omega)$	–	DFTF v konkrétním okně
R	–	velikost skoku mezi DTFT
$DTFT$	–	diskrétní Fourierova transformace

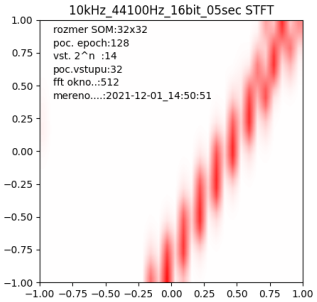
Princip STFT lze snadno pochopit na 3D zobrazení, znázorněném v tabulce 11.1. Zde je dobře patrná závislost času (osa X), frekvence (osa Y) a magnitudy (osa Z).

Při výpočtu STFT se definuje tzv. “window funkce”, což není nic jiného, nežli rozdělení signálu na kratičké úseky, v nichž se počítá FFT. Tím je získán vývoj signálu v čase. Pro “window funkce” existuje velké množství definic (Hanningovo, okno, trojúhelníkové okno, okno s Gaussovým rozložením). Každé okno má své jisté vlastnosti a každé je vhodné pro různé typy signálů. V našem případě je zvoleno Hanningovo okno, které má pro sledovaný typ signálu nejlepší výsledky.

Normální režim	Režim v poruše
	

Tabulka 11.1: *Spektrogramy motorů v normálním stavu a ve stavu poruchy*

V tabulce 11.1 jsou znázorněny spektrogramy vadného a dobrého motoru. Z obrázků je patrné, že u dobrého motoru převažují frekvecne v okolí 50 Hz, což je pro motor napájený z distribuční sítě normální kmitočtové spektrum. U motoru s vadnými ložisky je patrné velké množství parazitních kmitočtů v celém spektru od 50Hz do cca 16kHz. Pro kvalitní analýzu spektra by byl vhodný snímač, který by byl schopen zpracovat i mnohem větší frekvence (řádově do 80 až 100 kHz). Ten však nebyl k dispozici.



Obrázek 11.5: *Etalon 10 kHz ve frekvenční oblasti*

Na obrázku 11.5 je opět znázorněno, jak si neuronová síť poradí se signálem 10 kHz, ovšem ve frekvenční oblasti. Zde je patrná závislost sinusového průběhu signálu znázorněného pomocí Kohonenovy mapy.

Seznam obrázků

Seznam tabulek

Reference

[1] Leonard M Adleman. „Computing with DNA“. In: *Scientific american* 279.8 (1998), s. 34–41.

[2] Umut Asan a Secil Ercan. „An Introduction to Self-Organizing Maps“. In: led. 2012, s. 299–319. ISBN: 978-94-91216-76-3. DOI: 10.2991/978-94-91216-77-0_14.

[3] Aldebaran Group for Astrophysics. *Spolek ALDEBARAN GROUP FOR ASTROPHYSICS (zkratka AGA)*. 2021. URL: https://www.aldebaran.cz/.

[4] Andrew J Barbour a Robert L Parker. „Normalization of Power Spectral Density estimates“. In: (2015).

[5] François Chollet. *Deep Learning with Python*. Manning, lis. 2017. ISBN: 9781617294433.

[6] Frey a Pachova. *Vektorová a tenzorová analýza*. Státní nakladatelství technické literatury, n.p., Spálená 51 Praha, 1964.

[7] Mike PhD Fritze et al. „The Death of Moore's Law“. In: *STEPS: SCIENCE, TECHNOLOGY, ENGINEERING, AND POLICY STUDIES* (2016).

[8] Ian Goodfellow, Yoshua Bengio a Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[9] Raymond Greenlaw, H. James Hoover a Walter L. Ruzzo. *Limits to Parallel Computation: P-completeness Theory*. New York, NY, USA: Oxford University Press, Inc., 1995. ISBN: 0-19-508591-4.

[10] Sepp Hochreiter a Jürgen Schmidhuber. „Long Short-Term Memory“. In: *Neural Computation* 9.8 (1997), s. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[11] Fakultit Informatik et al. „Gradient Flow in Recurrent Nets: the Difficulty of Learning Long-Term Dependencies“. In: *A Field Guide to Dynamical Recurrent Neural Networks* (břez. 2003).

[12] Melody Kiang, Michael Hu a D.M. Fisher. „An extended self-organizing map network for market segmentation—a telecommunication example“. In: *Decision Support Systems* 42 (říj. 2006), s. 36–47. DOI: 10.1016/j.dss.2004.09.012.

[13] Melody Y. Kiang. „Extending the Kohonen self-organizing map networks“. In: *Computational Statistics and Data Analysis* 38.2 (2001), s. 161–180. ISSN: 0167-9473. DOI: https://doi.org/10.1016/S0167-9473(01)00040-8. URL: https://www.sciencedirect.com/science/article/pii/S0167947301000408.

[14] Teuvo Kohonen. *Self-Organizing Maps*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001. ISBN: 9783642569272 3642569277. URL: http://link.springer.com/book/10.1007/978-3-642-56927-2.

[15] R. Kuo et al. „Integration of self-organizing feature maps neural network and genetic K-means algorithm for market segmentation“. In: *Expert Syst. Appl.* 30 (ún. 2006), s. 313–324. DOI: 10.1016/j.eswa.2005.07.036.

[16] Demlová M. a Nagy J. *Matematika pro vysoké školy technické sešit III - Algebra*. SNTL = Nakladatelství technické literatury, n.p. Spálená 51 Praha, 1982.

[17] Christoph von der Malsburg. „Self-organization of orientation sensitive cells in striate cortex“. In: *Biological Cybernetics* 14 (led. 1973), s. 85–100. DOI: 10.1007/BF00288907.

[18] Warren S. McCulloch a Walter Pitts. „A logical calculus of the ideas immanent in nervous activity“. In: *The bulletin of mathematical biophysics* 5.4 (1943), s. 115–133. ISSN: 1522-9602. DOI: 10.1007/BF02478259. URL: https://doi.org/10.1007/BF02478259.

[19] Patrick R. McMullen. „A Kohonen self-organizing map approach to addressing a multiple objective, mixed-model JIT sequencing problem“. In: *Production Economics* (2001).

[20] A. NIELSEN Michael a L. CHUANG Isaac. *Quantum Computing*. Cambridge University Press, Cambridge 2000, 2000.

[21] Kulhánek P. *Vybrané kapitoly z teoretické fyziky 1*. Aldebaran Group for Astrophysics, 2020. ISBN: 978-80-906638-2-4. URL: www.aldebaran.cz.

[22] Tom Ran, Shai Kaplan a Ehud Shapiro. „Molecular implementation of simple logic programs“. In: *Nat Nano* 4.10 (2009), s. 642–648. ISSN: 1748-3387. DOI: 10.1038/nnano.2009.203. URL: http://dx.doi.org/10.1038/nnano.2009.203.

[23] „Neural Network Architecture“. In: *Encyclopedia of Machine Learning*. Ed. Claude Sammut a Geoffrey I. Webb. Boston, MA: Springer US, 2010, s. 716–716. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_587. URL: https://doi.org/10.1007/978-0-387-30164-8_587.

[24] Ralf C. Staudemeyer a Eric Rothstein Morris. „Understanding LSTM - a tutorial into Long Short-Term Memory Recurrent Neural Networks“. In: *CoRR* abs/1909.09586 (2019). arXiv: 1909.09586. URL: http://arxiv.org/abs/1909.09586.

[25] Jan Sýkora. *Teorie digitální komunikace*. Česká technika - nakladatelství ČVUT ISBN 80-01-02478-4, 2005.

[26] Kolektiv TMAI. *Legenda TM-AI k datům na AWS cloudu*. Tech. zpr. TAJMAC-ZPS, a.s., 2022.

[27] HEY Tony a WALTERS Patrick. *Nový kvantový vesmír*. Cambridge University Press, Cambridge 2003, 2003.

[28] Marc M. Van Hulle. „Self-organizing Maps“. In: *Handbook of Natural Computing*. Ed. Grzegorz Rozenberg, Thomas Bäck a Joost N. Kok. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, s. 585–622. ISBN: 978--3-540-92910-9. DOI: 10.1007/978-3-540-92910-9_19. URL: https://doi.org/10.1007/978-3-540-92910-9_19.

[29] František Vejražka. *Signály a soustavy*. Vydavatelství ČVUT, 1983.

[30] Wikipedia. *Wikipedia*. URL: http://wikipedia.org.