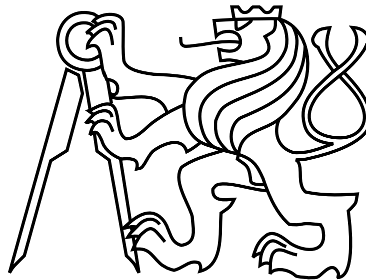
České vysoké učení technické v Praze
Fakulta strojní
Ústav přístrojové a řídicí techniky



Magisterská práce

Adaptivní predikce korekčních faktorů válcovacích sil

Bc. Adam Peichl

Vedoucí práce: Ing. Cyril Oswald, Ph.D.

Studijní program: Strojní inženýrství, prezenční, magisterský

Obor: Přístrojová a řídicí technika

11. června 2019

ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Peichl** Jméno: **Adam** Osobní číslo: **424057**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav přístrojové a řídicí techniky**

Studijní program: **Strojní inženýrství**

Studijní obor: **Přístrojová a řídicí technika**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Adaptivní predikce korekčních faktorů válcovacích sil

Název diplomové práce anglicky:

Adaptive prediction of rolling forces correction factors

Pokyny pro vypracování:

1) Navrhněte metodu vhodnou k predikci korekčních faktorů včetně rešerše.
2) Na základě vybrané metody vytvořte algoritmus pro výpočet korekčních faktorů.
3) Ověřte algoritmus na dodaných datech.

Seznam doporučené literatury:

- Madan Gupta, Liang Jin a Noriyasu Homma. "Static and dynamic neural networks: from fundamentals to advanced theory". John Wiley & Sons, 2004.
- Tomáš Hobza. "Matematická statistika". FJFI ČVUT Praha, 2011.
- Gejza Dohnal. "Základy stochastiky". FS ČVUT Praha, 2001.
- Léon Bottou, Frank E Curtis a Jorge Nocedal. "Optimization methods for large-scale machine learning". In: SIAM Review 60.2, 2018, s. 223-311
- Jorge J Moré. "The Levenberg-Marquardt algorithm: implementation and theory". In: Numerical analysis. Springer, 1978, s. 105–116

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Cyril Oswald, Ph.D., U12110.3

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **26.04.2019** Termín odevzdání diplomové práce: **12.06.2019**

Platnost zadání diplomové práce: ______________

Ing. Cyril Oswald, Ph.D.
podpis vedoucí(ho) práce

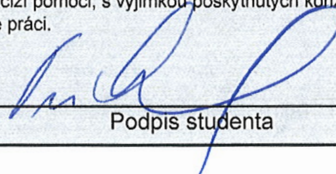
podpis vedoucí(ho) ústavu/katedry

prof. Ing. Michael Valášek, DrSc.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

26-04-2019
Datum převzetí zadání



Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Poděkování

Rád bych poděkoval panu Ing. Cyrilu Oswaldovi, Ph.D. který celý proces tvorby této práce sledoval a korigoval, firmě PTSW, která mi práci na této zajímavé problematice umožnila a v neposlední řadě také mé rodině, která mě vždy podporovala.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu.
Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

V Praze dne 12. 6. 2019 ....

Abstract

This work focuses on prediction of force and torque correction factors for metal sheets rolling obtained by mathematical-physical model, which leads to increase of an accuracy of a whole process. The goal of this work is a proposal of a software solution, its implementation and also following validation on real data obtained from measurements of the company *PT SOLUTIONS WORLDWIDE spol s.r.o.*

Considering the fact, that mathematical-physical model is very complex, we propose several methods consisting of data preprocessing algorithms, neural networks and machine learning.

Completed module is written in C++ and is included in software packages of a company PTSW at the moment. Its real application is prepared to be used in metal sheets rolling production lines.

Keywords: *prediction*, *metal sheet rolling*, *artificial intelligence*, *HONU*, *machine learning*

Abstrakt

Tato práce se zabývá predikcí korekčních faktorů silových a momentových parametrů pro válcování kovů získaných matematickým modelem, což vede ke zvýšení přesnosti celého procesu. Cílem práce je návrh softwarového řešení, jeho impementace a následná validace na reálných datech poskytnutých firmou *PT SOLUTIONS WORLDWIDE spol s.r.o.*

Vzhledem ke komplexnosti matematicko-fyzikálnáho modelu je v této práci prezentováno několik metod složených z algoritmů předzpracování dat, neuronových sítí a strojového učení.

Vytvořený modul v jazyce C++ je momentálně součástí softwarových balíků firmy PTSW, které mají být nasazeny v systémech řízení válcovacích linek.

Klíčová slova: *predikce*, *Válcování kovů*, *umělá inteligence*, *HONU*, *strojové učení*

Obsah

1 Úvod **1**
1.1 Cíl práce . . . 1
1.2 Členění práce . . . 2

2 Datová analýza **3**
2.1 Korelační analýza . . . 3
2.2 Analýza hlavních komponent (PCA) . . . 4
2.2.1 Standardizace . . . 4
2.2.2 Kovarianční matice . . . 4
2.2.3 Identifikace hlavních komponent . . . 5

3 Neuronové sítě **8**
3.1 Matematický model perceptronu . . . 8
3.2 Aktivační (výstupní) funkce . . . 9
3.2.1 Identita . . . 9
3.2.2 Binární aktivační funkce . . . 10
3.2.3 Sigmoida (logistická aktivační funkce) . . . 10
3.2.4 ReLU (rektifikovaná lineární aktivační funkce) . . . 10
3.3 Vícevrstvé neuronové sítě (MLP) . . . 10
3.4 Backpropagation . . . 12
3.4.1 Odvození . . . 12

4 Neuronové jednotky vyšších řádů **15**
4.1 Matematický model HONU . . . 15
4.2 Výpočetní složitost a problém se sílou aproximace . . . 16
4.3 HONU z pohledu praktického využití . . . 17

5 Neuronové sítě vyšších řádů **19**
5.1 Sigma-pi neuronové sítě . . . 19
5.2 QNN - Kvadratické neuronové sítě . . . 20

6 Učicí algoritmy **22**
6.1 Dávkové metody . . . 22
6.1.1 Batch Gradient Descent (BGD) . . . 23
6.1.1.1 Pseudokód . . . 23
6.1.2 Stochastic Gradient Descent (SGD) . . . 24

6.1.2.1 Pseudokód 25
6.1.3 Konjugované gradienty (CG) 25
6.1.3.1 Teorie 26
6.1.3.2 Pseudokód 27
6.1.4 Gauss-Newtonova metoda 27
6.1.4.1 Teorie 27
6.1.4.2 Pseudokód 28
6.1.5 Levenberg-Marquardtova metoda 28
6.1.5.1 Pseudokód 29
6.2 Metody vzorek po vzorku 29
6.2.1 Gradient Descent 29
6.2.1.1 Pseudokód 29

7 Analýza dat z procesu válcování 30
7.1 Kombinace vstupů 1 31
7.2 Kombinace vstupů 2 31

8 Návrh a validace modelů 35
8.1 Soustava dvou MLP (P01 - MLP) 35
8.1.1 Architektura 36
8.1.2 Proces učení 36
8.1.3 Validace 37
8.1.4 Zhodnocení 37
8.2 Soustava dvou HONU (P02 - HONU) 38
8.2.1 Architektura 38
8.2.2 Proces učení 38
8.2.3 Validace 39
8.2.4 Zhodnocení 41
8.3 Hluboké MLP (P03 - Deep MLP) 41
8.3.1 Architektura 41
8.3.2 Proces učení 42
8.3.3 Validace 42
8.3.4 Zhodnocení 43
8.4 Výběr architektury 44

9 Implementace do jazyka C++ 46
9.1 Architektura modulu v C++ 46
9.1.1 PolyUnit 47
9.1.2 Shaper 48
9.1.3 SGD 50
9.1.4 LMA 52
9.2 Závěrečné poznámky k implementaci 53

10 Závěr 54

Literatura 57

Seznam obrázků

2.1 Ilustrativní příklady několika typů rozložení dat a jejich korelačních koeficientů. 6
2.2 Ilustrační příklad PCA na 2D datech. Matice V jsou vlastní vektory (řádek je vlastní vektor) a vektor λ jsou vlastní čísla. 7

3.1 Schéma modelu perceptronu. $\mathbf{x} = [1, x_1, x_2, \ldots, x_n]^T$ je rozšířený vstupní vektor, $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \ldots, \theta_n]$ je vektor vah a $\phi(z)$ je aktivační funkce. Podobnost s biologickým neuronem je zřejmá na první pohled 9
3.2 Bežně používáné nebo známé aktivační funkce. Aktivační funkce jsou vyznačeny modře a jejich derivace jsou vyznačeny červeně typem čáry "-.". 11
3.3 Schéma obecného modelu vícevrstvé neuronové sítě (MLP). $P_i^{(j)}$ je perceptron číslo i vrstvy s indexem j. 12
3.4 Problém s binární relací XOR - lineární separovatelnost množin. 13
3.5 Část výpočetního grafu jednoduché MLP. Kde J je chybové kritérium, y je požadovaný výstup, $a^{(L)}$ je výstup vrstvy L, $z^{(L)}$ je výstup vrstvy L před aktivační funkcí, θ je váha a b je práh. 14

4.1 Blokové schéma neuronové jednotky vyššího řádu (HONU). $\mathbf{x}$ je vstupní vektor, θ jsou váhy, $\phi(.)$ je aktivační funkce a y je výstup. Toto schéma bylo volně převzato a mírně upraveno z [6]. 16
4.2 Porovnání 3 typů HONU. Závislost počtu trénovacích parametrů (vah) na počtu vstupů pro první tři rády HONU - tedy LNU, QNU a CNU. 17
4.3 Ilustrace tzv. overfittingu (přefitování). Pokud zvolíme moc aproximačně silný model (červená) a naše naměřená trénovací data obsahují chybný vzorek, je vidět, že model ztrácí generalizační schopnost. Naopak správně zvolený model (modrá) zachovává generalizační schopnost a blíží se k správnému funkčnímu průběhu (zelená). Obrázek byl převzat a upraven z [7]. 18

5.1 Obecné schéma S-PNN. Kde $x_1, x_2, \ldots, x_n$ jsou vstupy a y je výstup. Váhy ve vrstvě sumačního členu θ jsou určeny k trénování, zatímco váhy ve vrstvě součinových členů mají konstatní hodnotu (obvykle jsou to přirozená čísla). . 20
5.2 Navrhnutý model QNN. Tato síť je složená ze dvou QNU vrstev. Výstupní vrstva obsahuje pouze jediné QNU. Převzato z článku [7]. 21
5.3 Zavislost celkovém počtu vah QNN na počtu použitých QNU vstupní vrstvy. Převzato z článku [7]. 21

6.1 Ilustrační srovnání algoritmů BGD a SGD, na osách x a y jsou parametry θ_0 a θ_1. Pomyslná osa z (znázorněná jako barevné vrstevnice) je chybová funkce J_θ. Černý bod v souřadnících $[-0.28, 0.58]$ je lokální minimum. 25

7.1 Korelační komatice u kombinací vstupů 1 a 2. 32
7.2 PCA provedená na kombinaci vstupů 1 33
7.3 PCA provedená na kombinaci vstupů 2 34

8.1 Architektura řešení P01 - MLP. Vstupy jsou standardizovány, řešení obsahuje dvě MLP s totožnou architekturou. Silové MLP je trénováno na silový KF, momentové MLP je trénováno na momentový KF. 36
8.2 Proces učení. Ilustrativně pro dvě MLP (MLP FM3 - F a MLP FM4 - T). . . 37
8.3 Srovnání pro válcovací stolici FM1 38
8.4 Architektura řešení P02 - HONU. Vstupy jsou standardizovány, řešení obsahuje dvě HONU stejného řádu. Silové HONU je trénováno na silový KF, momentové HONU je trénováno na momentový KF. 39
8.5 Srovnání metod SBSGD a LMA pro silové HONU válcovací stolice FM1 (Pro všechny stolice byl průběh velmi podobný). Tento graf má demonstrovat rychlost konvergence algoritmu LMA oproti SBSGD. 40
8.6 Srovnání QNU a MF modelu pro válcovací stolici RM2 pomocí KDE grafů. . 40
8.7 Srovnání QNU a MF modelu pomocí krabicového grafu. 41
8.8 Architektura řešení P03 - Deep MLP. Vstupy jsou standardizovány. Informace o válcovací stolici je buď ve formátu *one-hot* a nebo pouze jedno číslo 42
8.9 Proces učení. Srovnání dvou hlubokých MLP sítí: Hluboká MLP - skalár a Hluboká MLP - one-hot. Povšiměte si, že každá MLP byla trénována různý počet epoch. 43
8.10 Srovnání hlubokých MLP pomocí KDE grafu. 43
8.11 Srovnání hlubokých MLP a MF modelu pomocí krabicového grafu. 44

9.1 Schéma modulu v C++. Část označená ---je celá řešená touto hlavičkovou knihovnou. Části mimo už byly vyřešené nebo v řešení firmou PTSW. 47

Seznam tabulek

4.1 Porovnání nejčastěji používaných řádů HONU (řády 1, 2 a 3). Poslední řádek je obecný pro HONU jakéhokoli řádu. 18

6.1 Srovnání výhod a nevýhod metody SGD 24

7.1 Doporučené kombinace vstupů podle expertů na válcování z firmy PTSW. . . 31

8.1 Použité kombinace vstupů pro P01 - MLP 37
8.2 Použité kombinace vstupů pro P02 - HONU 39

Seznam zkratek

BGD Batch gradient descent

CG konjugované gradienty (z angl. Conjugate gradients)

CNS centrální nervová soustava

CNU kubická neuronová jednotka (z angl. Cubic neural unit)

GNA Gauss-Newtonova metoda

HONU Neuronová jednotka vyššího řádu (z angl. Higher order neural unit)

KDE jádrový odhad (z angl. Kernel density estimation)

KF korekční faktor

LMA Levenberg-Marquardtova metoda

LNU lineární neuronová jednotka (z angl. Linear neural unit)

MF matematicko-fyzikální

MLP Vícevrstvá neuronová síť (z angl. Multi-layer perceptron)

MSE střední kvadratická chyba (z angl. Mean squared error)

NN neuronová síť (z angl. neural network)

NU neuronová jednotka (z angl. neural unit)

PCA analýza hlavních komponent (z angl. Principal component analysis)

PTSW firma PT Solutions Worldwide s.r.o.

QNN kvadratická neuronová síť (z angl. Quadratic neural network)

QNU kvadratická neuronová jednotka (z angl. Quadratic neural unit)

ReLU rektifikovaná lineární jednotka (z angl. Rectified linear unit)

S-PNN sigma-pi neuronová síť (z angl. Sigma-pi neural network)

SGD Stochastic gradient descent

XOR exkluzivní OR operátor

Kapitola 1

Úvod

Proces válcování kovů je speciálním případem *objemového tváření*. Z polotovaru válcujeme plech na požadovanou tloušťku pomocí *válcovací stolice*.

Pro návrh samotného výrobního postupu válcování plechů je třeba určit přítlačné síly a momenty tak, aby nebyly překročeny maximální hodnoty. V současné době se tyto veličiny určují pomocí velice komplexního matematicko-fyzikálního modelu.

Matematicko-fyzikální model je nejen složitý, ale vstupuje do něho mnoho parametrů, které jsou specifické pro konktrétní valcovaný materiál či válcovací stolici, jejichž číselné hodnoty jsou často určovány empiricky. Drobné nepřesnosti ve vstupech modelu mohou vést k výrazným nepřesnostem modelem vypočtených přítlačných sil a momentů.

Tyto modelem vypočtené přítlačné síly a momenty se konfrontují se skutečnými (změřenými) silami a momenty. Jejich porovnáním[1] vznikají tzv. *korekční faktory*, což jsou hodnoty, které slouží pro úpravu modelem vypočtených přítlačných sil a momentů.

Korekční faktory jsou vypočteny po každém průchodu plechu válcovací stolicí. Nejedná se o konstantu, ale o proměnnou, která značně mění svou hodnotu na základě parametrů konktrétního válcovacího procesu.

1.1 Cíl práce

Cílem této práce je navrhnout, otestovat a implementovat metodu využívající algoritmy strojového učení, parametry polotovaru, předpočítané parametry výrobního postupu válcování a historicky naměřené hodnoty k predikci potřebného korekčního faktoru ještě před samotným průchodem polotovaru válcovací stolicí.

Konktétně definované cíle práce (seřazené chronologicky) tedy jsou:

1. Analýza dat
2. Návrh algoritmů strojového učení
3. Implementace algoritmů do jayzka python a jejich otestování na reálných datech
4. Implementace vybraného a odladěného algoritmu do jazyka C++

[1] Poměrem mezi např. změřenou a vypočtenou sílou.

1.2 Členění práce

Tato práce se pomyslně dělí na teoretickou část (kapitoly 2 - 6) a praktickou část (kapitoly 7 - 9).

V teoretické části jsou popsány teoretické nástroje, potřebné k analyzovaní dat dodaných firmou PTSW (kapitola 2), modely neuronových sítí (kapitoly 3, 4 a 5) a také algoritmy strojového učení (kapitola 6).

V praktické části jsou popsány výsledky datové analýzy (kapitola 7), detailně zdokumentovány (architektura a validace) navržené adaptivní numerické modely (kapitola 8), v závěru praktické části je zdokumentovaný modul v jazyce C++ (kapitola 9).

Kapitola 2

Datová analýza

V této kapitole budou představeny vybrané teoretické nástroje a metody vhodné pro analýzu dat, případně i jejich *předzpracování*.

Pokud vytváříme numerický adaptivní model, který má predikovat korekční faktory v reálném čase, případně se v reálném čase i učit (tzv. *online režim učení*), je žádoucí (z hlediska výpočetní náročnosti) redukovat vstupní prostor na minimum.

Datová analýza nám proto pomůže vybrat potenciálně nejpřínosnější vstupy a odfiltrovat vstupy, které nejsou relevantní.

V neposlední řadě nám analýza vstupního a výstupního prostoru pomůže při volbě architektury adaptivního numerického modelu a také při volbě vhodných učicích algoritmů.

2.1 Korelační analýza

Korelační analýza (*korelační koeficient* případně *korealační matice*) je způsob, kterým můžeme zjistit lineární závislost vstupů našeho numerického adaptivního modelu.

Zásadní je pojem korelační koeficient, což je číslo, které reprezentuje *závislost* případně *nezávislost* dvou mezi sebou porovnávaných vstupů.

Nechť máme dvě náhodné veličiny X a Y, potom jejich *Pearsonův korelační koeficient* bude:

$$corr(X,Y) = \frac{cov(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y} \tag{2.1}$$

Kde, $cov(X,Y)$ je *kovariance*, $E[...]$ je *očekávaná hodnota*[1], μ_X, μ_y jsou *střední hodnoty*[2] a σ_X, σ_Y jsou *směrodatné odchylky.*

Následně platí, že $corr(X,Y) \in \langle -1, 1 \rangle$, přičemž $corr(X,Y) = 1$ znamená nejvyšší pozitivní lineární závislost, $corr(X,Y) = -1$ znamená nejvyšší negativní lineární závislost a $corr(X,Y) = 0$, znamená, že mezi veličinami X a Y není žádná lineární závislost.

Je ovšem velmi důležité si uvědomit, že sice platí, že pokud jsou X a Y *nezávislé*, potom $corr(X,Y) = 0$, ale na opačnou stranu tato implikace neplatí. Jinými slovy pokud jsou veličiny X a Y nekorelované, neznamená to, že jsou nezávislé.

[1]Expected value
[2]Mean value

Korelační matice je čtvercová matice korelačních koeficientů, na jejíž diagonále jsou hodnoty 1[3] a v příslušném řádku a sloupci je vždy příslušný korelační koeficient s odpovídajícími indexy. Nechť jsou X_1, $X_2, \dots, X_n$ náhodné veličiny, potom korelační matice bude ve tvaru:

$$\mathbf{M}_{corr} = \begin{bmatrix} corr(X_1, X_1) & corr(X_1, X_2) & \dots & corr(X_1, X_n) \\ corr(X_2, X_1) & corr(X_2, X_2) & \dots & corr(X_2, X_n) \\ \vdots & \vdots & \ddots & \vdots \\ corr(X_n, X_1) & corr(X_n, X_2) & \dots & corr(X_n, X_n) \end{bmatrix} \tag{2.2}$$

Matice $\mathbf{M}_{corr}$ je navíc *symetrická*[4] a dobře se vizualizuje pomocí *teplotní mapy*.

2.2 Analýza hlavních komponent (PCA)

Analýza hlavních komponent (PCA[5]) je statistický algoritmus, který se se používá k redukci dimenzionality, přičemž se snaží zachovat co největší objem informace [1].

Redukcí dimenzionality vstupního prostoru se samozřejmě sníží i potenciální přesnost adaptivního modelu. Výhodou PCA je, že redukci dimenzionality provádíme tak, aby se potenciální přesnost adaptivního algoritmu ovlivnila co nejméně.

Algoritmus PCA je tedy svým způsobem jak předzpracování dat (transformace), tak datová analýza a přináší mimo popsaných výhod také zrychlení procesu učení [2].

2.2.1 Standardizace

Prvním krokem algoritmu PCA je *standardizace*. Cílem je změnit rozsah našich vstupních proměných tak, aby každá přispívala k analýze PCA rovnoměrně.

Této transformaci se také říká *z-scoring*:

$$x_z = \frac{x - \mu_x}{\sigma_x} \tag{2.3}$$

Kde μ_x je průměrná hodnota a σ_x je směrodatná odchylka.

2.2.2 Kovarianční matice

Druhým krokem tohoto algoritmu je snaha kvantifikovat, jak se vstupní proměnné liší od průměrných hodnot. Jinými slovy, jestli je mezi vstupními daty nějaký vztah.

Kovarianční matice je symetrická matice $n \times n$, kde n je počet dimenzí (počet vzorků měření, počet senzorů, ...):

$$\mathbf{M}_{cov} = \begin{bmatrix} cov(X_1, X_1) & cov(X_1, X_2) & \dots & cov(X_1, X_n) \\ cov(X_2, X_1) & cov(X_2, X_2) & \dots & cov(X_2, X_n) \\ \vdots & \vdots & \ddots & \vdots \\ cov(X_n, X_1) & cov(X_n, X_2) & \dots & cov(X_n, X_n) \end{bmatrix} \tag{2.4}$$

[3] Platí totiž, že veličina je sama se sebou maximálně pozitivně korelovaná.

[4] Protože platí $corr(X, Y) = corr(Y, X)$.

[5] Principal component analysis

Vzhledem k tomu, že platí $cov(X_i, X_i) = var(X_i)$ na diagonále kovarianční matice jsou variance. Jednotlivé kovarianční koeficienty se spočítají podle obecného vztahu vztahu:

$$cov(X,Y) = \frac{1}{M}\sum_{i=1}^{M}(x_i - \overline{x})(y_i - \overline{y}) \tag{2.5}$$

2.2.3 Identifikace hlavních komponent

Posledním krokem této analýzy je identifikace hlavních komponent. Jedním způsobem, jak to lze provést, je zjistit vlastní čísla a vlastní vektory kovarianční matice, kde vlastní čísla nesou informaci o tom, jak jsou komponenty seřazeny podle důležitosti a vlastní vektory nesou informaci o tom, jak máme originální data do těchto komponent transformovat. Máme tedy matici $\mathbf{V}$, kde každý řádek odpovídá vlastnímu vektoru komponenty:

$$\mathbf{V} = [\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n]^T \tag{2.6}$$

A vektor vlastních čísel $\mathbf{\Lambda}$:

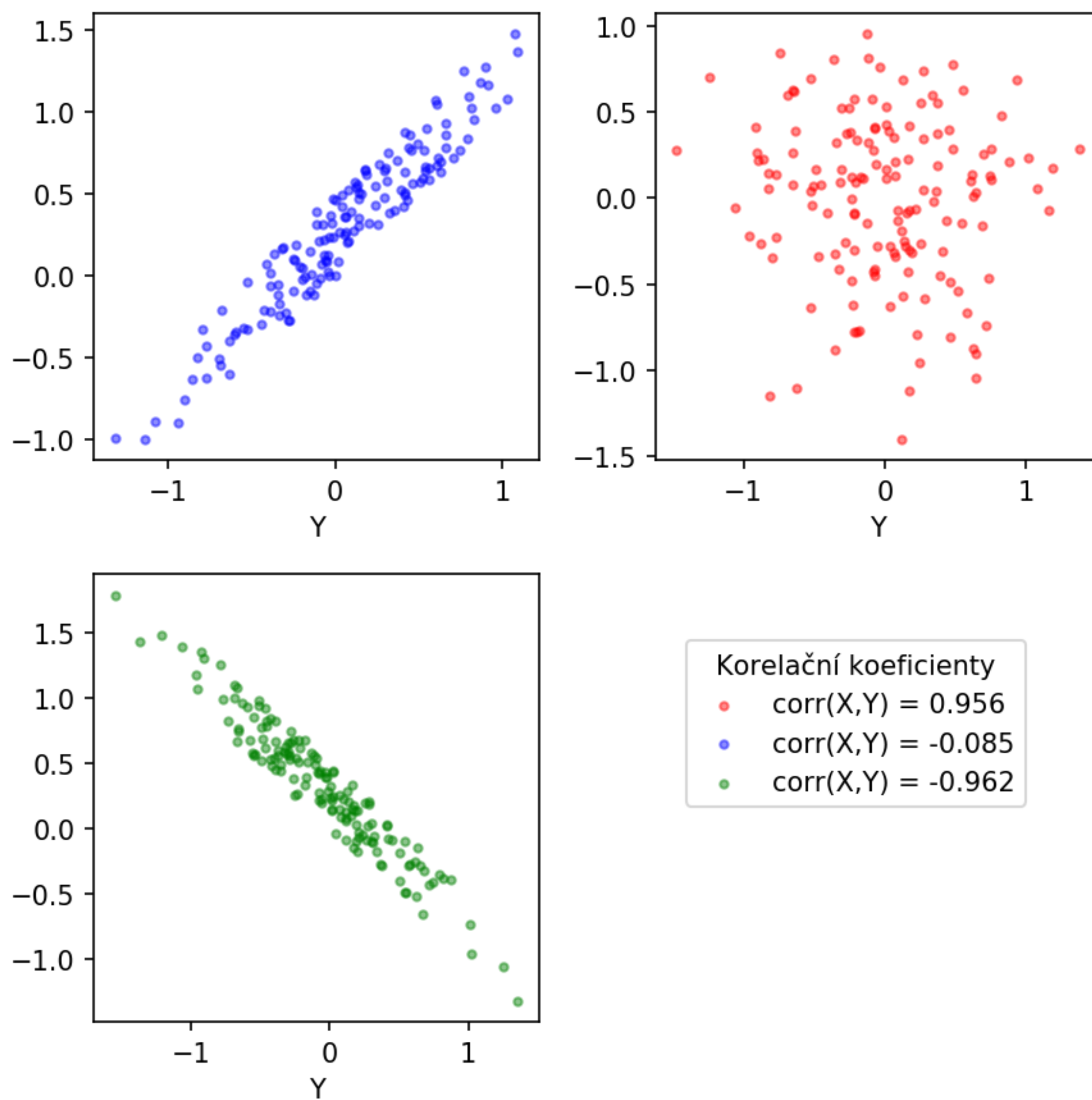
$$\mathbf{\Lambda} = [\lambda_1, \lambda_2, \ldots, \lambda_n]^T \tag{2.7}$$

Potom samotná transformace bude:

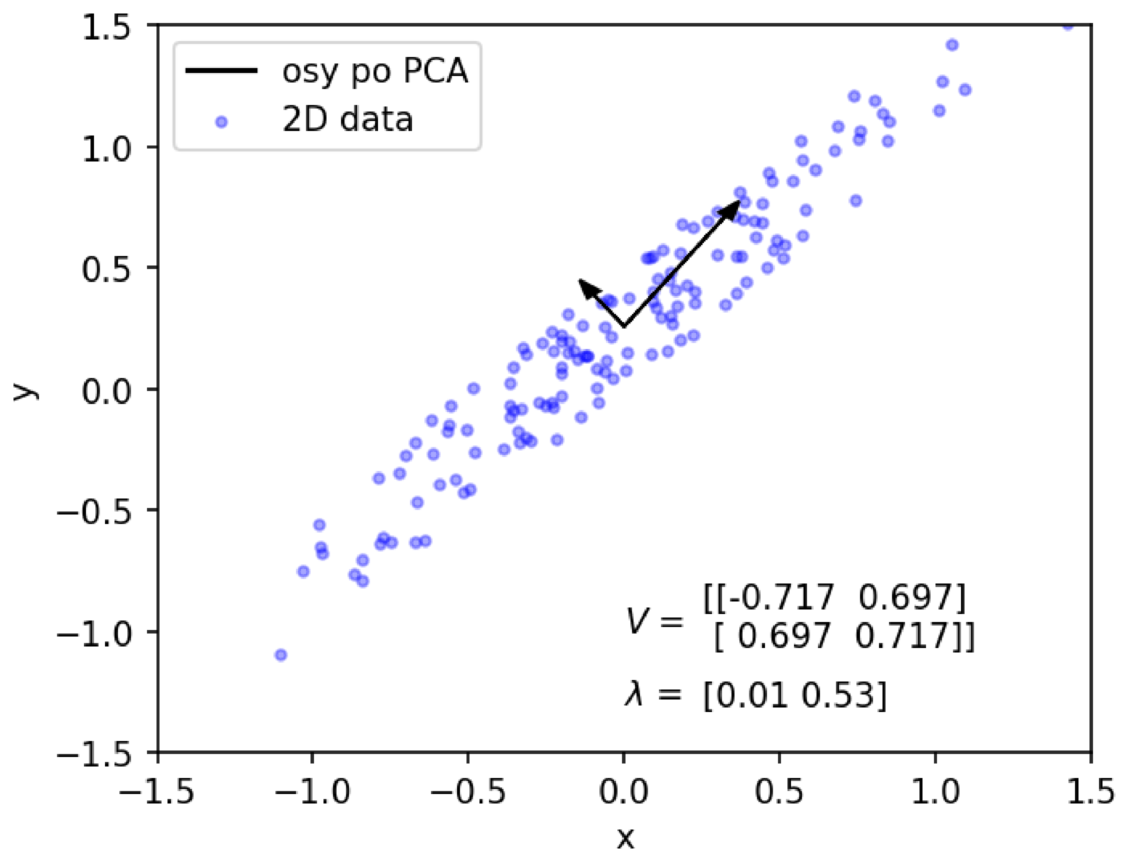
$$\mathbf{X}_t = \mathbf{V}_O \mathbf{X}_z^T \tag{2.8}$$

Kde $\mathbf{V}_O$ jsou vlastní vektory srovnané podle důležitosti komponenty (podle velikosti vlastních čísel) a $\mathbf{X}_z$ jsou standardizovaná původní data.

Jednoduše můžeme redukovat dimenzionalitu tak, že vezmeme pouze k prvních sloupců matice $\mathbf{X}_t$ (transformovaných dat).



Obrázek 2.1: Ilustrativní příklady několika typů rozložení dat a jejich korelačních koeficientů.



Obrázek 2.2: Ilustrační příklad PCA na 2D datech. Matice V jsou vlastní vektory (řádek je vlastní vektor) a vektor λ jsou vlastní čísla.

Kapitola 3

Neuronové sítě

Neuronové sítě (NN) jsou sítě (často zobrazované jako *orientované grafy*), které se skládají z jednoduchých procesorů, které jsou navzájem bohatě propojené. Tento graf propojení se obvykle nazývá *topologie* sítě. Jednoduché procesory nazýváme *neurony* nebo *neuronové jednotky* (NU) z toho důvodu, že velice zjednodušeně modelují chování skutečných biologických neuronů v CNS a jejich matematický popis je inspirován jejich vlastnostmi [3].

Tato kapitola popisuje základní neuronové sítě (MLP) včetně vlastností a krátké historie jejich vzniku. Popisují se zde také některá úskalí např. *lineární separovatelnost množin.*

3.1 Matematický model perceptronu

Perceptron je ve své podstatě nejjednoduší model *dopředné neuronové sítě* a také základní stavební kámen *MLP* (Více-vrstvé perceptronové neuronové sítě[1]).

Obsahuje dvě části. První část se nazývá *Adaptivní lineární kombinátor*, jehož výstup je dán rovnicí:

$$z = \sum_{i=0}^{n} \theta_i \, x_i = \boldsymbol{\theta} \, \mathbf{x} \tag{3.1}$$

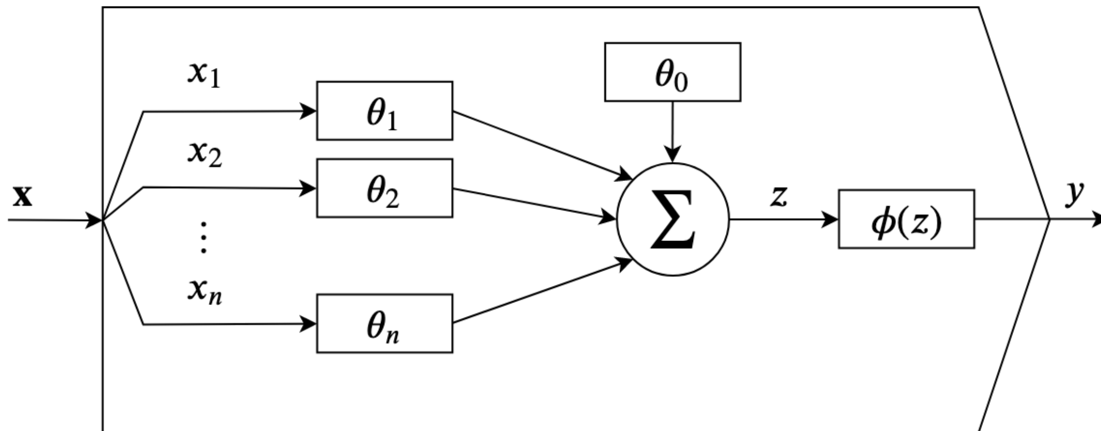
Kde $\mathbf{x}$ je *rozšířený vstupní vektor* ve tvaru $[1, \, x_1, \, x_2, \, \ldots, \, x_n]^T$, což je vektor vstupů rozšířených o 1 na prvním místě. Tato přidaná 1 odpovídá váze θ_0, která simuluje *práh*[2]. $\boldsymbol{\theta}$ je *vektor vah* $[\theta_0, \theta_1, \, \theta_2, \, \ldots, \, \theta_n]$, což jsou adaptivní parametry perceptronu.

Druhá nezbytná součást perceptronu je aktivační (výstupní) funkce $\phi(z)$, která zaručuje nelinearitu výstupu, kompletní tvar rovnice perceptronu tedy bude vypadat takto:

$$y = \phi\left(\sum_{i=0}^{n} \theta_i \, x_i \right) = \phi(\boldsymbol{\theta} \, \mathbf{x}) \tag{3.2}$$

[1]Multi-layer perceptron neural network

[2]Bias



Obrázek 3.1: Schéma modelu perceptronu. $\mathbf{x} = [1, x_1, x_2, \ldots, x_n]^T$ je rozšířený vstupní vektor, $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \ldots, \theta_n]$ je vektor vah a $\phi(z)$ je aktivační funkce. Podobnost s biologickým neuronem je zřejmá na první pohled

3.2 Aktivační (výstupní) funkce

V neuronových sítích jsou do jednotlivých neuronů ve vstupní vrstvě přiváděny číselné hodnoty (vstupy). Každý neuron je definován sadou parametrů[3]. Vektorový součin parametrů a vstupního vektoru je výstup neuronové jednotky. Aktivační funkce je potom *matematická brána* mezi výstupem neuronu a vstupem do další vrstvy, případně konečným výstupem.

Existuje mnoho různých druhů aktivačních funkcí, většina jednoduchých neuronových sítí používá *identitu*, *sigmoidu* nebo *ReLU*, existují však mnohem komplikovanější aktivační funkce jako například *softmax* nebo *leaky ReLU*.

Ačkoli každá jednotlivá neuronová jednotka může mít svoji jedinečnou aktivační funkci, nebývá to zvykem. Obvykle se používají stejné aktivační funkce pro celé vrstvy neuronových sítí.

Důležité je také zmínit, že aktivační funkce jsou prvkem, který přináší požadovanou nelinearitu. Vícevrstvá MLP složená z vrstev perceptronů s lineární aktivační funkcí (identita) má naprosto stejnou aproximační sílu jako obyčejný *lineární kombinátor*.

3.2.1 Identita

Nejjednodušší aktivační funkce je *identita*, která je definovaná vztahem:

$$\phi(z) = z \tag{3.3}$$

A její derivace je:

$$\frac{d\phi(z)}{dz} = 1 \tag{3.4}$$

Tato aktivační funkce se využívá například jako výstupní funkce neuronovýcj jednotek vyšších řádů, u kterých požadovanou nelinearitu přináší kombinace vstupů a není potřeba používát nelineární složité výstupní funkce.

[3]Obvykle nazývané váhy.

3.2.2 Binární aktivační funkce

Binární aktivační funkce je aktivační funkce založená na prahu, což znamená, že pokud je vstupní hodnota nad určitý práh, aktivuje se neuronová jednotka a odešle signál dál.

Zásadním nedostatkem binární aktivační funkce spočívá v tom, že neumožňuje vícehodnotové výstupy.

$$\phi(z) = \begin{cases} 1, & \text{jestliže } z \geq 0 \\ 0, & \text{jinak} \end{cases} \tag{3.5}$$

A její derivace je:

$$\frac{d\phi(z)}{dz} = \begin{cases} 0, & \text{jestliže } z \neq 0 \\ \text{nedefinováno}, & \text{jestliže } z = 0 \end{cases} \tag{3.6}$$

3.2.3 Sigmoida (logistická aktivační funkce)

Z historického hlediska byla sigmoida jedna z prvních používaných aktivačních funkcí.

Výhody této aktivační funkce jsou mimo jiné monotónost na celém intervalu $(-\infty, \infty)$, spojitá derivace a obor hodnot $H(\phi) = (0, 1)$. Je popsaná následujícím vztahem:

$$\phi(z) = \frac{1}{1 + e^{-z}} \tag{3.7}$$

A její derivace je potom:

$$\frac{d\phi(z)}{dz} = \phi(z)\big(1 - \phi(z)\big) \tag{3.8}$$

3.2.4 ReLU (rektifikovaná lineární aktivační funkce)

ReLU si vydobila pověst univerzální, jednoduché a přitom nelineární aktivační funkce. Mimo jiné také vyřešila problém sigmoidálních aktivačních funkcí s přeučením NN. Je popsána vztahem:

$$\phi(z) = \begin{cases} z, & \text{jestliže } z \geq 0 \\ 0, & \text{jinak} \end{cases} \tag{3.9}$$

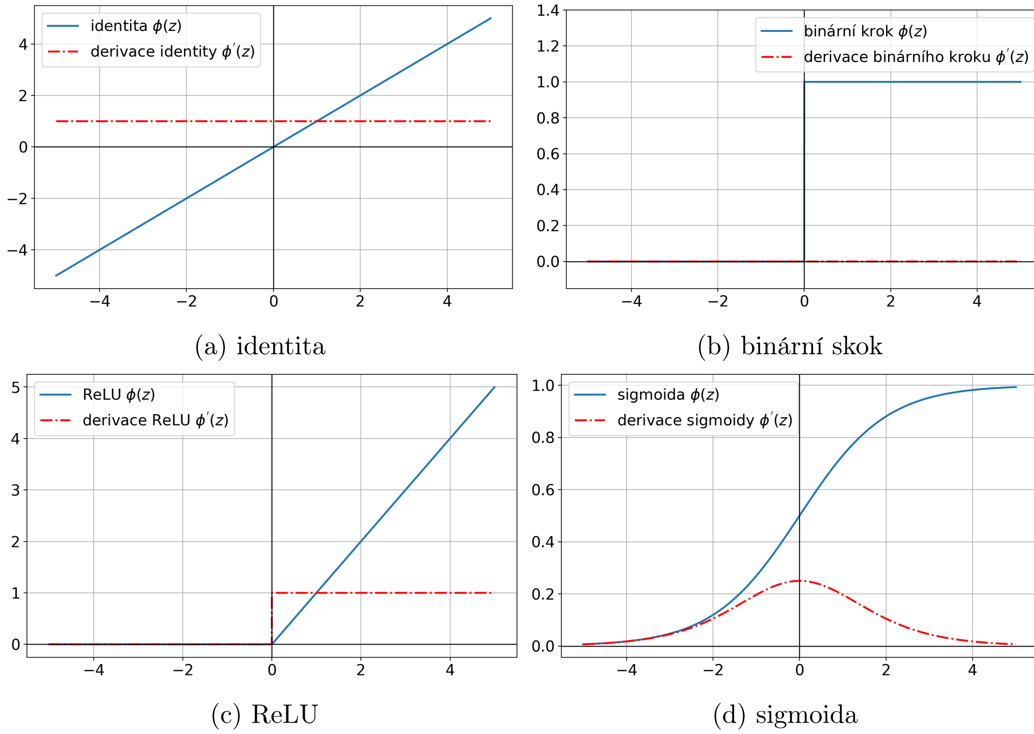
A její derivace je:

$$\frac{d\phi(z)}{dz} = \begin{cases} 1, & \text{jestliže } z > 0 \\ 0, & \text{jestliže } z < 0 \\ \text{nedefinováno}, & z = 0 \end{cases} \tag{3.10}$$

3.3 Vícevrstvé neuronové sítě (MLP)

Za vynálezce perceptronu je považován americký psycholog Frank Rosenblatt. Rosenblatt vyšel z představy neuronu McCullocha-Pittse[4]. Rosenblatův první model byl vlastně jeden jediný perceptron, který dokazál řešit pouze problém s *lineárně separovatelnými množinami.*

[4]Značně zjednodušená představa z roku 1943.



(a) identita (b) binární skok

(c) ReLU (d) sigmoida

Obrázek 3.2: Bežně používáné nebo známé aktivační funkce. Aktivační funkce jsou vyznačeny modře a jejich derivace jsou vyznačeny červeně typem čáry "-.".

Historicky je zmámý *XOR problém*, který ukázal, že jeden perceptron nemůže řešit nelineární klasifikační problémy [4]. Ukázalo se však, že *vícevrstvé neuronové sítě* (MLP) jsou schopné nejen aproximovat funkci XOR, ale také mnoho další nelineárních funkcí.

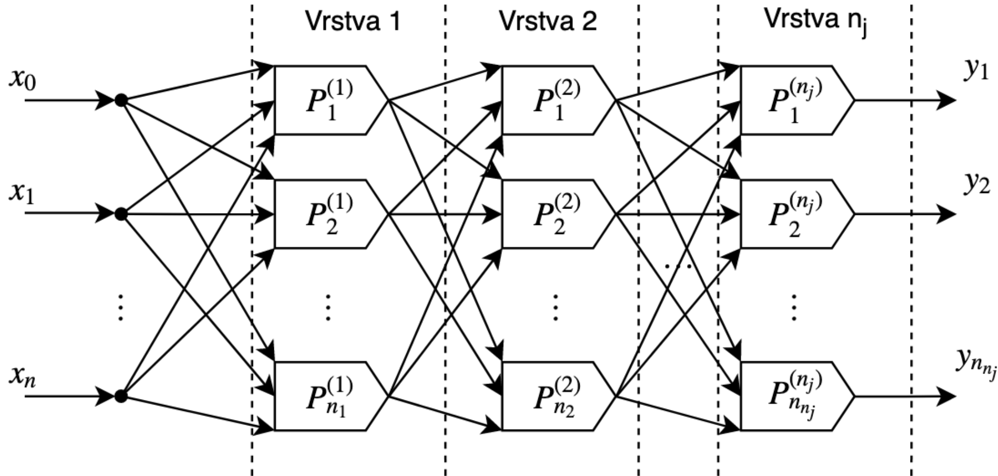
Vicevrstvé neuronové sítě (MLP) se skládají z mnoha perceptronů v každé *vrstvě*, které jsou *plně propojené*. První vrstva se nazývá *vstupní*, prostřední vrstvy se nazývají *skryté* a poslední *výstupní*.

Pro výstup z první vrstvy tedy bude platit:

$$OUT^{(1)} = \phi^{(1)}\left(\mathbf{W^{(1)}}\,\mathbf{x}\right) \tag{3.11}$$

Kde $\mathbf{W^{(1)}}$ je matice vah první vrstvy a $\mathbf{x}$ je rozšířený vstupní vektor:

$$\mathbf{W}^{(1)} = \begin{bmatrix} \theta^{(1)}_{1,0} & \theta^{(1)}_{1,1} & \dots & \theta^{(1)}_{1,n} \\ \theta^{(1)}_{2,0} & \theta^{(1)}_{2,1} & \dots & \theta^{(1)}_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ \theta^{(1)}_{n_1,0} & \theta^{(1)}_{n_1,1} & \dots & \theta^{(1)}_{n_1,n} \end{bmatrix} \quad \mathbf{x} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$



Obrázek 3.3: Schéma obecného modelu vícevrstvé neuronové sítě (MLP). $P_i^{(j)}$ je perceptron číslo i vrstvy s indexem j.

Obecná rovnice Vícevrstvé neuronové sítě (MLP) bude ve tvaru:

$$MLP(\mathbf{x}) = \phi^{(n_j)}\big(\dots \underbrace{\phi^{(2)}\big(\mathbf{W^2} \quad \underbrace{\phi^{(1)}\big(\mathbf{W^{(1)}}\,\mathbf{x}\big)}_{\text{výstup první vrstvy}}\ \big)}_{\text{výstup druhé vrstvy}}\big) \tag{3.12}$$

Tato rovnice je platná za dvou nutných předpokladů: všechny neuronové jednotky v jedné vrstvě mají stejnou aktivační funkci $\phi(.)$ a všechny vrstvy jsou mezi sebou *plně propojené*.

Snadno lze ukázat, že pokud použijeme *identitu* jako aktivační funkci pro všechny vrstvy MLP, degraduje nám vícevrstvá MLP pouze do jedné vrstvy (jinými slovy bude mít aproximační sílu lineárního kombinátoru).

3.4 Backpropagation

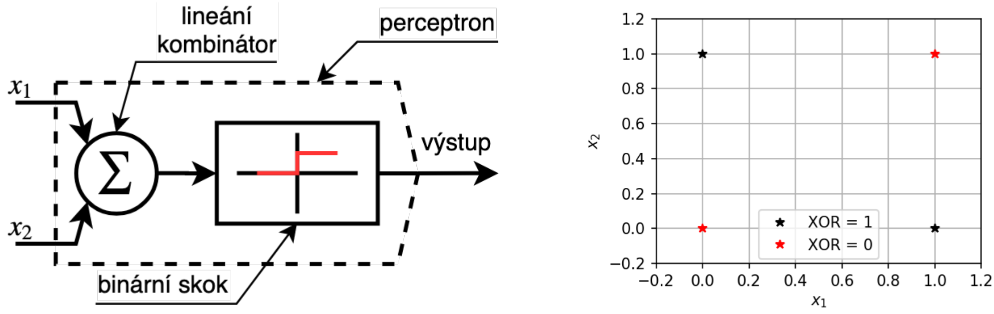
V roce 1969 Minsky a Papert formulovali tzv. XOR problém. Šlo o problém s lineární separovatelností množin. Jednovrstvá neuronová síť (perceptron) není schopná řešit problémy s nelinárně separovatelnými množinami. Počáteční nadšení z v té době ještě čerstvě vymyšlených neurnových sítí opadlo.

Už v té době bylo známo, že některé problémy tohoto typu (nelineárně separovatelné množiny) mohou být řešeny pomocí MLP, zásadní problém byl však v tom, že nikdo netušil, jak tyto modely neuronových sítí učit.

Zlom nastal v roce 1974. Paul J. Werbos představil svůj algoritmus učení, který nazval *backpropagation* (*zpětné šíření chyby*) [5].

3.4.1 Odvození

Uvažujme zjednodušený případ MLP, která má pouze dvě vrstvy. Každá vrstva obsahuje pouze jeden perceptron.



(a) Model perceptronu k formulaci problému lineární separovatelnosti množin funkcí XOR. Dva vstupy x_1 a x_2 s aktivační funkcí binární skok.

(b) Vstupní prostor $\{0, 1\}^2$. Osa x je vstup x_1, osa y je vstup x_2, což jsou vstupy perceptronu. Nelze sestrojit přímku, která by spearovala tyto dvě množiny bodů.

Obrázek 3.4: Problém s binární relací XOR - lineární separovatelnost množin.

Dále nechť je naše chybové kritérum J ve tvaru:

$$J = (a^{(L)} - y)^2 \tag{3.13}$$

Kde $a^{(L)}$ je výstup z poslední vrstvy (index L) a y je požadovaný výstup. Potom z výpočetního grafu sítě (Obr. 3.5) a s pomocí *řetízkového pravidla* dostaneme gradient našeho chybového kritéria. Dále nechť $z^{(L)}$ je výstup MLP před aplikací aktivační funkce a $a^{(L)}$ je výstup po aplikaci aktivační funkce:

$$z^{(L)} = \theta^{(L)} a^{(L-1)} + b^{(L)} \tag{3.14}$$

$$a^{(L)} = \phi(z^{(L)}) \tag{3.15}$$

Potom můžeme pomocí řetízkového pravidla vyjádřit, jakou změnu máme udělat u váhy $\theta^{(L)}$:

$$\frac{\partial J}{\partial \theta^{(L)}} = \frac{\partial z^{(L)}}{\partial \theta^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial J}{\partial a^{(L)}} \tag{3.16}$$

Kde z rovnice 3.13 dostaneme:

$$\frac{\partial J}{\partial a^{(L)}} = 2(a^{(L)} - y) \tag{3.17}$$

Parciální derivace $\frac{\partial a^{(L)}}{\partial z^{(L)}}$ bude záviset na použité aktivační funkci $\phi(.)$:

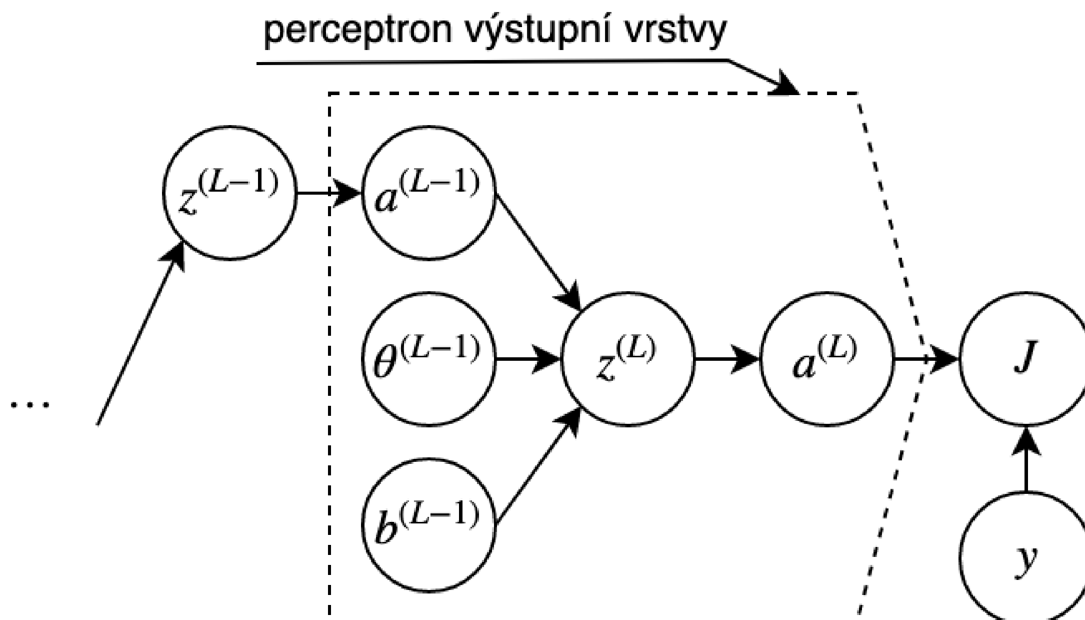
$$\frac{\partial a^{(L)}}{\partial z^{(L)}} = \frac{\partial\big(\phi(z^{(L)})\big)}{\partial z^{(L)}} = \phi'(z^{(L)}) \tag{3.18}$$

A z rovnice 3.14 dostaneme poslední potřebnou parciální derivaci:

$$\frac{\partial z^{(L)}}{\partial \theta^{(L)}} = a^{(L-1)} \tag{3.19}$$

Dosadíme tyto tři parciální derivace do rovnice 3.16:

$$\frac{\partial J}{\partial \theta^{(L)}} = a^{(L-1)} \phi'(z^{(L)}) 2(a^{(L)} - y) \tag{3.20}$$



Obrázek 3.5: Část výpočetního grafu jednoduché MLP. Kde J je chybové kritérium, y je požadovaný výstup, $a^{(L)}$ je výstup vrstvy L, $z^{(L)}$ je výstup vrstvy L před aktivační funkcí, θ je váha a b je práh.

Dostaneme obecný předpis o změně váhy $\theta^{(L)}$ pro minimalizaci chybového kritéria J.

Obdobně bychom dostali podobné vzorce pro všechny parametry MLP sítě. Obecně můžeme tedy psát:

$$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial \theta^{(1)}} & \frac{\partial J}{\partial b^{(1)}} & \cdots & \frac{\partial J}{\partial \theta^{(L)}} & \frac{\partial J}{\partial b^{(L)}} \end{bmatrix}^T \tag{3.21}$$

Je třeba si uvědomit, že vektor ∇J má obvykle ohromné rozměry a učení velkých neuronových sítí vyžaduje velkou výpočetní sílu a kapacitu.

Kapitola 4

Neuronové jednotky vyšších řádů

Pokud uvažujeme model neuronové jednotky popsané v sekci 3.1, mluvíme vlastně o lineárním kombinátoru s aktivační funkcí. Jinými slovy to znamená, že tento model neuronové jednotky je schopen obsáhnout pouze lineární korelace mezi vstupním vektorem a vektorem vah [6].

Je proto intuitivní rozšíření na *neuronové jednotky vyšších řádů* (HONU). To je velmi jednoduchý krok k tomu, aby neuronové jednotky mohli zachytit i korelace vyšších řádů.

U HONU a hlavně NN z HONU složených bylo prokázáno, že mají dobré výpočetní a rozpoznávací vlastnosti, stejně tak jako se dobře učí [6].

4.1 Matematický model HONU

HONU jsou základní stavební kameny HONN, které patří do širší skupiny *S-PNN* (neuronové sítě typu $\Sigma\Pi$). Blokové schéma těchto neuronových jednotek je znázorněno na obrázku 4.1. Obecné HONU můžeme popsat rovnící:

$$y = \phi\left(\theta_0 + \sum_{i_1}^{n} \theta_{i_1} x_{i_1} + \sum_{i_1,i_2}^{n} \theta_{i_1,i_2} x_{i_1} x_{i_2} + \cdots + \sum_{i_1,\ldots,i_N}^{n} \theta_{i_1,\ldots,i_N} x_{i_1} \ldots x_{i_N}\right) \tag{4.1}$$

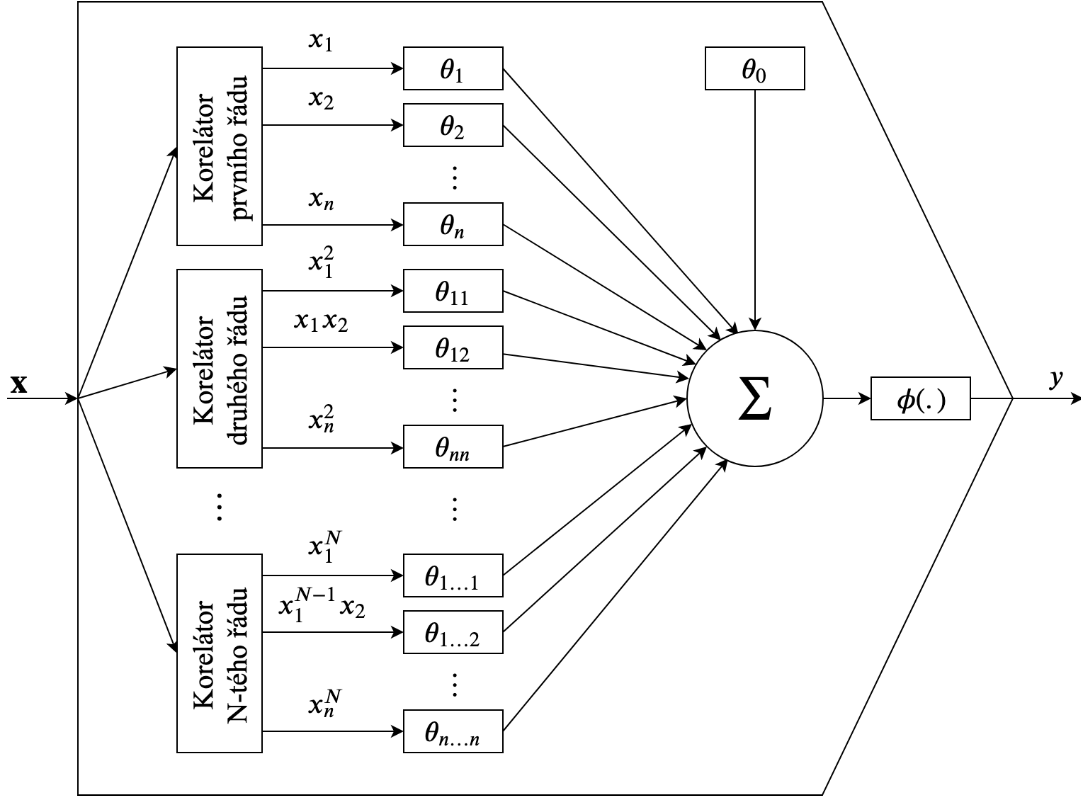
Kde $\mathbf{x} = [x_1, x_2 \ldots, x_n]^T$ je vstupní vektor, y je výstup a $\phi(.)$ je aktivační funkce, která by měla být *striktně monotónní* [6].

Dále může být každá suma z rovnice 4.1 spojena s řádem HONU a platí, že HONU vyššího řádu obsahuje všechny členy HONU nižších řádu:

$$HONU^1(\mathbf{x}) = LNU(\mathbf{x}) = \phi\left(\theta_0 + \sum_{i_1}^{n} \theta_{i_1} x_{i_1}\right) \tag{4.2}$$

Pro HONU druhého řádu bychom mohli použít první dvě sumy z rovnice 4.1:

$$HONU^2(\mathbf{x}) = QNU(\mathbf{x}) = \phi\left(\underbrace{\theta_0 + \sum_{i_1}^{n} \theta_{i_1} x_{i_1}}_{\text{1. řád HONU}} + \sum_{i_1,i_2}^{n} \theta_{i_1,i_2} x_{i_1} x_{i_2}\right) \tag{4.3}$$



Obrázek 4.1: Blokové schéma neuronové jednotky vyššího řádu (HONU). $\mathbf{x}$ je vstupní vektor, θ jsou váhy, $\phi(.)$ je aktivační funkce a y je výstup. Toto schéma bylo volně převzato a mírně upraveno z [6].

A analogicky pro HONU třetího řádu použijeme první tři sumy z rovnice 4.1:

$$HONU^3(\mathbf{x}) = CNU(\mathbf{x}) = \phi\Bigg(\underbrace{\underbrace{\theta_0 + \sum_{i_1}^{n} \theta_{i_1}\, x_{i_1}}_{\text{1. řád HONU}} + \sum_{i_1,i_2}^{n} \theta_{i_1,i_2}\, x_{i_1} x_{i_2}}_{\text{2. řád HONU}} + \sum_{i_1,i_2,i_3}^{n} \theta_{i_1,i_2,i_3}\, x_{i_1} x_{i_2} x_{i_3} \Bigg) \tag{4.4}$$

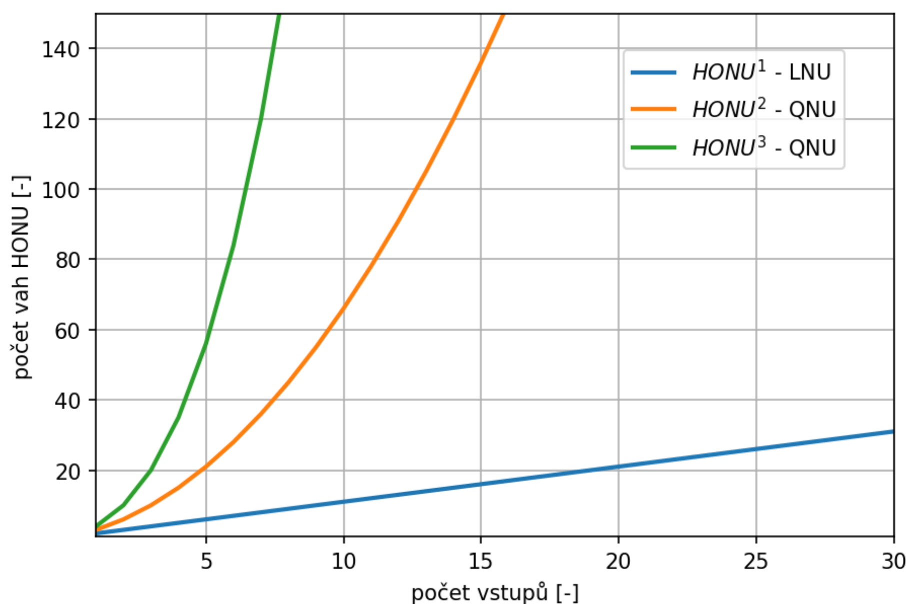
Podle tohoto jednoduchého klíče můžeme postupovat dále a vytvářet HONU vyšších řádů, musíme však dát pozor na to, že počet vah (parametrů) HONU se dramaticky zvyšuje se zvyšováním řádu, což je popsáno v sekci 4.2.

4.2 Výpočetní složitost a problém se sílou aproximace

Výpočetní složitost je jedím ze dvou hlavních problémů HONU. Počet trénovacích parametrů vzhledem k počtu vstupů roste totiž podle tohoto vztahu:

$$\sum_{j=0}^{N} \binom{n+j}{j} = \sum_{j=0}^{N} \frac{(n+j)!}{j!\, n!} \tag{4.5}$$

Kde N je řád HONU a n je počet vstupů a jak můžeme vidět v grafu 4.2, se zvyšujícím se řádem roste počet vah velmi strmě. Druhým problémem je volba správného řádu. Obecně



Obrázek 4.2: Porovnání 3 typů HONU. Závislost počtu trénovacích parametrů (vah) na počtu vstupů pro první tři rády HONU - tedy LNU, QNU a CNU.

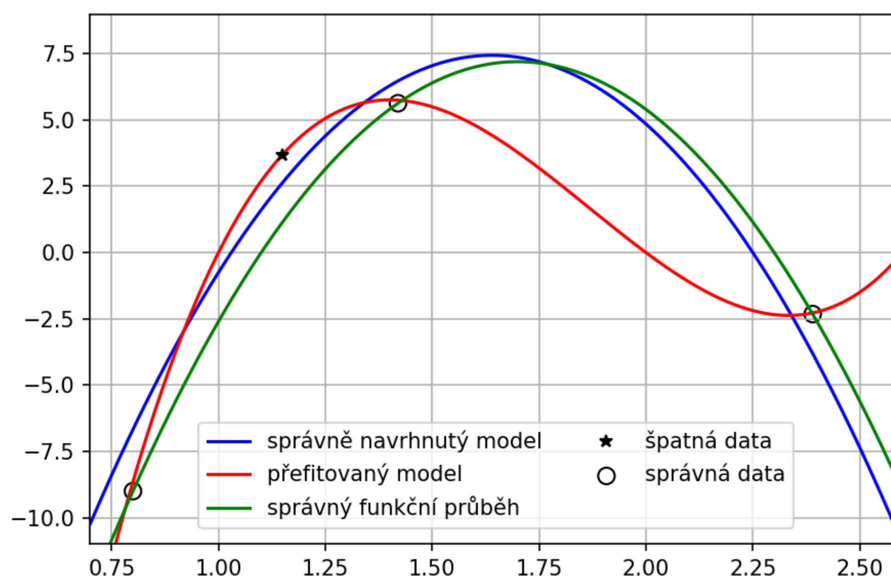
platí, že vyšší řád HONU umožňuje složitější model a při volbě správného učicího algoritmu se správnými hyperparametry také dosahuje přesnější aproximace.

U modelů neuronových sítí však existuje jakási rovnováha mezi aproximační sílou a vhodnou mírou generalizace, což je ilustrováno na obrázku 4.3.

4.3 HONU z pohledu praktického využití

Podiváme se opět na obecnou rovnici HONU (rovnice 4.1). Z této rovnice je patrné, že můžeme velmi lehce vytvořit jakýkoli řád HONU přidáním dalšího sumačního členu, jak však bylo popsáno v sekci 4.2, máme problém s velmi rychle se rozrůstajícím vektorem vah.

Pokud se podíváme na praktické využití, se zvyšujícím se řádem velmi rychle narazíme na strop hardwaru, který máme k dispozici (obrázek 4.2). Proto se v praxi většinou využívají HONU pouze prvního, druhého a třetího řádu.



Obrázek 4.3: Ilustrace tzv. overfittingu (přefitování). Pokud zvolíme moc aproximačně silný model (červená) a naše naměřená trénovací data obsahují chybný vzorek, je vidět, že model ztrácí generalizační schopnost. Naopak správně zvolený model (modrá) zachovává generalizační schopnost a blíží se k správnému funkčnímu průběhu (zelená). Obrázek byl převzat a upraven z [7].

Řád	**Název**	**Používaná zkratka**	**Počet vstupů**	**Počet vah**
1	Lineární NU	LNU	n	$n+1$
2	Kvadratická NU	QNU	n	$\frac{(n+1)(n+2)}{2}$
3	Kubická NU	CNU	n	$\sum_{j=0}^{3}\binom{n+j}{j}$
N	NU N-tého řádu	$HONU^N$	n	$\sum_{j=0}^{N}\binom{n+j}{j}$

Tabulka 4.1: Porovnání nejčastěji používaných řádů HONU (řády 1, 2 a 3). Poslední řádek je obecný pro HONU jakéhokoli řádu.

Kapitola 5

Neuronové sítě vyšších řádů

Neuronové sítě, které se skládají z neuronových jednotek prvního řádu[1], zaznamenaly v minulosti řadu úspěchů (XOR problém, adaptivní řízení, klasifikační a identifikační problémy, ...).

Kombinace vyšších řádů (QNU, CNU, ...) a parametrů (vah) přináší samozřejmě vyšší aproximační výkon, přináši však také řadu závažných problémů. Největším je pravděpodobně velmi rychle rostoucí počet prametrů učení.

Pro zachycení nelinearit vyšších řádů tedy logicky slouží *neuronové sítě vyšších řádů* (HONN). Tyto sítě jsou schopné zachytit nejen nelineární závislosti mezi jednotlivými složkami, ale ukázalo se, že mají dobré výpočetní a rozpoznávací vlastnosti, tak jako splňují tzv. *Stone-Weierstrass* teorém, který je teoretickou nutností *universálních funkčních aproximátorů.* [8].

5.1 Sigma-pi neuronové sítě

HONU, které jsme popsali v kapitole 4, jsou podmnožinou velké skupiny neuronových sítí, které se nazývají *Sigma-pi neuronové sítě* (S-PNN[2]).

Sigma-pi neuronové sítě se skládají ze dvou vrstev. Pi vrstva obsahuje N *součinových členů*, které jsou následně přes váhy θ_i přivedeny do jediného *sumačního členu*, na nějž je následně aplikována aktivační funkce, která se ale obvykle uvažuje jako identita $\phi(z) = z$.

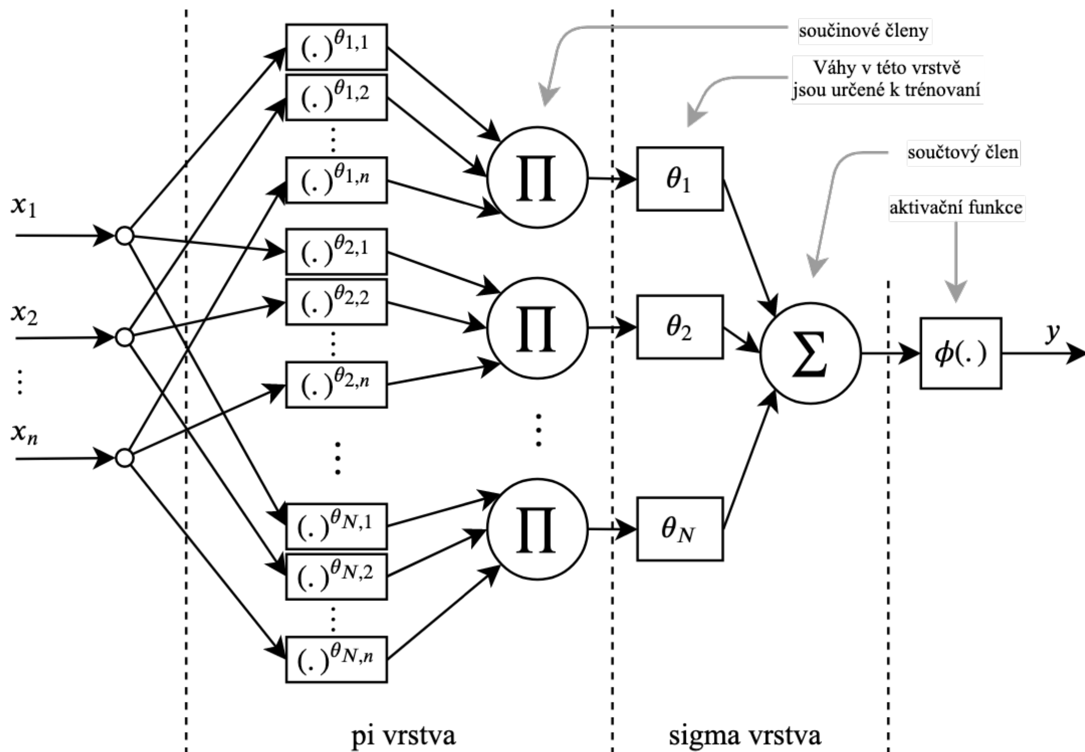
Obecné schéma S-PNN je na obrázku 5.1 a rovnice popisující model takovéto sítě je ve tvaru:

$$y = \phi\left(\sum \theta_i \prod_{j=1}^{n} (x_j)^{\theta_{i,j}} \right) \tag{5.1}$$

Tyto neuronové sítě splňují teorém *Stone-Weierstrass* pokud použijeme aktivační funkci identitu $\phi(z) = z$, což znamená, že jsou universálními aproximátory.

[1] Kterým se obvykle říká perceptrony, LNU nebo pouze lineární kombinátory, pokud je jejich výstupní funkce identita.

[2] Sigma-Pi Neural Network



Obrázek 5.1: Obecné schéma S-PNN. Kde $x_1, x_2, \ldots, x_n$ jsou vstupy a y je výstup. Váhy ve vrstvě sumačního členu θ jsou určeny k trénování, zatímco váhy ve vrstvě součinových členů mají konstatní hodnotu (obvykle jsou to přirozená čísla).

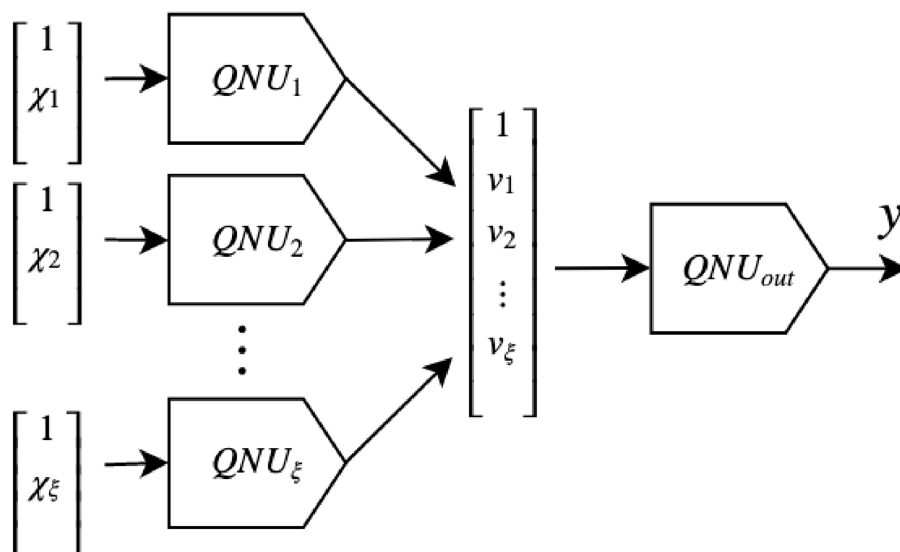
5.2 QNN - Kvadratické neuronové sítě

Speciálním případem HONN je *kvadratická neuronová síť* (*QNN*), jejímž základním stavebním kamenem jsou QNU - kvadratické neuronové jednotky.

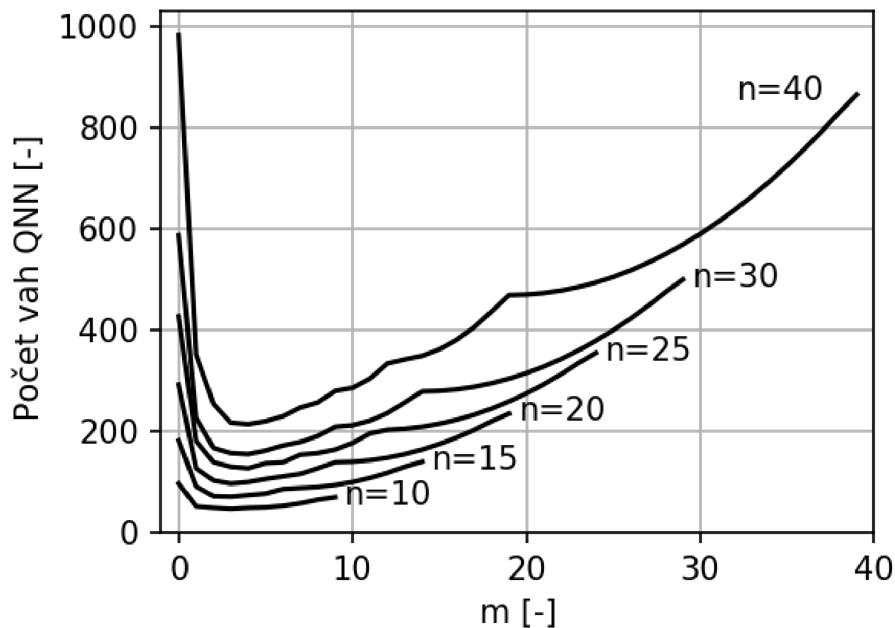
Snadno si lze představit, že v jednoduché MLP síti (obrázek 3.3) nahradíme jednotlivé perceptrony kvadratickými NU (QNU). Takováto síť bude však mít více vah k natrénování a u větších sítí to bude znamenat mnohem větší vektor vah $\boldsymbol{\theta}$.

Jedním z možných způsobů, jak zmenšit počet trénovacích parametrů celého systému, je modifikace struktury sítě odebráním některých propojení. Z plně propojených vrstev se tak stanou jen částečně propojené. Svým způsobem odebereme sítí schopnost vybrat si procesem učení, které vstupy jsou a které nejsou důležité, získáme tím však značně redukovaný vektor vah $\boldsymbol{\theta}$. Tento přístup byl představen v článku, který se věnuje validaci dat naměřených v turbíně [7].

Vstupní prostor byl rozřazen do m shluků a byla použita dvouvrstvá QNN. Podařilo se tak zredukovat počet trénovacích parametrů aniž by se zásadním způsobem ovlivnila prediktivní schopnost modelu.



Obrázek 5.2: Navrhnutý model QNN. Tato síť je složená ze dvou QNU vrstev. Výstupní vrstva obsahuje pouze jediné QNU. Převzato z článku [7].



Obrázek 5.3: Zavislost celkovém počtu vah QNN na počtu použitých QNU vstupní vrstvy. Převzato z článku [7].

Kapitola 6

Učicí algoritmy

V předcházejících kapitolách bylo představeno mnoho různých modelů NN. Jmenovitě MLP, S-PNN, HONU, HONN a jejich deriváty. Tyto matematické modely je možné si představit jako velice složité funkce vstupů (příznaků) a trénovacích parametrů (vah):

$$model(\mathbf{x}) = F(\mathbf{x}, \boldsymbol{\theta}) \tag{6.1}$$

Po sestrojení modelu NN (vytvoření architektury) musíme také najít vektor parametrů $\boldsymbol{\theta}$, který nejlépe odpovídá našemu problému. Tomuto procesu se říká *optimalizace* nebo často také učení či trénování.

Z pohledu matematiky optimalizací rozumíme minimalizaci *chybového kritéria*, které může mít spoustu tvarů, nejjednoduší a zároveň i často používané je *střední kvadratická chyba* (MSE[1]).

Jinými slovy se jedná o algoritmus, který se snaží měnit parametry našeho modelu tak, aby funkcí $F(\mathbf{x}, \boldsymbol{\theta})$ proložil množinu *vstupů-výstupů* tak, že chybové kritérium je co nejmenší.

Tomuto procesu se také říká *učení s učitelem*, protože naše trénovací množina je ve tvaru vstup-výstup a na jednotlivé vstupy známe správné výstupy. Učení s učitelem není jediné učení (např. *shlukové algoritmy* nebo *samoorganizačni mapy* se učí *učením bez učitele*), pro tuto práci však tyto přístupy k učení nejsou podstatné.

Z hlediska učení s učitelem se obvykle rozlišují dvě kategorie, které mají jak své teoretické tak ale i praktické opodstatnění. Jedná se o: *dávkové metody* a *metody vzorek po vzorku*, které jsou popsány v následujících dvou sekcích.

6.1 Dávkové metody

Dávkové metody se jmenují dávkové, jelikož v jedné iteraci učení (trénování) bereme v úvahu celou dávku[2] (matice $\mathbf{X}$).

Často se kvůli hardwarovým omezením celá trénovací data rozdělí do několika dávek, které se střídají v procesu učení.

[1] Mean squared error
[2] Batch

Výhodou těchto metod je, že jsou účinější (počítají s velkým objemem dat najednou). Nevýhodou je, že jsou obvykle výpočetně náročné a často neumožňují nasazení v *online režimu.*

6.1.1 Batch Gradient Descent (BGD)

Metoda *Batch Gradient Descent* (dále jen BGD) je numerický iterativní algoritmus pro hledání funkčního minima[3].

Nechť $\boldsymbol{\theta}$ je vektor parametrů našeho systému, matice $\mathbf{X}$ je matice vstupů[4] a $\mathbf{y}$ je vektor výstupů. Dále předpokládejme, že funkce $h_\theta(\mathbf{x})$ je funkcí, jejíž parametry $\boldsymbol{\theta}$ chceme optimalizovat:

$$h_\theta(\mathbf{x}) = \sum_{j=0}^{n-1} \theta_j x_j \tag{6.2}$$

Naše *chybové kritérium* bude definováno následovně[5]:

$$J_{train}(\boldsymbol{\theta}) = \frac{1}{M} \sum_{i=1}^{M} \frac{1}{2} \big(h_\theta(\mathbf{x}^{(i)}) - y^{(i)}\big)^2 \tag{6.3}$$

Aktualizační pravidlo[6]:

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha \frac{\partial}{\partial \boldsymbol{\theta}} J_{train}(\boldsymbol{\theta}^{(k)}) \tag{6.4}$$

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha \frac{1}{M} \sum_{i=1}^{M} \big(h_\theta(\mathbf{x}^{(i)}) - y^{(i)}\big) \mathbf{x}^{(i)} \tag{6.5}$$

Kde k značí iteraci, M je počet řádků matice $\mathbf{X}$ a α je parametr, který ovlivňuje délku kroku[7].

6.1.1.1 Pseudokód

Algoritmus 1 Batch Gradient Descent

1: **function** BGD($\mathbf{X}, \mathbf{y}, \alpha, k_{max}$)
2: **inicializace** θ ▷ náhodný nebo nulový vektor
3: **for** $k \leftarrow 1, \ldots, k_{max}$ **do** ▷ k_{max} - počet iterací
4: $\mathbf{e} \leftarrow \mathbf{X}\boldsymbol{\theta} - \mathbf{y}$ ▷ vektor chyb
5: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \frac{\alpha}{M}\mathbf{X}^T\mathbf{e}$ ▷ aktualizace parametrů θ
6: **return** θ

[3] V našem případě hledáme funkční minimum J_{train}.
[4] Jeden řádek matice $\mathbf{X}$ odpovídá jedné realizaci vstupů. Tomu samozřejmě odpovídá řádek sloupcového vektoru $\mathbf{y}$.
[5]
[6] Update Rule
[7] V různé literatuře se tomuto parametru říká jinak. Například *faktor zapomínání*, *learning rate*, ...

6.1.2 Stochastic Gradient Descent (SGD)

Při implementaci algoritmu BGD (subsekce 6.1.1) se ovšem potýkáme s několika problémy, přičemž největším z nich je výpočetní náročnost *chybového kritéria* (rovnice 6.3). Jakmile totiž stoupá složitost našeho parametrického systému[8], stoupá i výpočetní náročnost. V každém kroku totiž musíme sečít jednotlivé chybové funkce všechn řádků matice **X**.

Algoritmus *Stochastický gradient descent* (SGD) v jendom kroku počítá pouze s jedním řádkem matice **X**, který je náhodně vybrán, což je fundamentálně důležitým prvkem algoritmu.

Naše funkce, jejíž parametry chceme optimalizovat, bude ve stejném tvaru jako u algoritmu SGD (rovnice 6.2), tedy:

$$h_\theta(\mathbf{x}) = \sum_{j=0}^{n-1} \theta_j x_j \tag{6.6}$$

Dále zavedeme funkci *cost*:

$$cost(\boldsymbol{\theta}, (\mathbf{x}^{(i)}, y^{(i)})) = \frac{1}{2}\big[h_\theta(\mathbf{x}^{(i)}) - y^{(i)}\big]^2 \tag{6.7}$$

A naše chybová funkce bude ve tvaru[9]:

$$J_\theta = \frac{1}{M} \sum_{i=1}^{M} cost\big(\boldsymbol{\theta}, (\mathbf{x}^{(i)}, y^{(i)})\big) \tag{6.8}$$

Potom aktualizační pravidlo:

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha \frac{\partial}{\partial \boldsymbol{\theta}} cost\big(\boldsymbol{\theta}, (\mathbf{x}^{(k)}, y^{(k)})\big) \tag{6.9}$$

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha\big(h_\theta(\mathbf{x}^{(k)}) - y(k)\big)\mathbf{x}^{(k)} \tag{6.10}$$

Kde k je opět počet iterací a parametr α ovlivňuje délku kroku. Nejdůležitější součástí této metody[10] je proces *náhodného výběru* před každdou iterací. Lépe je tento postup znázorněn v pseudokódu algoritmu SGD (algoritmus 2).

výhody	**nevýhody**
vysoká efektivita nižší výpočetní náročnost(ve srovnání s BGD) velmi jednoduchá implementace dobře fungující se stoupající složitostí a velikostí	mnoho hyperparametrů[11] velmi senzitivní na velikost vstupů v některých případech „krouží“ kolem minima

Tabulka 6.1: Srovnání výhod a nevýhod metody SGD

[8] Jinými slovy roste velikost vektoru $\boldsymbol{\theta}$ a také velikost matice **X**.

[9] Povšiměte si, že je je naprosto stejná jako u algoritmu BGD(rovnice 6.3)

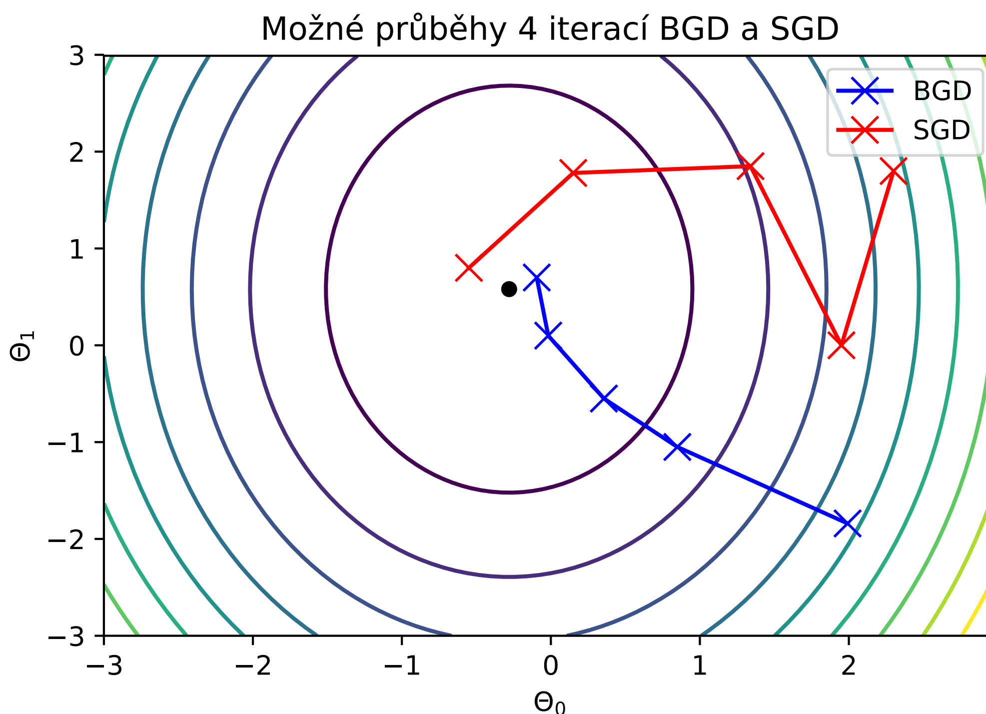
[10] Proto se tato metoda jmenuje *Stochastic Gradient Descent*, ze slova stochastický nebo také náhodný.

[11] Jako například regularizační parametr, počet iteraci, ...

6.1.2.1 Pseudokód

Algoritmus 2 Stochastic Gradient Descent

1: **function** SGD($\mathbf{X}, \mathbf{y}, \alpha, k_{max}$)
2: **inicializace** θ ▷ náhodný nebo nulový vektor
3: **for** $k \leftarrow 1, \ldots, k_{max}$ **do** ▷ k_{max} - počet iterací
4: $(\mathbf{x}^{(k)}, y^{(k)}) \leftarrow$ náhodný výběr z $(\mathbf{X}, \mathbf{y})$
5: $e \leftarrow \mathbf{x}^{(k)}\boldsymbol{\theta} - y^{(k)}$ ▷ e - chyba, skalár
6: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha e \mathbf{x}^{(k)}$ ▷ aktualizace parametrů
7: **return** θ



Obrázek 6.1: Ilustrační srovnání algoritmů BGD a SGD, na osách x a y jsou parametry θ_0 a θ_1. Pomyslná osa z (znázorněná jako barevné vrstevnice) je chybová funkce J_θ. Černý bod v souřadnících $[-0.28, 0.58]$ je lokální minimum.

6.1.3 Konjugované gradienty (CG)

Dalším z optimalizačních algoritmů z rodiny *gradientních metod* jsou *konjugované gradienty* (CG).

Největší výhodou algoritmu CG je, že k dosažení přijatelného[12] minima je obvykle potřeba řádově mnohem méně iterací, než v algoritmech v předchozích subsekcích (6.1.1 a

[12] Myšleno s přijatelnou chybou.

6.1.2). Navíc (jak je vidět v algoritmu 3) je iterativní výpočet konjugovaných vektorů ve své podstatě velmi ekonomický[13].

6.1.3.1 Teorie

Metoda konjugovaných gradientů je metoda používaná k nalezení řešení (vektoru θ) výrazu:

$$\mathbf{y} = \mathbf{X}\boldsymbol{\theta} \tag{6.11}$$

Nebo jinými slovy, náš problém můžeme transformovat na nalezení kořenů výrazu:

$$\mathbf{X}\boldsymbol{\theta} - \mathbf{y} \tag{6.12}$$

V první řadě je potřeba zavést koncept tzv. *konjugovaných* vektorů:

$$\langle \mathbf{u}, \mathbf{v} \rangle_{\mathbf{Q}} = \mathbf{u}^T \mathbf{Q} \mathbf{v} \tag{6.13}$$

A budeme říkat, že tyto dva vektory jsou konjugované v závislosti na $\mathbf{Q}$ (nebo také *Q-ortogonální*), pokud je výsledek vztahu (rovnice 6.13) 0. Přičemž řešení můžeme vždy psát ve tvaru:

$$\boldsymbol{\theta}^* = \sum_{i=1}^{n} \alpha_i \mathbf{p}_i \tag{6.14}$$

Pokud se vrátíme k našemu problému formulovaném v rovnici 6.11, obě straný rovnice vynásobíme vektorem $\mathbf{p}_k^T$ (nějakým *Q-ortogonálním* vektorem):

$$\mathbf{p}_k^T \mathbf{y} = \mathbf{p}_k^T \mathbf{Q} \boldsymbol{\theta} \tag{6.15}$$

$$\mathbf{p}_k^T \mathbf{y} = \mathbf{p}_k^T \mathbf{Q} \sum_{i=1}^{n} \alpha_i \mathbf{p}_i \tag{6.16}$$

Nyní použijeme Q-ortogonalitu, a jelikož platí:

$$\forall \mathbf{p}_i (i \neq k \implies \alpha_i \mathbf{p}_k^T \mathbf{Q} \mathbf{p}_i = 0)$$

Jinými slovy, pokud jsou dva vektoy $\mathbf{p}_k$ a $\mathbf{p}_i$ Q-ortogonální, součin je roven 0 a můžeme je z rovnice vypustit.Zbyde nám tedy pouze vektor $\mathbf{p}_k$ a dostáváme tedy výraz:

$$\mathbf{p}_k^T \mathbf{y} = \alpha_k \mathbf{p}_k^T \mathbf{Q} \mathbf{p}_k \tag{6.17}$$

Konečně tedy můžeme vyjádřit koeficienty α_k:

$$\alpha_k = \frac{\mathbf{p}_k^T \mathbf{y}}{\mathbf{p}_k^T \mathbf{Q} \mathbf{p}_k} \tag{6.18}$$

[13]K výpočtu konjugovaného vektoru v $k+1$ je potřeba znát pouze konjugovaný vektor v k.

6.1.3.2 Pseudokód

Algoritmus 3 Konjugované Gradienty

1: **function** CONG($\mathbf{X}, \mathbf{y}, k_{max}$)
2: **inicializace** θ_0 ▷ náhodný nebo nulový vektor
3: $\mathbf{r}_0 \leftarrow \mathbf{y} - \mathbf{X}\boldsymbol{\theta}$
4: $\mathbf{p}_0 \leftarrow \mathbf{r}_0$
5: **for** $k \leftarrow 0, \ldots, k_{max} - 1$ **do**
6: $\alpha_k \leftarrow \frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{p}_k^T \mathbf{X} \mathbf{p}_k}$
7: $\boldsymbol{\theta}_{k+1} \leftarrow \boldsymbol{\theta}_k + \alpha_k \mathbf{p}_k$
8: $\mathbf{r}_{k+1} \leftarrow -\alpha_k \mathbf{X} \mathbf{p}_k$
9: **if** $\mathbf{r}_{k+1}$ je dostatečně malé **then**
10: opusť for-smyčku
11: **else**
12: $\beta \leftarrow \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k}$
13: $\mathbf{p}_{k+1} \leftarrow \mathbf{r}_{k+1} + \beta_k \mathbf{p}_k$
14: **return** $\boldsymbol{\theta}_{k+1}$

6.1.4 Gauss-Newtonova metoda

Gauss-Newtonova metoda (GNA) je numerický iterativní algoritmus, který se používá pro nalezení funkčního minima.

6.1.4.1 Teorie

Naším cílem je přizpůsobit množinu M bodů z $\{(\mathbf{x}_i, y_i), \mathbf{x} \in \mathbf{X} \ a \ y_i \in \mathbf{y}\}$ pomocí nelineární funkce:

$$y = f(\mathbf{x}, \boldsymbol{\theta}) \tag{6.19}$$

Přičemž chceme nalézt takový vektor parametrů $\boldsymbol{\theta}$, který bude minimalizovat naše chybové kritérium ve tvaru:

$$J = \sum_{M}^{i=1} e_i^2 = \sum_{M}^{i=1} \left[y_i - f(\mathbf{x}, \boldsymbol{\theta})\right]^2 \tag{6.20}$$

Kde $\mathbf{e}$ je *residuum* a může být zapsané ve vektorové formě:

$$\mathbf{e} = \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_M \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_M \end{bmatrix} - \begin{bmatrix} f(\mathbf{x}_1, \boldsymbol{\theta}) \\ f(\mathbf{x}_2, \boldsymbol{\theta}) \\ \vdots \\ f(\mathbf{x}_M, \boldsymbol{\theta}) \end{bmatrix} = \mathbf{y} - f(\boldsymbol{\theta}) \tag{6.21}$$

Pokud zapíšeme i chybové kritérium ve vektorové formě:

$$J = \sum_{M}^{i=1} e_i^2 = \mathbf{e}^T \mathbf{e} = \left\|\mathbf{e}^2\right\| = \|\mathbf{y} - \mathbf{f}(\boldsymbol{\theta})\|^2 \tag{6.22}$$

Můžeme psát aktualizační obecné aktualizační pravidlo:

$$\boldsymbol{\theta}^{l+1} = \boldsymbol{\theta}^{l} + \Delta\boldsymbol{\theta}^{l} \tag{6.23}$$

Kde $\Delta\boldsymbol{\theta}^{\,l}$:

$$\Delta\boldsymbol{\theta}^{l} = -\big[\mathbf{J}^T\mathbf{J}\big]\mathbf{J}^T\mathbf{e} \tag{6.24}$$

Kde jestliže naše funkce $f(\mathbf{x}, \boldsymbol{\theta})$ bude ve tvaru $\sum_{i=0}^{n} \theta_i \, x_i$, potom $\Delta\boldsymbol{\theta}^{\,l}$ bude ve tvaru:

$$\Delta\boldsymbol{\theta}^{l} = -\big[\mathbf{X}^T\mathbf{X}\big]\mathbf{X}^T\mathbf{e} \tag{6.25}$$

6.1.4.2 Pseudokód

Algoritmus 4 Gauss-Newtonova metoda

1: **function** GNA($\mathbf{X}, \mathbf{y}, k_{max}$)
2: **inicializace** $\boldsymbol{\theta}$ ▷ náhodný nebo nulový vektor
3: **for** $k \leftarrow 1, \ldots, k_{max}$ **do** ▷ k_{max} - počet iterací
4: $\mathbf{e} \leftarrow \mathbf{X}\boldsymbol{\theta} - \mathbf{y}$
5: $\Delta\boldsymbol{\theta} \leftarrow \big[\mathbf{X}^T\mathbf{X}\big]^{-1}\mathbf{X}\mathbf{e}$
6: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$ ▷ aktualizace parametrů
7: **return** θ

6.1.5 Levenberg-Marquardtova metoda

Levenberg-Marquardtova metoda (LMA) je numerický iterativní algoritmus pro hledání minima funkce, která kombinuje Gauss-Newtonovu metodu a metodu gradient descent tak, aby byly eliminovány problémy Gauss-Newtonovy metody daleko od extrému [9]. Tato metoda v sobě kombinuje klíčové vlastnosti jako jsou robustnost, efektivnost, konvergenci a numerickou stabilitu [10].

Aktualizační pravidlo při použití metody LMA pro HONU je:

$$\boldsymbol{\theta}^{(l+1)} = \boldsymbol{\theta}^{(l)} + \big[\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I}\big]^{-1}\mathbf{X}\mathbf{e} \tag{6.26}$$

Kde λ je *tlumící faktor*[14] a v podstatě určuje poměr metod, které LMA kombinuje, $\mathbf{X}$ je matice vstupů, $\mathbf{I}$ je jednotková matice a $\mathbf{e}$ je vektor chyb našeho modelu podle:

$$\mathbf{e} = \mathbf{X}\,\boldsymbol{\theta} - \mathbf{y} \tag{6.27}$$

Kde $\mathbf{y}$ je vektor výstupů.

[14] Dumping factor

6.1.5.1 Pseudokód

Algoritmus 5 Levenberg-Marquardt

```
1: function LMA(X, y, λ, k_max)
2:     inicializace θ                          ▷ náhodný nebo nulový vektor
3:     for k ← 1, ..., k_max do                ▷ k_max - počet iterací
4:         e ← Xθ − y
5:         Δθ ← [X^T X + λI]^{-1} Xe
6:         θ ← θ + Δθ                          ▷ aktualizace parametrů
7:     return θ
```

6.2 Metody vzorek po vzorku

Učící algoritmy typu *vzorek po vzorku* jsou stejně jako dávkové učící metody numerické iterativní algoritmy. Narozdíl od dávkových však v jedné iteraci učení pracují pouze s jedním párem typu vstup-výstup (vzorkem).

V praxi to znamená, že proces je mnohem rychlejší a je tím pádem vhodný k online učení (učení za běhu), častěji se však objevují problémy s konvergencí.

6.2.1 Gradient Descent

V subsekcích 6.1.1 a 6.1.2 byly popsány dvě gradientní metody (BGD a SGD). Existuje samozřejmě i nedávková varianta metody nejmenšího spádu, obvykle nazývaná pouze *gradient descent* nebo také SBSGD[15].

Tato metoda je vlastně BGD nebo SGD pouze s jedním trénovacím vzorkem. Můžeme tedy aktualizační pravidlo psát ve tvaru:

$$\boldsymbol{\theta}^{(k+1)} = \boldsymbol{\theta}^{(k)} - \alpha\big(h_\theta(\mathbf{x}) - y\big)\mathbf{x} \tag{6.28}$$

Kde $\mathbf{x}, y$ je trénovací vzorek typu vstup výstup.

6.2.1.1 Pseudokód

Algoritmus 6 SBS Gradient Descent

```
1: function SBSGD(θ, x, y, α)
2:     e ← xθ − y
3:     Δθ ← −α e x^T
4:     θ ← θ + Δθ                              ▷ aktualizace parametrů
5:     return θ
```

[15] Sample by sample gradient descent

Kapitola 7

Analýza dat z procesu válcování

Matematicko-fyzikální model válcování je komplexní a složitý. Potenciálních vstupů adaptivního numerického algoritmu určeného ke korekci MF modelu je tedy mnoho a jejich počet je někdy nutné redukovat.

Tato redukce je nutná pro snížení výpočetní komplexnosti a tím i značnému zrychlení algoritmů učení a výpočtů.

Vedlejším produktem této redukce může také být zvýšení míry porozumnění MF modelu.

Firma PTSW dodala naměřená a MF modelem vypočtená data. Celkem tedy bylo k dispozici 17 potenciálních vstupů a 2 výstupy (silový a momentový korekční faktor). Jedná se o tyto potenciální vstupy:

1. **vstupní tloušťka** - vstupní tloušťka válcovaného polotovaru
2. **výstupní tloušťka** - výstupní tloušťka válcovaného polotovaru
3. **šířka** - šířka válcovaného polotovaru
4. **síla změřená** - naměřená síla
5. **síla vypočtená** - vypočtená síla MF modelem
6. **průhyb** - průhyb válcovací stolice
7. **moment změřený** - naměřený moment
8. **moment vypočtený** - vypočtený moment MF modelem
9. **rychlost** - rychlost průjezdu polotovaru válcovací stolicí
10. **teplota** - teplota válcovaného polotovaru
11. **napětí**
12. **napěťový poměr**
13. **poloha** - vzájemná poloha válcovací stolice a polotovaru

14. **kontaktní délka**
15. **konstanta Kf** - empirická konstanta válcování

Firma PTSW také sestavila dvě potenciální kombinace vstupů, které by podle expertů na válcování ve firmě PTSW měly mít největší vliv na korekční faktory:

Kombinace 1	**Kombinace 2**
Šířka	Vstupní tloušťka
Teplota	Výstupní tloušťka
Napětí	Šířka
Napěťový poměr	Rychlost
Poloha	Teplota

Tabulka 7.1: Doporučené kombinace vstupů podle expertů na válcování z firmy PTSW.

7.1 Kombinace vstupů 1

Kombinace vstupů 1 (z tabulky 7.1) byla podrobena korelační analýze a také analýze PCA.

Z korelační analýzy (Obrázek 7.1a) je na první podhled patrné, že některé vstupy jsou spolu korelované. Například korelační koeficient mezi vstupy napětí a napěťový poměr je blízký -1.

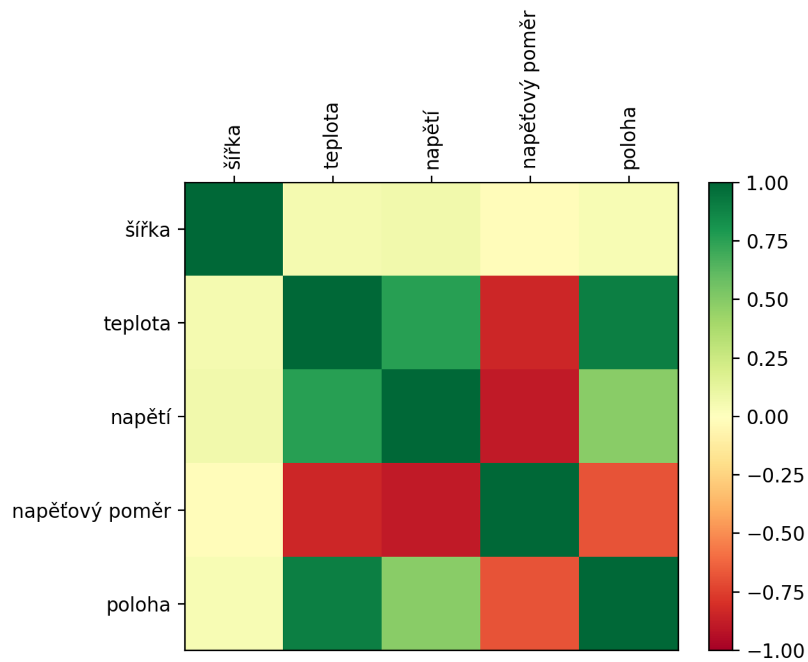
Analýza PCA (Obrázek 7.2) ukázala, že je existuje lineární závislost mezi vstupy napětí a poloha, bylo by tedy pravděpodobně výhodné jeden z těchto vstupů odebrat a nahradit ho jiným.

7.2 Kombinace vstupů 2

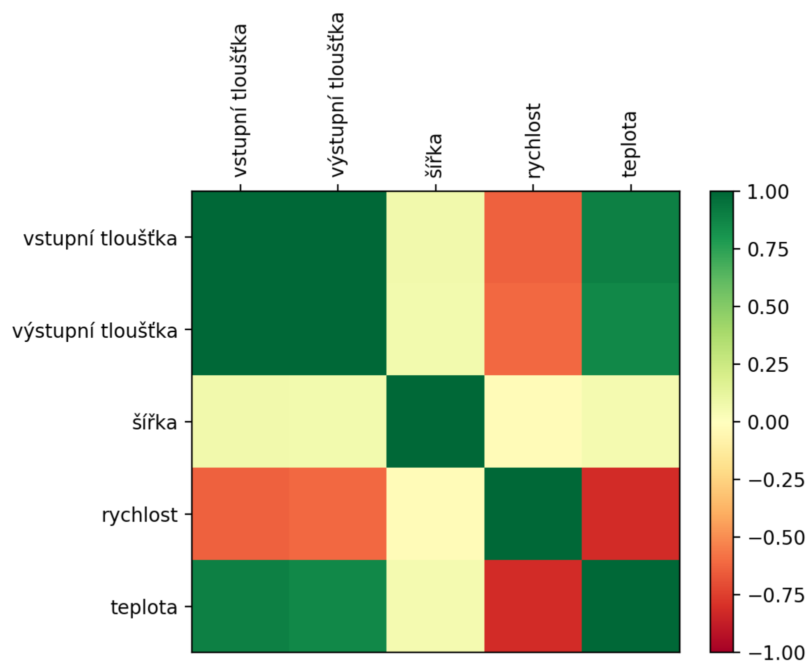
Kombinace vstupů 2 (z tabulky 7.1) byla také podrobena korelační analýze a analýze PCA.

U kombinace vstupů 2 jsou vysoce pozitivně korelované vstupy vstupní tloušťka a výstupní tloušťka. Bylo by pravděpodobně lepší tyto dva vstupy nahradit jejich rozdílem, nebo poměrem.

Analýza PCA (obrázek 7.3) naopak ukázala, že všechny vstupy jsou provázané.

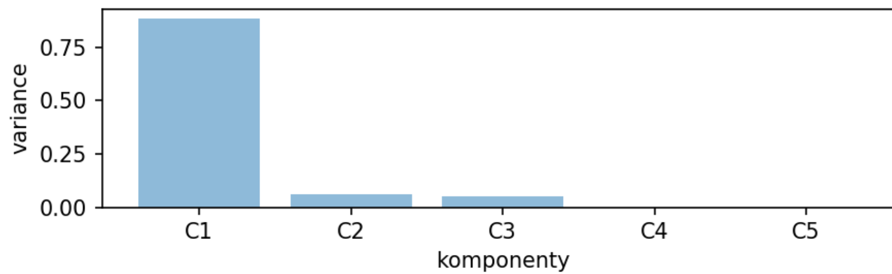


(a) Korelační matice kombinace vstupů 1 vizualizovaná pomocí teplotní mapy.

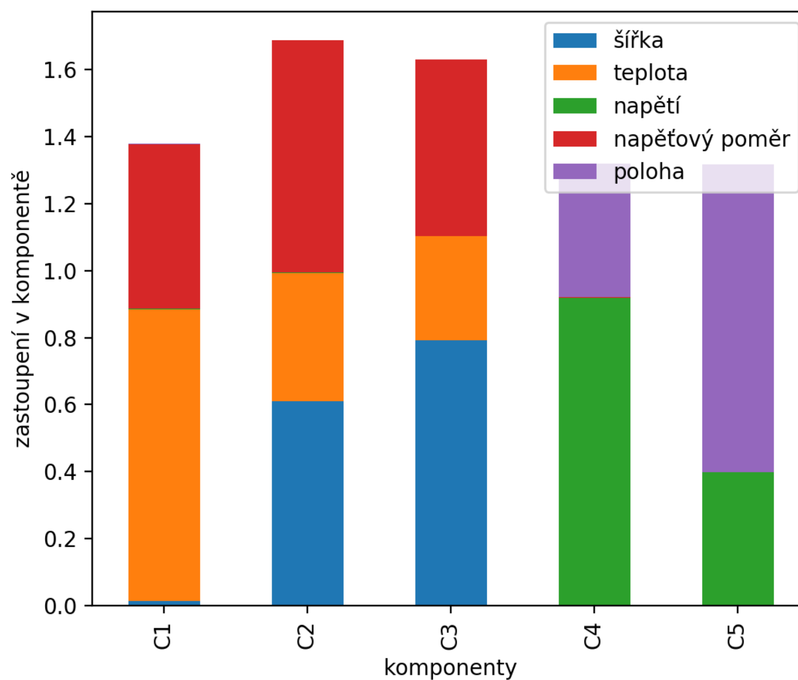


(b) Korelační matice kombinace vstupů 2 vizualizovaná pomocí teplotní mapy.

Obrázek 7.1: Korelační komatice u kombinací vstupů 1 a 2.

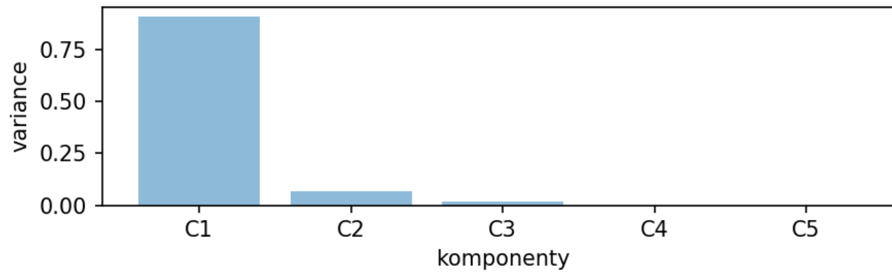


(a) Variance jednotlivých komponent po PCA.

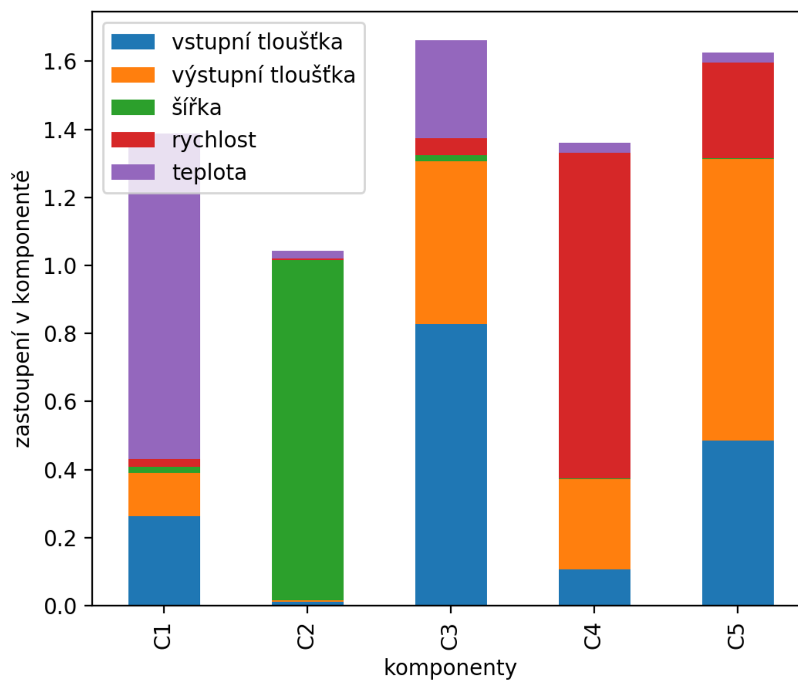


(b) Zastoupení vstupů v jednotlivých komponentách PCA.

Obrázek 7.2: PCA provedená na kombinaci vstupů 1



(a) Variance jednotlivých komponent po PCA.



(b) Zastoupení vstupů v jednotlivých komponentách PCA.

Obrázek 7.3: PCA provedená na kombinaci vstupů 2

Kapitola 8

Návrh a validace modelů

V této kapitole jsou popsány veškeré architektury, které byly navrženy a validovány pomocí programovacího jazyka *python* [11].

Jmenovitě byly použity knihovny:

1. *NumPy* - knihovna pro vědecké výpočty v jazyce Python, která mimo jiné obshauje: algoritmy pro lineární algebru, sofistikované nástroje pro integraci C a C++ kódu a modul pro generování náhodných čísel [12].

2. *TensorFlow* - knihovna pro operaci s tenzory, umělou inteligenci a strojové učení [13].

3. *Pandas* - knihovna která umožňuje manipulaci s velkým objemem dat a jejich analýzu [14].

4. *Matplotlib* - knihovna pro vizualizaci dat [15].

5. *Seaborn* - knihovna pro statistické vizualizace [16].

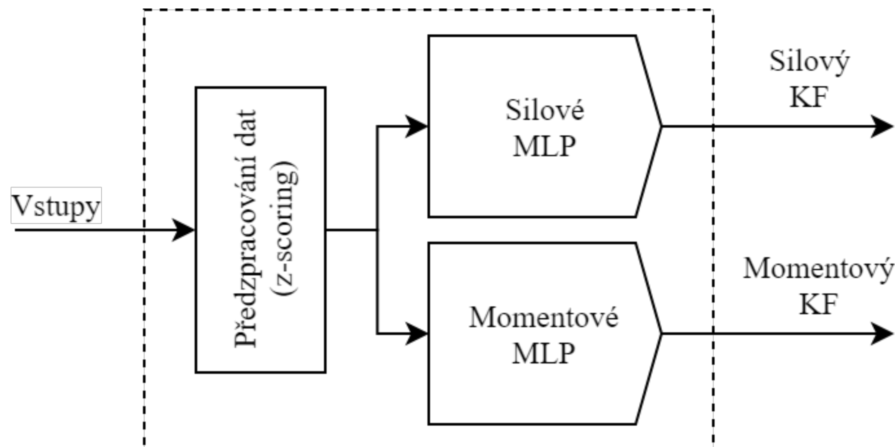
Každá architektura má označení (např. P01 - MLP), které odpovídá složce se stejným názvem v příloze *PTSW - Neural Network Research*.

Každá architektura je také ve své sekci představena a detailně zdokumentována. Následně se v subsekci validace zhodnotí výsledky případně dosažená přesnost nebo chybovost.

8.1 Soustava dvou MLP (P01 - MLP)

Řešení s označením *P01 - MLP* je velice intuitivně navržené řešení, které využívá jednoduchou architekturu MLP.

Pro každou válcovací stolici (a také materiál) je vytvořena tato soustava dvou MLP, která predikuje korekční faktory (silový a momentový).



Obrázek 8.1: Architektura řešení P01 - MLP. Vstupy jsou standardizovány, řešení obsahuje dvě MLP s totožnou architekturou. Silové MLP je trénováno na silový KF, momentové MLP je trénováno na momentový KF.

8.1.1 Architektura

Architektura tohoto řešení je velice jednoduchá. Schéma je vidět na obrázku 8.1. Skládá se ze dvou MLP - dvojčat, které mají totožnou architekturu, ale jsou učeny na silový respektive momentový korekční faktor (KF). Přičemž obě MLP z obr. 8.1 (silové a momentové MLP) mají stejné vstupy. Jde tedy o soustavu dvojčat MLP, které jsou ale trénované na jiné výstupy (silová na silový KF a momentová na momentový KF).

Co se týče architektury samotných MLP, byly použity pouze 2 vrstvy perceptronů. Vzhledem k tomu, že architektura těchto MLP je tak jednoduchá, můžeme dokonce snadno určit funkci, která tyto MLP popisuje:

$$MLP(\mathbf{x}) = \phi^{(2)}\Big(\mathbf{W^{(2)}}\phi^{(1)}\big(\mathbf{W^{(1)}}\,\mathbf{x}\big)\Big) \tag{8.1}$$

Kde jako aktivační funkce jednotlivých vrstev byly použity nejdříve aktivační funkce ReLU, ukázalo se však, že tyto aktivační funkce nejsou pro tento problém vhodné. Nakonec má každá vrstva svoji aktivační funkci určenou následovně:

$$\phi^{(1)} - \text{sigmoida}; \quad \phi^{(2)} - \tanh \tag{8.2}$$

Přičemž byly zvoleny kombinace vstupů dle doporučení expertů na válcování z firmy PTSW (tabulka 8.1).

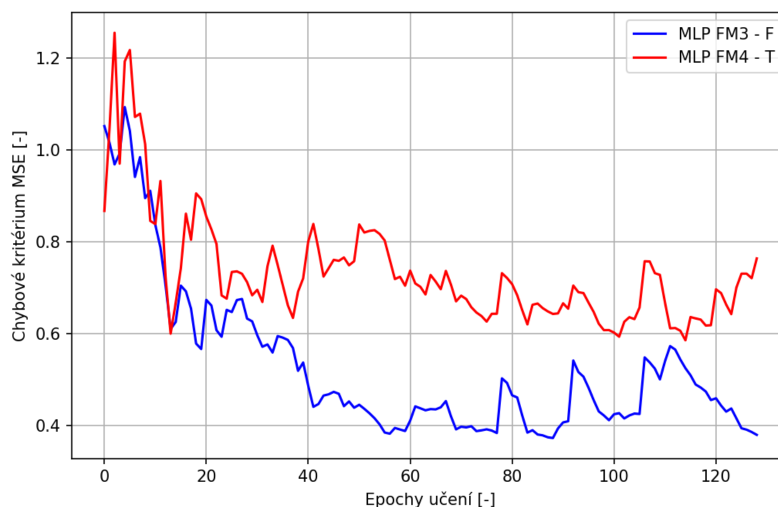
8.1.2 Proces učení

Celá architektura byla navržena pomocí knihoven *tensorflow* a *tflearn*. A byla učena algoritmem SGD (algoritmus 2). Z grafu 8.2 je patrné, že zatímco MLP FM3 F[1] dokázala snížit chybové kritérium k 0,4, u MLP FM4 T je proces učení divočejší a není naučená tak dobře.

[1]Označení MLP FM4 F znamená, že to byla MLP pro válcovací stolici FM4 pro silový korekční faktor (F).

Kombinace 1	**Kombinace 2**
Šířka	Vstupní tloušťka
Teplota	Výstupní tloušťka
Napětí	Šířka
Napěťový poměr	Rychlost
Poloha	Teplota

Tabulka 8.1: Použité kombinace vstupů pro P01 - MLP



Obrázek 8.2: Proces učení. Ilustrativně pro dvě MLP (MLP FM3 - F a MLP FM4 - T).

8.1.3 Validace

Pro validaci (ověření) řešení P01 - MLP jsem použil data, která tato soustava dvou MLP nikdy neměla k dispozici k trénovaní (přibližně 80 trénovacích dvojic typu vstup-výstup, které byly odloženy bokem před procesem trénování).

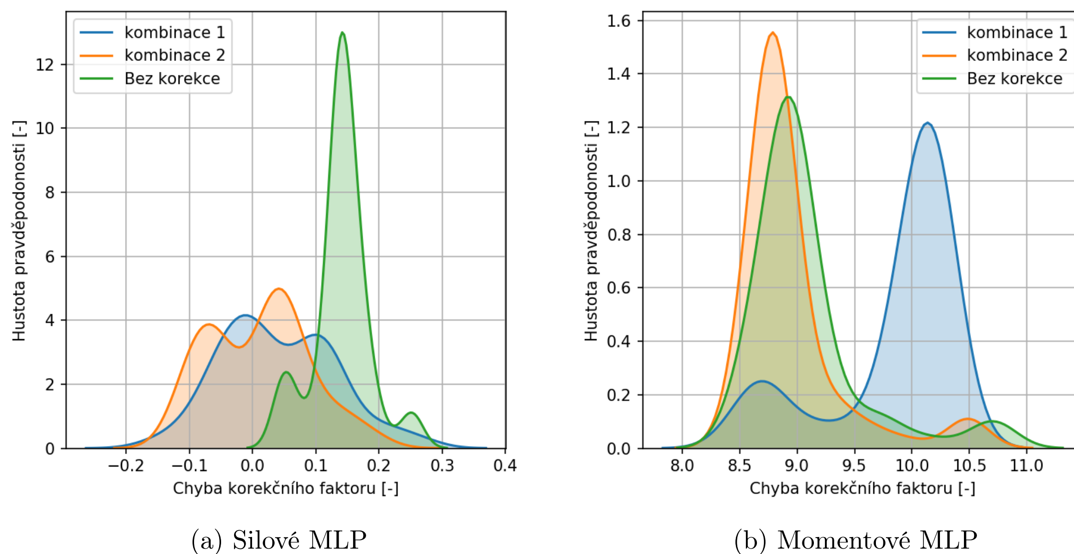
V následujících grafech je vidět porovnání *hustoty pravděpodobnosti* chyby korekčního faktoru predikovaného soustavou MLP (kombinace 1 - modrá, kombinace 2 - oranžová) pro dvě kombinace různých vstupů. Tato pravděpodobnostní hustota je konfrontována s MF modelem (Bez korekce - zelená).

Grafy byly vytvořeny metodou KDE (kernel density estimation), jako jádro byla použita gaussovská funkce.

8.1.4 Zhodnocení

Z grafů v subsekci 8.1.3 je patrné, že ne všechny MLP se dokáží naučit správně predikovat korekční faktory.

Dále je potřeba si uvědomit, že ačkoli je navrhovaná architektura MLP velmi jednoduchá, pro průmyslové využití není tak výhodná, jako například HONU.



(a) Silové MLP (b) Momentové MLP

Obrázek 8.3: Srovnání pro válcovací stolici FM1

Další velkou nevýhodou této architektury je, že každá valcovací stolice musí mít dvě dedikované MLP, což samozřejmě v praxi přináší řadu dalších problémů.

8.2 Soustava dvou HONU (P02 - HONU)

Další navržené řešení (ozmačení *P02 - HONU*) využívá architekturu HONU. Podobně jako v sekci 8.1 je pro každou stolici a také materiál vytvořit soustavu dvou HONU, které predikují korekční faktory.

8.2.1 Architektura

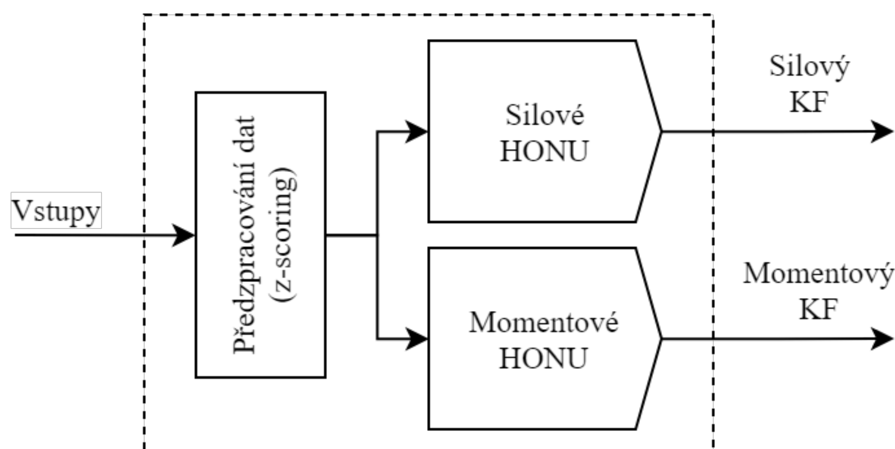
Architektura tohoto řešení vychází s návrhu P01 - MLP. MLP sítě jsou však nahrazeny HONU (obrázek 8.4).

Přičemž byly zvoleny následující kombinace vstupů (kombinace 1 a 3 jsou kombinace navrhnuté experty, kombinace 2 je pouze rozšířením kombinace 1 o předchozí korekční faktor, viz tabulka 8.2).

8.2.2 Proces učení

Architektura HONU byla navržena pouze za pomocí knihovny *NumPy* a byla učena algoritmy SBSGD (algoritmus 6) a LMA (algoritmus 5).

Z grafu 8.5 je patrné, že trénovací metoda LMA konverguje rychleji, než metoda SBSGD, která je vhodnější pro *online* trénování.



Obrázek 8.4: Architektura řešení P02 - HONU. Vstupy jsou standardizovány, řešení obsahuje dvě HONU stejného řádu. Silové HONU je trénováno na silový KF, momentové HONU je trénováno na momentový KF.

Kombinace 1	**Kombinace 2**	**Kombinace 3**
Šířka Teplota Napětí Stupeň napětí Poloha	Šířka Teplota Napětí Stupeň napětí Poloha Předchozí KF	Šířka Vstupní tloušťka Výstupní tloušťka Rychlost Teplota

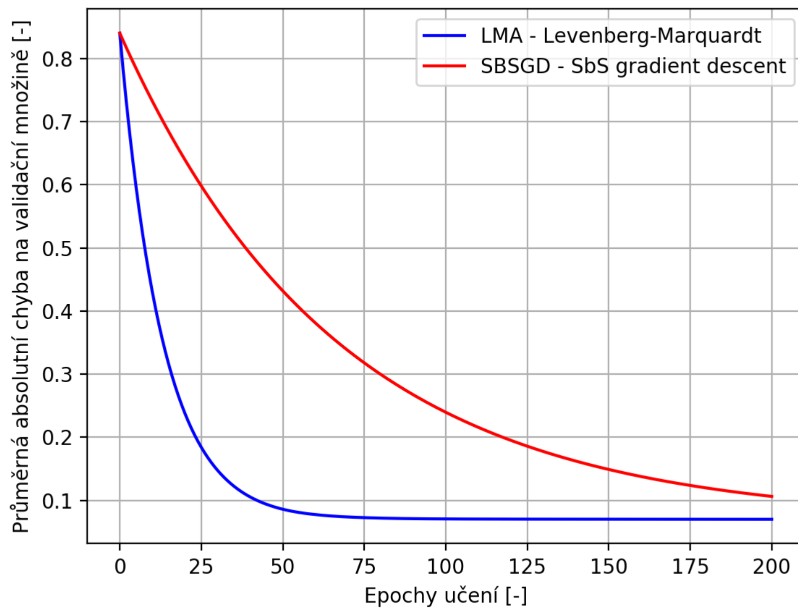
Tabulka 8.2: Použité kombinace vstupů pro P02 - HONU

Trénovací matice obsahuje 600 párů typu vstup-výstup, 200 učících epoch, $\alpha = 0.005$ - pro SGD, $\lambda = 0.1$ - pro metodu LMA, validační množina obsahuje 200 párů typu vstup-výstup.
Ačkoli byly odzkoušeny HONU řádů 1,2 a 3, nejlepší generalizační vlastnosti projevily HONU druhého řádu (tedy QNU).

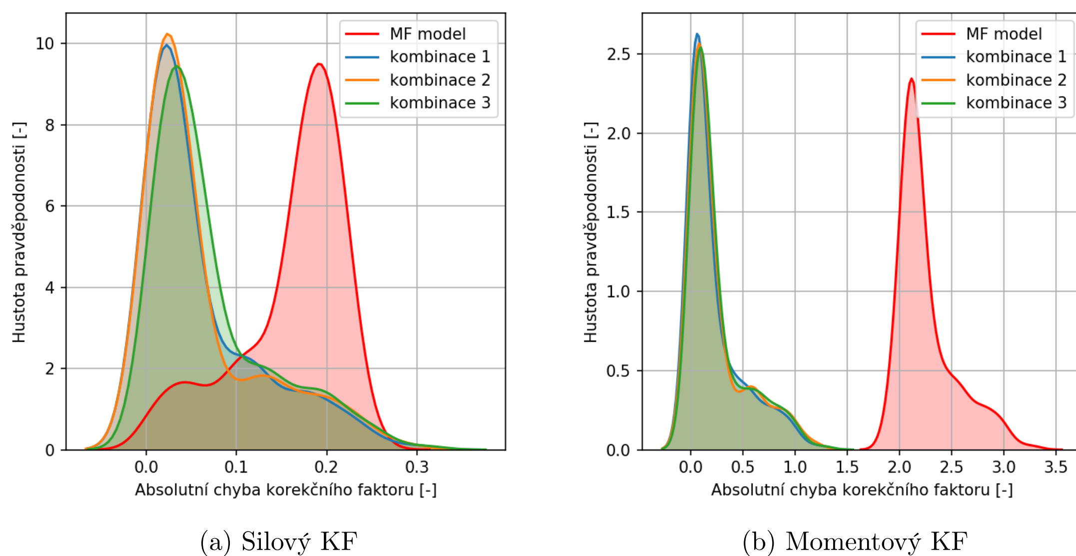
8.2.3 Validace

Pro validaci řešení P02 - HONU jsme opět použili data, která tato soustava dvou HONU (QNU) nikdy neměla k dispozici během procesu trénování (v tomto případě se jednalo přibližně o 200 párů typu vstup-výstup).

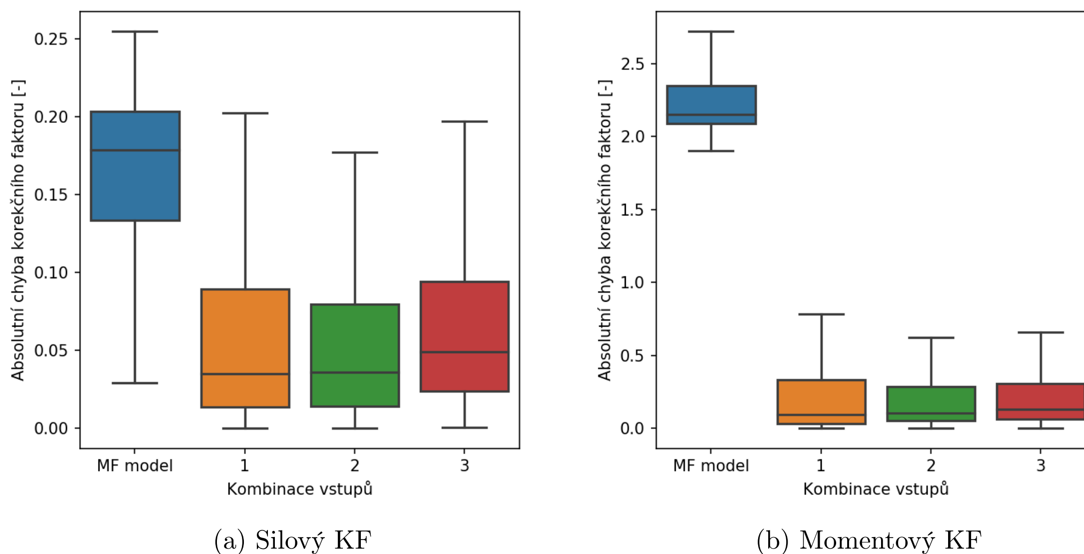
V následujících grafech je vidět porovnání hustoty pravděpodobnosti chyby korekčního faktoru predikovaného soustavou dvou QNU. Grafy byly vytvořeny metodou KDE a krabicovými grafy pomocí knihovny *seaborn*.



Obrázek 8.5: Srovnání metod SBSGD a LMA pro silové HONU válcovací stolice FM1 (Pro všechny stolice byl průběh velmi podobný). Tento graf má demonstrovat rychlost konvergence algoritmu LMA oproti SBSGD.



Obrázek 8.6: Srovnání QNU a MF modelu pro válcovací stolici RM2 pomocí KDE grafů.



(a) Silový KF (b) Momentový KF

Obrázek 8.7: Srovnání QNU a MF modelu pomocí krabicového grafu.

8.2.4 Zhodnocení

Z grafů prezentovaných v sekci 8.2.3 (KDE graf 8.6 a krabicový graf 8.7) je jasné, že tato architektura přináší zlepšení v predikci silových i momentových korekčních faktorů. Tato architektura je výhodná, protože je velmi snadno trénovatelná, předtrénovatelná a jemné ladění parametrů HONU může probíhat i za chodu (online režim učení).

Další výhodou je i fakt, že každá stolice má soustavu dvou HONU. Pokud se tedy do válcovacího procesu přidá nová stolice, stačí pouze přidat a dotrénovat jednu soustavu dvou nových HONU.

Tento fakt je však zároveň i mírnou nevýhodou, protože pro každý válcovací proces je obvykle potřeba i několik desítek HONU (každá válcovací stolice potřebuje dvě).

8.3 Hluboké MLP (P03 - Deep MLP)

V sekci 8.1 byla popsána architektura dvou MLP pro každou stolici, která má spoustu výhod (např. přidání nové stolice do válcovacího procesu znamená pouze natrénovat dvě nové MLP). Její největší nevýhodou však je, že např. pro válcovací proces se čtyřmi různými válcovacími stolicemi je potřeba celkem 8 nezávislých MLP.

V této sekci je představena architektura jedné MLP, která je učena na všechny válcovací stolice zároveň (jedna MLP je tedy pro celou válcovací linku).

8.3.1 Architektura

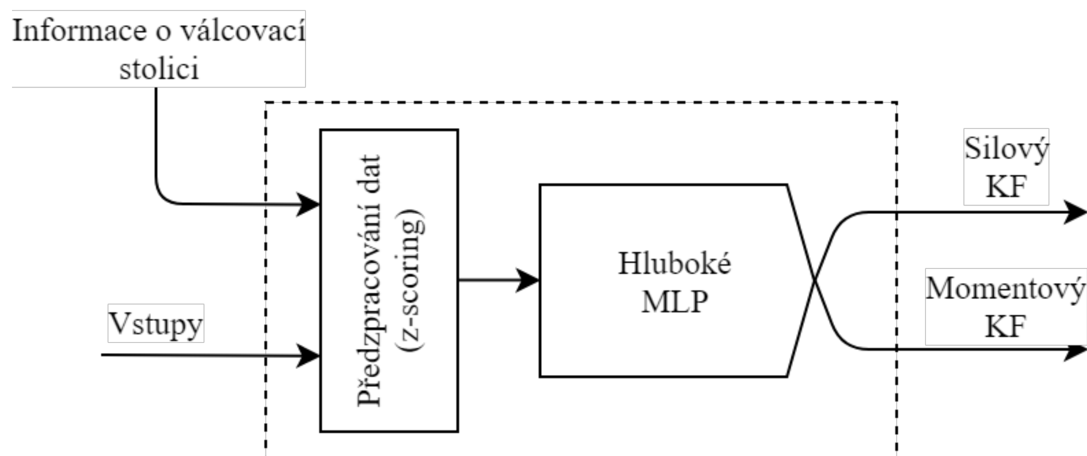
Architektura tohoto řešení je složitější, byla použita MLP s několika skrytými vrstvami. Schéma je vidět na obrázku 8.8, přičemž byly představeny dvě velmi podobné konstelace

MLP:

1. Přidání jednoho vstupu (skaláru), který válcovacím stolicím přiřadil číslo. (Hluboká MLP - skalár)

2. Přidáním celého vektoru ve formátu *one-hot*[2]. Jeden vektor reprezentuje jednu stolici. (Hluboká MLP - one-hot)

Obě konstelace však mají společné to, že nějakým způsobem obsahují informaci o stolici. Jako aktivační funkce byly použity: ReLU, sigmoida, tanh a softplus.



Obrázek 8.8: Architektura řešení P03 - Deep MLP. Vstupy jsou standardizovány. Informace o válcovací stolici je buď ve formátu *one-hot* a nebo pouze jedno číslo

8.3.2 Proces učení

Celá architektura byla navržena pomocí knihoven *tensorflow* a *tflearn* (stejně jako u architektury P01 - MLP). A byla učena algoritmem SGD (algoritmus 2). Na první pohled je patrné, že tyto MLP sítě jsou mnohem těžší na natrénování, než u architektury P01 - MLP. Epoch učení bylo potřeba v řádech stovek až tisíců.

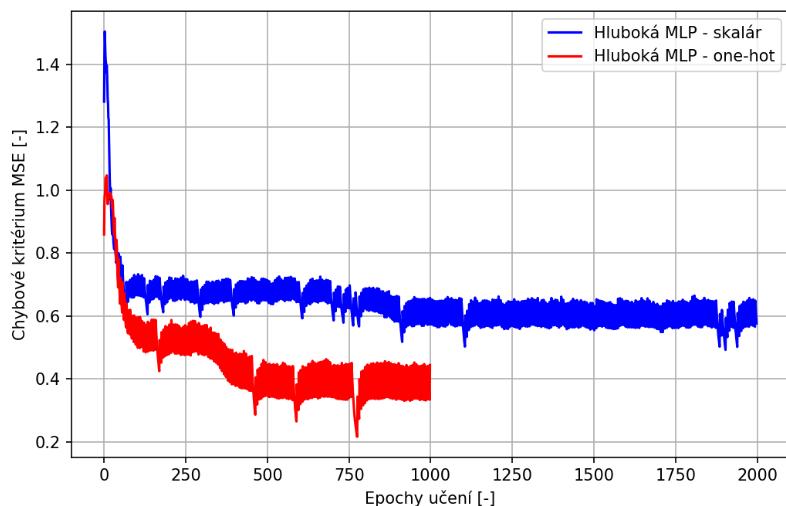
8.3.3 Validace

Pro validaci (ověření) této architektury jsem opět použil data, která MLP nikdy během procesu trénování neměla k dispozici (opět se jednalo o přibližně 80 dvojic typu vstup-výstup).

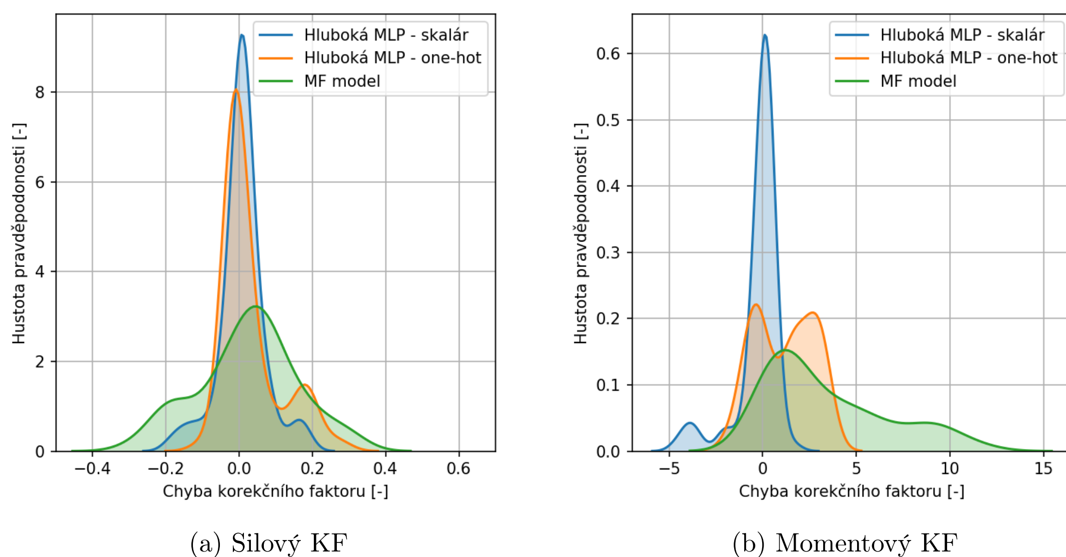
V následujících grafech je vidět porovnání hustoty pravděpodobnosti chyby korekčních faktorů pro oba typy kódování informace o stolici.

Grafy byly vytvořeny metodou KDE a krabicovými grafy pomocí knihovny *seaborn.*

[2]Formát one-hot vypadá následovně: uvažujme 2 válcovací stolice, první bude označená 1, druhá 2, pokud toto označeí převedeme do formátu one-hot, bude označení první stolice vektor $[1, 0]$ a druhé stolice $[0, 1]$.



Obrázek 8.9: Proces učení. Srovnání dvou hlubokých MLP sítí: Hluboká MLP - skalár a Hluboká MLP - one-hot. Povšiměte si, že každá MLP byla trénována různý počet epoch.



(a) Silový KF

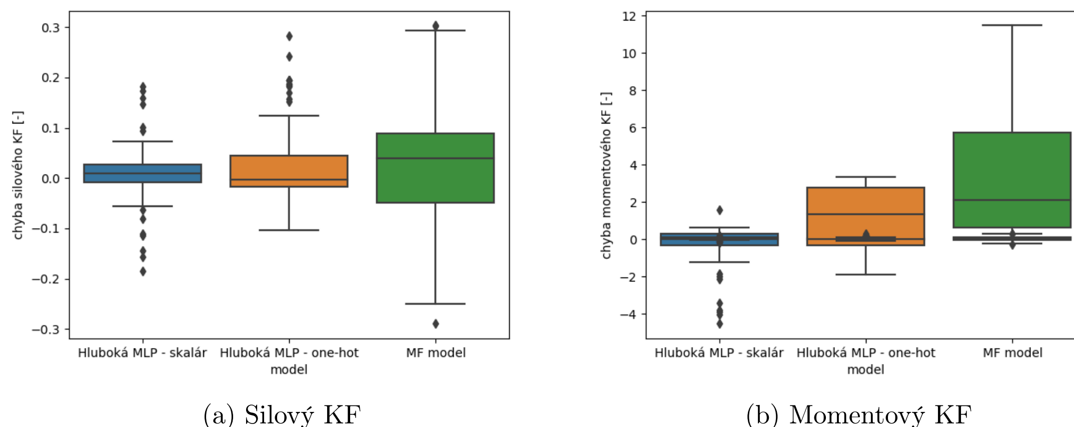
(b) Momentový KF

Obrázek 8.10: Srovnání hlubokých MLP pomocí KDE grafu.

8.3.4 Zhodnocení

Z grafů prezentovaných v sekci 8.3.3 (KDE graf 8.10 a krabicový graf 8.11) je na první pohled jasné, že navrhnutá architektura přináší značné zlepšení v predikci korekčních faktorů jak silových tak momentových.

Tato architektura je výhodná, protože jedna hluboká MLP pokryje několik válcovacích



(a) Silový KF

(b) Momentový KF

Obrázek 8.11: Srovnání hlubokých MLP a MF modelu pomocí krabicového grafu.

stolic a navíc oba dva korekční faktory (silový a momentový). V ideálním případě tak máme jednu MLP, která predikuje korekční faktory pro celou linku (samozřejmě v rámci jednoho válcovaného materiálu).

Nevýhodou je, že architektura je neprůhledná (hluboká MLP) a pokud se do výrobního procesu přidá jedna válcovací stolice, je nutné MLP přetrénovat.

8.4 Výběr architektury

V této kapitole byli ve třech sekcích představeny tři navrhované architektury numerického adaptivního algoritmu pro predikci silových a momentových korekčních faktorů:

1. **P01 - MLP** - dvě MLP pro každou válcovací stolici
   Dvouvrstvá architektura MLP přináší výhody jednoduché implementace a rychlého procesu učení. Architektura byla validováná na dvou kombinacích vstupů.
   Ačkoli tato architektura vykazovala u některých válcovacích stolic zlepšení, bohužel některé MLP se nebyly schopné závislost přesvědčivě naučit.

2. **P02 - HONU** - dvě HONU pro každou válcovací stolici
   Architektura HONU také přináší výhody relativně jednoduché implementace a rychlého učení (pro nízké řády HONU). Architektura byla validována dokonce na třech kombinacích vstupů (přičemž kombinace 2 dokonce obsahovala předchozí korekční faktor).
   HONU lze také trénovat snadno v online režimu. Nevýhodou je, že pro každou válcovací stolici a materiál je potřeba dvou dedikovaných HONU.

   Nejlépe se osvědčila HONU druhého řádu (QNU).

3. **P03 - Deep MLP** - jedna hluboká MLP pro válcovací linku
   Tato architektura nepřináší výhody jednoduché implementace (MLP síť obsahuje skryté vrstvy). Učení v online režimu je diskutabilní.

Velkou výhodou je, že je potřeba pouze jeden numerický adaptivní model pro celou válcovací linku (sadu několika válcovacích stolic). Nevýhodou je, že zatímco u předchozích dvou architektur přidání nové válcovací stolice nebyl problém (přidali se dvě nové HONU nebo MLP), u této architektury je potřeba MLP přetrénovat, případně i změnit.

Vzhledem k uvedeným výhodám a nevýhodám jednotlivých architektur a také k přihlédnutí přání firmy PTSW jsem jako nejvhodnější vybral architekturu P02 - HONU.

Kapitola 9

Implementace do jazyka C++

Po té, co byla provedena analýza dat, návrh modelu a také jeho validace, bylo potřeba vytvořit modul (hlavičkovou knihovnu) v programovacím jazyce C++, protože tento modul má sloužit ke korekci MF modelu válcování, který je už implementovaný v C++. Požadavek na jazyk C++ byl tedy součástí zadání od firmy PT Solutions Worldwide.

Vzhledem k tomu, že tento modul bude nasazen do válcovacích linek nejenom v České republice, veškerá dokumentace, názvy tříd, proměných a method jsou v anglickém jazyce.

K projektu byla použita externí knihovna *Eigen* [17]. Je to hlavičková knihovna s licencí *MPL2*[1], což znamená, že při přidání této knihovny do vlastního projektu nevznikají žádné zásadnější problémy a nejsou potřeba žádné další knihovny, na kterých by knihovna Eigen mohla záviset.

Hlavičková knihovna Eigen obsahuje mimo jiné již odladěné funkce pro maticové a vektorové výpočty (v podstatě kompletní nástroje pro lineární algebru), některé numerické řešiče a další přidružené algoritmy.

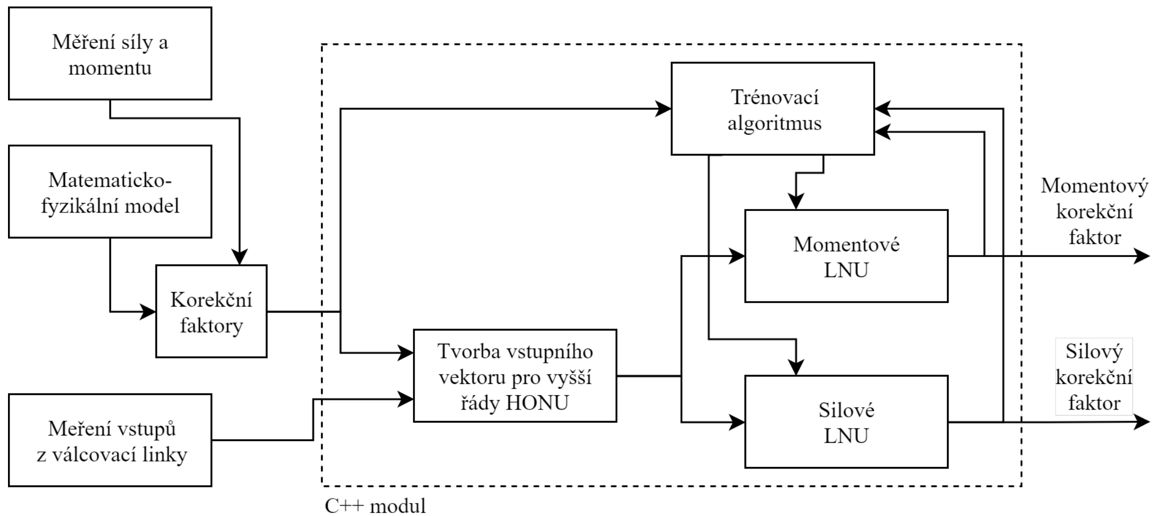
9.1 Architektura modulu v C++

Architektura navrhnutého modulu v C++ je na obrázku 9.1. Uvnitř obdelníku, jehož strany tvoří čárkovaná čára (- - -) jsou celkem 4 bloky (*Trénovací algoritmus*, *Silové LNU*, *Momentové LNU* a *Tvorba vstupního vektoru pro vyšší řády HONU*). Veškeré tyto bloky bylo potřeba naprogramovat jako třídy s potřebnými metodamy a proměnnými, přičemž dohromady tvoří odevzdanou hlavičkovou knihovnu.

Bylo tedy potřeba implementovat následující třídy:

1. **PolyUnit** (Na obr. 9.1 Momentové LNU a Silové LNU) - Polynominální jednotka, tato třída se chová jako LNU. Spolu s třídou Shaper tvoří HONU jakéhokoli řádu.
2. **Shaper** (Na obr. 9.1 Tvorba výstupního vektoru pro vyšší řády HONU) - Tato třída vytvoří z vektoru vstupů požadovaný vstupní vektor pro LNU tak, aby dohromady tvořili HONU daného řádu.

[1] Mozilla Public License Version 2.0 - licence popsaná např. zde: *http://www.mozilla.org/en-US/MPL/2.0/*



Obrázek 9.1: Schéma modulu v C++. Část označená - - - je celá řešená touto hlavičkovou knihovnou. Části mimo už byly vyřešené nebo v řešení firmou PTSW.

3. **SGD** (Na obr. 9.1 Trénovací algoritmus) - Trenér navrhnutý podle algoritmu SGD.
4. **LMA** (Na obr. 9.1 Trénovací algoritmus) - Trenér navrhnutý podle algoritmu LMA.

9.1.1 PolyUnit

Třída *PolyUnit* byla navržena podle LNU v kapitole 4.

Použiji rovnici 4.2, kterou je nutné převést do vektorové formy. Jako aktivační funkci $\phi(.)$ použijeme identitu (jinými slovy aktivační funkci neuvažujeme) a v poslední řadě budeme uvažovat *rozšířený vstupní vektor* (což nám umožní práh θ_0 včlenit do vektoru vah). Z původní rovnice:

$$HONU^1(\mathbf{x}) = LNU(\mathbf{x}) = \phi\left(\theta_0 + \sum_{i=1}^{n} \theta_i\, x_i\right) \tag{9.1}$$

tedy dostaneme jednoduchý tvar:

$$LNU(\mathbf{x}_A) = \boldsymbol{\theta}\, \mathbf{x}_A \tag{9.2}$$

Kde vektor $\mathbf{x}_A$ je rozšířený vektor[2].

Vektor $\boldsymbol{\theta}$ je to jediné, co je potřeba k definici třídy PolyUnit a spolu s délkou vektoru $\boldsymbol{\theta}$ to budou jediné třídové proměnné. Hlavičková definice třídy PolyUnit potom bude vypadat:

```
class PolyUnit {
  public:
    PolyUnit() = default;
    PolyUnit(int nw);
    void setWeights(Eigen::MatrixXd newWeights);
```

[2]Například rozšířením vektoru $\mathbf{x} = [x_1, x_2, x_3]$ dostaneme vektor $\mathbf{x}_A = [1, x_1, x_2, x_3]$.

```
    Eigen::MatrixXd getWeights(void);
    Eigen::MatrixXd value(Eigen::MatrixXd X);
    Eigen::MatrixXd value(Eigen::MatrixXd X, Eigen::MatrixXd y);
    void update(Eigen::MatrixXd dw);
  private:
    int NumWeights;
    Eigen::MatrixXd weights;
};
```

Tato třída obsahuje (kromě dvou výše popsaných privátních třídových proměnných *NumWeights* a *weights*) několik klíčových metod:

1. **PolyUnit** - konstruktor
   Tato metoda inicializuje třídu PolyUnit, přičemž sestavý počáteční vektor vah (weights) jako náhodne hodnoty z uniformního rozložení mezi 0 a 1.
   Konstruktor potřebuje jeden funkční argument počet vah (nw), konstruktor je *přetížený*, pokud není dodán argument, vytvoří výchozí prázdnou třídu.

2. **getWeights** - metoda pro manuální nastavení vah
   Tato metoda podle argumentu newWeights přenastaví vektor vah třídy PolyUnit.

3. **setWeights** - metoda pro získání vektoru vah
   Tato metoda bez argumentu vrátí vektor vah. Je vhodná, například pokud bychom chtěli archivovat informaci o stavu LNU, případně vytvořit zálohu a po špatném přetrénování vrátit váhy do původního stavu.

4. **value** - metoda výpočtu (podle 9.2)
   Tato metoda je přetížená, pokud jediný argument bude rozšířený vstupní vektor $\mathbf{x}_A$ (X), vrátí hodnotu $LNU(\mathbf{x}_A)$, pokud však kromě rozšířeného vektoru přidáme argument správné hodnoty y, dostaneme chybu $(y - LNU(\mathbf{x}_A))$, což je velice užitečné při interakci s trénovacími algoritmy.

5. **update** - metoda aktualizace vah
   Tato metoda vyžaduje jeden argument dw a provede aktualizaci vah podle obecného aktualizačního pravidla, což znamená, že k aktuálním váhám přičte hodnotu argumentu dw.

9.1.2 Shaper

Třída *Shaper* mění vektor vstupů na rozšířený vektor vstupů potřebný pro požadovaný řád HONU. Hlavičková definice této třídy potom bude:

```
class Shaper {
    public:
        Shaper() = default;
        Shaper(int ord, int nfeat);
        int getNumWeigts(void);
        Eigen::MatrixXd shapeVec(Eigen::MatrixXd inputVec);
```

```
        Eigen::Matrix<int, Eigen::Dynamic, Eigen::Dynamic> filters;
        Eigen::Matrix<int, Eigen::Dynamic, 1> filter_sums;
    private:
        int order;
        int nfeatures;
        int nweights;
};
```

Třída Shaper obsahuje tři privátní třídové proměnné:

1. **order** - řád HONU
   Hodnota 1 odpovídá LNU, hodnota 2 odpovídá QNU, ...

2. **nfeatures** - počet vstupů

3. **nweights** - počet vah pro HONU
   Hodnota této proměnné bude vypočtena podle rovnice 4.5 z kapitoly 4.

Dále třída Shaper obsahuje také dvě veřejné třídové proměnné **filters** a **filter_sums**.

Třídová proměnná **filters** je matice ve tvaru:

$$filters = \begin{bmatrix} 1 & 2 & 3 & \dots & nfeatures+1 \\ 1 & 3 & 6 & \dots & \binom{nfeatures+2}{nfeatures} \\ 1 & 4 & 10 & \dots & \binom{nfeatures+3}{nfeatures} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & order+2 & \binom{order+2}{2} & \dots & \binom{nfeatures+order}{nfeatures} \end{bmatrix} \tag{9.3}$$

A třídová Proměnná **filter_sums** je sloupcový vektor ve tvaru:

$$filter_sums = \begin{bmatrix} \sum_{i=0}^{nfeatures+1} filters_{0,i} \\ \sum_{i=0}^{nfeatures+1} filters_{1,i} \\ \sum_{i=0}^{nfeatures+1} filters_{order-1,i} \end{bmatrix} \tag{9.4}$$

Obě tyto proměnné jsou vytvořeny konstruktorem.

Třída **Shaper** obsahuje také několik třídových metod:

1. **Shaper** - konstruktor
   Tato metoda inicializuje třídu Shaper, přičemž sestaví třídové proměnné filters a filter_sums podle rovnic 9.3 respektive 9.1.2.
   Konstruktor potřebuje dva argumenty (order - řád HONU, nfeatures - délka vstupního vektoru), pokud nejsou dodány argumenty, vytvoří výchozí prázdnou třídu.

2. **getNumWeights** - metoda pro získání počtu vah
   Tato metoda vrátí počet vah potřebných pro konstrukci HONU (závisí na řádu HONU a počtu vstupů)

3. **shapeVec** - metoda pro vytvoření vstupního vektoru pro HONU
   Tato metoda pomocí předpřipravených třídových proměnných filters a filter_sums vytvoří z vstupního vektoru vektor pro HONU požadovaného řádu.
   Například pro QNU (HONU druhého řádu) z vektoru $[x_1, x_2]$
   vytvoří vektor $[1, x_1, x_1^2, x_2, x_1x_2, x_2^2]$.

Vzhledem k tomu, že se jedná opravdu o jedinečné řešení, přikládám tuto metodu jako kód:

```
Eigen::MatrixXd Shaper::shapeVec(Eigen::MatrixXd vector){
    /** \                SHAPE VECTOR
     *    ----------------------------------------------
     *    Creates input vector for HONU of desirable
     *    order (variable order) from first order input vector.
     * Arguments: order - int, order of Shaper (>0)
     *            nfeatures - int, original lenght of
     *                        input data
     * Returns:   Eigen::MatrixXd vector - input for HONU
     * Notes:     Ads bias to the first position of a vector
     * TODO:      ---
     */
    Eigen::Matrix <double, Eigen::Dynamic, 1> lnuvector;
    lnuvector.resize(this->nfeatures+1, 1);
    lnuvector(0,0) = 1;
    lnuvector.bottomRows(this->nfeatures) = vector;
    if (this->order == 1) return lnuvector;
    Eigen::Matrix<double, Eigen::Dynamic, 1> honu_vector = lnuvector;
    for (int o=0; o < this->order-1; o++) {
        Eigen::MatrixXd comb_matrix = lnuvector * honu_vector.transpose();
        honu_vector.resize(this->filter_sums[o], 1);
        int k = 0;
        for (int i = 0; i < comb_matrix.rows(); i++) {
            for (int j = 0; j < this->filters(o,i); j++) {
                honu_vector(k,0) = comb_matrix(i,j);
                k++;
            }
        }
    }
   return honu_vector;
}
```

9.1.3 SGD

Třída SGD je implementovaný algoritmus 2 z kapitoly 6.

Jeho výhodou je, že je velmi rychlý (ve srovnání s LMA) a proto je vhodný k učení v režimu online.

Třída SGD obsahuje jedinou třídní proměnnou *learning_rate*, která odpovídá parametru α (délka kroku proti směru gradientu chybového kritéria[3]). Dále obsahuje 3 třídní metody:

1. **SGD** - konstruktor
   Tato metoda inicializuje třídu SGD, požadovaný argument je parametr učení α (learning_rate).

2. **setLR** - metoda pro přenastavení parametru učení α

3. **train** - metoda pro trénování HONU
   Tato metoda podle nastavených parametrů učení (α a epochy) a podle matice vstupů a vektoru požadovaných výstupů provede aktualizaci parametrů třídy PolyUnit (proces učení).

Následující část kódu je implemetace třídové metody *train*:

```
PolyUnit SGD::train(PolyUnit pu, Eigen::MatrixXd X,
                    Eigen::MatrixXd y, int epochs/*=1*/) {
  /** \         GRADIENT DESCENT TRAINING ALG.
   *    ---------------------------------------------
   *    Computes delta weights from error and updates
   *    PolyUnit pu using gradient descent
   *    training algorithm
   * Arguments: pu - PolyUnit, Polynominal Unit to update
   *            X - MatrixXd, has to have shape (N,M)
   *                where N is number of weights of PolyUnit
   *                and M is number of vectors X
   *            y - MatrixXd, has to have shape (1,M)
   *                real values(targets)
   *            epochs - int, number of training repetition
   * Returns:   pu - PolyUnit, Trained Polynominal Unit
   * Notes:     ---
   * TO DO:     1) Normalized learning rate?
   */
  Eigen::MatrixXd error = pu.value(X.col(0),y.col(0));
  Eigen::MatrixXd dweights = X.col(0) * this->learning_rate * error;
  int columns = X.cols();

  for (int k = 0; k < epochs; k = k + 1 ) {
    for (int j = 0; j < columns; j = j+1) {
      error = pu.value(X.col(j),y.col(j));
      dweights = X.col(j) * this->learning_rate * error;
      pu.update(dweights);
    }
  }
  return pu;
}
```

[3]Z anglického *learning rate - parametr učení*

9.1.4 LMA

Třída LMA je implementovaný algoritmus 5 z kapitoly 6.

Výhodou tohoto algoritmu je, že konverguje v méně krocích, než algoritmus SGD. Bohužel se tento učící algoritmus nedá využít v režimu online učení.

Třída LMA obsahuje stejně jako SGD jedinou třídní proměnnou *learning_rate*, tato proměnná odpovídá parametru λ a dále obsahuje 3 třídní metody:

1. **LMA** - konstruktor
   Tato metoda inicializuje třídu LMA, pořadovaný je pouze parametr učení λ (learning_rate).

2. **setLR** - metoda pro přenastavení parametru učení λ

3. **train** - metoda pro trénování HONU
   Tato metoda podle nastavených parametrů učení (λ a epochy) a podle matice vstupů a vektoru požadovaných výstupů provede aktualizaci třídy PolyUnit.

Následující část kódu je implementace třídové metody train:

```
PolyUnit LMA::train(PolyUnit pu, Eigen::MatrixXd X,
                    Eigen::MatrixXd y, int epochs = 1){
  /** \        LEVENBERG-MARQUARDT TRAINING ALG.
   *    ----------------------------------------------
   *    Computes delta weights from error and updates
   *    PolyUnit pu using Levenberg-Marquardt training
   *    algorithm
   * Arguments: pu - PolyUnit, Polynominal unit to TRAIN
   *            X - MatrixXd, has to have shape (N,M)
   *                where N is number of weights of PolyUnit
   *                and M is number of vectors X
   *            y - MatrixXd, has to have shape (1,M)
   *                real values(targets)
   *            epochs - int, number of training repetition
   * Returns:   pu - PolyUnit, Trained Polynominal Unit
   * Notes:     ---
   */
  int N = X.rows();
  Eigen::MatrixXd X_T = X.transpose();
  Eigen::MatrixXd I = Eigen::MatrixXd::Identity(N,N);
  for (int k = 0; k < epochs; k = k + 1) {
    Eigen::MatrixXd Y_calc = pu.getWeights().transpose() * X;
    Eigen::MatrixXd error = y-Y_calc;
    Eigen::MatrixXd var1 = (X * X_T) * (I/this->learning_rate);
    Eigen::MatrixXd S_1 = var1.inverse();
    Eigen::MatrixXd dweights = (S_1 * X) * error.transpose();
    pu.update(dweights);
```

```
  }
  return pu;
}
```

9.2 Závěrečné poznámky k implementaci

Implementovaný modul byl samozřejmě navrhnut a naprogramován pro korekci válcovacích parametrů. Implementace však byla provedena univerzálně v tom smyslu, že je velmi jednoduché použít tento modul na jiný problém.

Do budoucna tak tento modul může být využit firmou PTSW ke konstrukci složitějších modelů, může použít vyšší řády HONU a použít oba implementované trénovací algoritmy.

Je také velice snadné vytvořit jiný trénovací algoritmus, pokud se dodrží stanovená struktura modulu.

Kapitola 10

Závěr

Cílem této práce bylo na základě analýzy dat z procesu válcování dodaných firmou PTSW navrhnout, implementovat a ověřit adaptivní numerický algoritmus pro predikci korekčních faktorů válcovacích sil (silového a momentového korekčního faktoru).

Z datové analýzy (korelační analýza a PCA, kapitola 7) byly určeny potenciální vhodné kombinace vstupů a také vhodné architektury numerických adaptivních algoritmů (kapitola 8). Jmenovitě byly vytvořeny, implementovány a otestovány tři architektury (zhodnocené v sekci 8.4):

1. **P01 - MLP** - dvojice MLP pro každou válcovací stolici
2. **P02 - HONU** - dvojice HONU pro každou válcovací stolici
3. **P03 - Deep MLP** - Hluboká MLP pro válcovací linku

Po vyhodnocení základních vlastností architektur (složitost implementace, schopnost učení v offline a online režimu) a schopností predikovat korekční faktory (kvalita aproximace) byla jako nejvýhodnější vyhodnocena architektura *P02 - HONU* - dvojice HONU pro každou válcovací stolici.

Tato architektura dosáhla dobrých aproximačních výsledků (predikce silových a momentových korekčních faktorů, obrázky 8.6 a 8.7) a prokazatelně je v kombinaci s MF modelem lepší, než MF model samotný.

Tato architektura také prokazuje schopnost dobře se učit (obrázek 8.2), přičemž byly zvoleny dva učící algoritmy:

1. **SBSGD** - Gradient descent vzorek po vzorku
   Metoda vhodná pro učení v online režimu (je však také použitelná v offline režimu), která je výpočetně velice jednoduchá.
2. **LMA** - Levenberg-Marquardtova metoda
   Metoda vhodná pro učení v offline režimu, která lépe a rychleji konverguje.

Datová analýza, návrh numerického adaptivního algoritmu a jeho validace probíhala v jazyce python. Firma PTSW požadovala implementovaný algoritmus v podobě hlavičkové knihovny

C++. Z tohoto důvodu byla vybraná architektura (P02 - HONU) implementována do jazyka C++ jako hlavičková knihovna (kapitola 9).

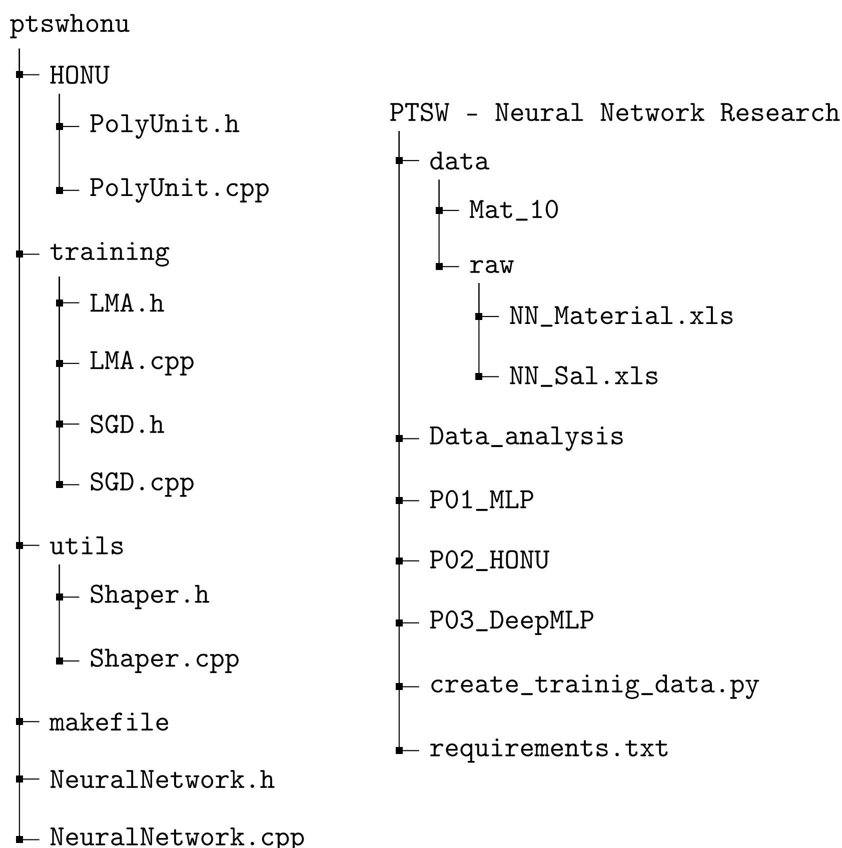
Výhodou implementace v jazyce C++ je (mimo kompatibility se softwarovými balíky firmy PTSW, což ale byla podmínka nutná) hlavně zrychlení chodu adaptivního numerického algoritmu.

Implementace do jazyka C++ byla provedena univerzálně v tom smyslu, že ačkoli bylo doporučeno použít HONU druhého řádu (QNU) s učícími algoritmy LMA a SBSGD, je velice jednoduché použít HONU v podstatě jakéhokoli řádu (s jistým omezením kvůli výpočetní složitosti popsané v sekci 4.2) a dokonce je velice jednoduché přidat nový učící algoritmus.

Hlavičková knihovna v C++ tak může firmě PTSW sloužit i v jiných oblastech než je predikce korekčních faktorů válcovacích sil.

Obsah přiloženého CD

Přiložené CD obsahuje kromě diplomové práce ve formátu PDF (*DP.pdf*) také dva adresáře:



Adresář *ptswhonu* obsahuje kompletní hlavičkovou knihovnu v jazyce C++ tak, jak je popsána v kapitole 9, včetně ukázkového příkladu použití (soubory *NeuralNetwork.h* a *NeuralNetwork.cpp*) a *makefile*.

Adresář *PTSW - Neural Network Research* obsahuje veškeré skripty v jazyce python, které se týkají jak datové analýzy (7) tak návrhu veškerých architektur popsaných v kapitole 8.

Literatura

[1] Karl Pearson. Liii. on lines and planes of closest fit to systems of points in space. *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, 2(11):559–572, 1901.

[2] Zakaria Jaadi 1692107327533655. A step by step explanation of principal component analysis, Feb 2019.

[3] Vladimír Mařík, Olga Štěpánková, and Jiří Lažanskỳ. *Umělá inteligence*, volume 1. Academia, 1993.

[4] Marvin Minsky and Seymour A Papert. *Perceptrons: An introduction to computational geometry.* MIT press, 2017.

[5] Paul J Werbos et al. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[6] Madan Gupta, Liang Jin, and Noriyasu Homma. *Static and dynamic neural networks: from fundamentals to advanced theory.* John Wiley & Sons, 2004.

[7] Ivo Bukovsky, Martin Lepold, and Jiri Bila. Quadratic neural unit and its network in validation of process data of steam turbine loop and energetic boiler. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–7. IEEE, 2010.

[8] Madan M Gupta, Ivo Bukovsky, Noriyasu Homma, Ashu MG Solo, and Zeng-Guang Hou. Fundamentals of higher order neural networks for modeling and simulation. In *Artificial Higher Order Neural Networks for Modeling and Simulation*, pages 103–133. IGI Global, 2013.

[9] Jiří Limpouch. Levenberg-marquardtova metoda, 2000. Dostupné z http://kfe.fjfi.cvut.cz/ limpouch/numet/extrem/node12.html; navštíveno 26. 3. 2019.

[10] Jorge J Moré. The levenberg-marquardt algorithm: implementation and theory. In *Numerical analysis*, pages 105–116. Springer, 1978.

[11] Programovací jazyk python. Dostupné z https://www.python.org/; navštíveno 18. 4. 2019.

[12] Numpy - knihovna pro vědecké výpočty a lineární algebru. Dostupné z http://www.numpy.org/; navštíveno 20. 4. 2019.

[13] Tensorflow - platforma pro ai a ml. Dostupné z https://www.tensorflow.org/; navštíveno 21. 4. 2019.

[14] Pandas - knihovna pro datovou analýzu. Dostupné z https://pandas.pydata.org/; navštíveno 20. 4. 2019.

[15] Matplotlib - knihovna pro vizualizace. Dostupné z https://matplotlib.org/; navštíveno 20. 3. 2019.

[16] Seaborn - knihovna pro statistické vizualizace. Dostupné z https://seaborn.pydata.org/; navštíveno 20. 4. 2019.

[17] Eigen - hlavičková knihovna c++. Dostupné z http://eigen.tuxfamily.org/; navštíveno 15. 4. 2019.