FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

Název:	Knihovna implementující vícevrstvé neuronové sítě v ABRA Gen
Student:	Bc. Tomáš Jelínek
Vedoucí:	Ing. Martin Daněk
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce zimního semestru 2021/22

Pokyny pro vypracování

Cílem práce je rozšíření ABRA frameworku o knihovnu implementující vícevrstvé neuronové sítě. Knihovna zpřístupní rozhraní pro používání vícevrstvých neuronových sítí zvolené topologie, poskytne podporu pro standardizaci a normalizaci dat a umožní perzistenci trénovaného modelu. Jednotlivé funkcionality budou otestovány pomocí jednotkových testů. Knihovna bude implementována v jazyce Delphi.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 9. června 2020

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práce

Knihovna implementující vícevrstvé neuronové sítě v ABRA Gen

Bc. Tomáš Jelínek

Katedra softwarového inženýrství
Vedoucí práce: Ing. Martin Daněk

2. února 2021

Poděkování

Děkuji vedoucímu práce Ing. Martinu Daňkovi za vedení mé práce. Poděkování patří také mým rodičům za poskytnutou podporu při studiu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principu při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisu. Dále prohlašuji, že jsem s Českým vysokým učením technickým v Praze uzavřel dohodu, na jejímž základě se ČVUT vzdalo práva na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona. Tato skutečnost nemá vliv na ustanovení § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisu.

V Praze dne 2. února 2021 . . . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií

Odkaz na tuto práci

Jelínek, Tomáš. *Knihovna implementující vícevrstvé neuronové sítě v ABRA Gen*. Diplomová práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2021.

Abstrakt

Práce se zabývá návrhem a implementací knihovny rozšiřující ABRA Gen o modul implementující vícevrstvé neuronové sítě. Navržený modul umožňuje vytvářet a konfigurovat sítě zvolených topologií. Sítím je možné přidávat vrstvy s konfigurovatelným počtem perceptronů, aktivačními funkcemi a způsoby inicializace vah. Model vzniklý inicializací navržené sítě je možné trénovat a použít pro predikci. Současně s tím je řešena otázka předzpracování dat v oblasti datové transformace, standardizace a normalizace.

Klíčová slova ABRA Gen, vícevrstvé neuronové sítě, perceptron, strojové učení, předzpracování dat, umělá inteligence, Delphi

Abstract

The thesis deals with the design and implementation of a library extending ABRA Gen by a module implementing multilayer neural networks. The designed module allows you to create and configure networks of selected topologies. It is possible to add layers with a configurable number of perceptrons, activation functions and weight initialization techniques. The model created by initializing the designed network can be trained and used for prediction. At the same time, the issue of data preprocessing in the field of data transformation, standardization and normalization is addressed.

Keywords ABRA Gen, multilayer neural networks, perceptron, machine learning, data preprocessing, artificial inteligence, Delphi

Obsah

Seznam obrázků

Úvod

Neuronové sítě zaznamenávají v posledních letech nebývalý rozvoj. Nacházejí široké pole uplatnění napříč rozličnými obory lidské činnosti. Aplikace neuronových sítí v medicíně pomáhá lékařům přesněji diagnostikovat onemocnění, zcela běžné je využití pro strojový překlad textu, rozpoznávání obličejů nebo autonomní řízení automobilů. Neuronové sítě dnes zvládnou komponovat hudbu, malovat obrazy a dokonce věrohodně napodobit hlas známých osobností. Rozvoj v této oblasti byl dříve nejčastěji v rukou technologických gigantů. Díky rozšíření poznatků o fungování neuronových sítí a dostupností frameworků umožňujících jejich použití, nachází stále více a více firem jejich uplatnění ve svém oboru.

Informační ERP systém ABRA Gen je na trhu již téměř 30 let. Za tuto dobu prošel rozsáhlým vývojem a neustálým rozšiřováním své funkcionality. Díky modulární architektuře umožňuje systém snadné řízení, plánování a automatizaci podnikových procesů. Za zmínku stojí zejména moduly CRM, BI, Nákup, Prodej, Finance a účetnictví, Mzdy a personalistika, Sklad, Výroba, Kapacitní plánování a IoT. Všechny tyto moduly produkují či uchovávají obrovské množství dat. Díky snaze o neustálé zdokonalování a zkvalitnění produktu nabízeného zákazníkům a jejich aktuálním potřebám, nastal čas na rozšíření systému o modul umožňující analýzu jejich dat pomocí umělé inteligence, konkrétně neuronových sítí.

Téma knihovna implementující vícevrstvé neuronové sítě v ABRA Gen jsem si zvolil, neboť řeší aktuální potřebu vytvořit nástroj pro tvorbu a používání neuronových sítí v systému ABRA Gen. Tento nástroj bude sloužit k analýze a získávání informací z dat. Práce by měla být přínosem pro všechny programátory a konzultanty, protože umožní jednoduše integrovat funkcionalitu neuronových sítí do jimi implementovaných řešení. Při tvorbě modulu mám tak příležitost pracovat v oboru informačních technologií, který je můj oblíbený a věnuji se mu i ve svém volném čase.

Práce se zaměřuje na návrh a implementaci modulu rozšiřujícího ABRA Gen o možnost vytvářet, trénovat a používat vícevrstvé neuronové sítě.

První kapitola představí čtenáři hlavní cíle této práce. Kapitola s názvem Analýza reprezentuje teoretickou část práce. Jsou zde popsány pro implementaci klíčové vlastnosti systému ABRA Gen, vysvětleny základní koncepty a principy fungování vícevrstvých neuronových sítí a techniky předzpracování dat. Třetí kapitola analyzuje požadavky na funkcionalitu modulu, navrhuje předpokládaný proces užití a popisuje detaily návrhu a implementace modulu. Čtvrtá kapitola je věnována testování funkcionality z předchozí kapitoly. S výsledky a shrnutím celé práce je čtenář seznámen v poslední kapitole Závěr.

Kapitola 1

Cíl práce

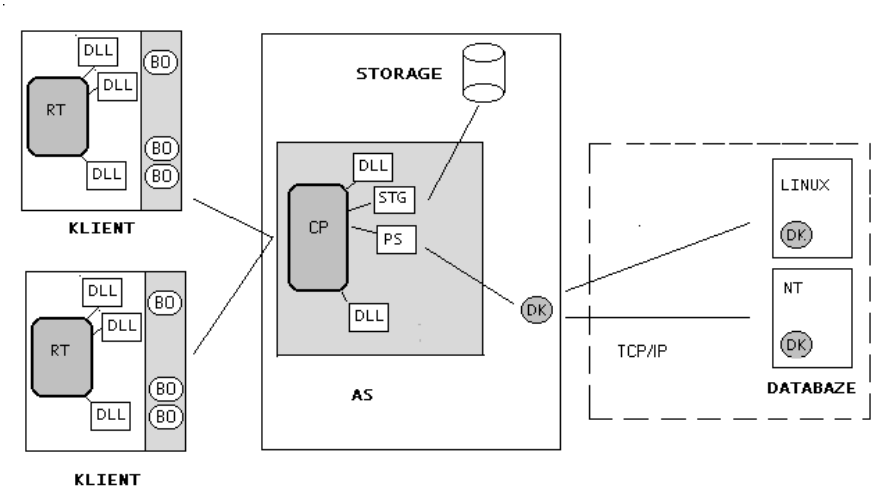
Cílem práce je návrh a implementace knihovny rozšiřující ABRA Gen o modul implementující vícevrstvé neuronové sítě. Modul bude umožňovat vytvářet a konfigurovat sítě zvolených topologií. Samozřejmostí bude možnost přidávání vrstev, volba způsobu inicializace a aktivačních funkcí. Sítě musí být možné trénovat a použít pro predikci. Modul dále umožní perzistenci vytvořených neuronových sítí a vytrénovaných vah modelu. Současně s tím bude řešena otázka předzpracování dat pro trénink a predikci, zejména v oblasti standardizace a normalizace dat. Jednotlivé funkcionality budou otestovány pomocí jednotkových případně skriptingových testů.

Kapitola 2

Analýza

2.1 ABRA Gen

Informační ERP systém ABRA Gen [1] je naprogramován ve vývojovém prostředí Delphi 10.x. Jedná se o modulární systém, který je přizpůsobitelný každému zákazníkovi přesně dle jeho potřeb. Moduly jsou dále členěny do Agend, které spolu logicky souvisí a jsou potřebné k řešení dané problematiky. Systém využívá třívrstvou architekturu typu client/server (obrázek 2.1).



Obrázek 2.1: Třívrstvá architektura (převzato z [1])

Klasická třívrstvá architektura se skládá z prezentační vrstvy, která poskytuje vstupně výstupní rozhraní aplikace uživateli, aplikační vrstvy, která zajišťuje veškeré výpočty a logiku v aplikaci, a datové vrstvy zodpovědné za správu dat. V architektuře Abra Gen jsou tyto koncepty reprezentovány následujícími vrstvami:

- **Klient**

  Klient je bežící runtime systému ABRA Gen, na který jsou navázány dynamické knihovny zajišťující rozhraní pro komunikaci s uživatelem. Oproti prezentační vrstvě se zde Klient stará i o správu business objektů, které zajišťují logiku fungování systému.

- **Aplikační server - Provider**

  Aplikační server na rozdíl od aplikační vrstvy neobsahuje byznys logiku systému. Jeho primární účel je poskytnout rozhraní (SQL dotazy) pro komunikaci s databází. Dále také zajišťuje přístup do repozitoře, kde jsou uloženy definice pro vytvoření příslušného Business objektu v paměti Klienta. Repozitoř také obsahuje dynamické a statické SQL dotazy, informace o dostupných licencích a další. Aplikační server zároveň zajišťuje provádění zámků a kontrolu licencí při práci více uživatelů.

- **Databáze**

  Tato vrstva má stejný účel jako vrstva datová v klasické architektuře, tj. zajišťuje ukládání, načítání a správu dat. Technicky je vyřešena pomocí relačního databázového serveru Firebird, MSSQL nebo Oracle.

V následující části práce jsou vysvětleny některé klíčové pojmy [2] systému ABRA Gen odkazované v kapitolách Návrh a Implementace.

- **Business objekt**

  Business objekt je nevizuální objekt reprezentující nějakou datovou strukturu a je na něj nahlíženo pomocí interface rozhraní jehož metody implementuje. Záznamy z databáze jsou mapovány přesně na konkrétní položky objektu, který zajišťuje a hlídá jejich konzistenci a validitu. Úkolem business objektu je provádění potřebných výpočtů a implementace nezbytné logiky v systému.

- **Kolekce**

  Kolekce je struktura business objektu reprezentující jeho vlastnost pomocí prvků stejného typu. Počet prvků kolekce se může dynamicky měnit a lze k nim přistupovat pomocí indexů. Hlavičkový business objekt může obsahovat kolekce řádkových business objektů, což umožňuje zajistit, aby pokud je to žádoucí, načítání a ukládání řádků probíhalo v jedné transakci s hlavičkou.

- **Číselník**

  Číselník je možné interpretovat jako tabulku, kde sloupce tabulky jsou vlastnosti a řádky jejich hodnoty. Základní charakteristikou číselníku je fakt, že každý řádek má svůj unikátní identifikátor ID. Jejich nejčastější použití je pro výběr hodnot do tzv. číselníkových položek, kde chceme vybírat hodnoty z předem připravené omezené množiny. V systému ABRA Gen je možné nad číselníky vytvářet číselníkové agendy, které umožňují uživatelům číselníky prohlížet a editovat.

- **Dataset**

  Dataset je objekt ve vizuální části systému zpřístupňující data z vícezáznamové datové struktury. Dva nejčastěji používané typy datasetů jsou objektový a memory dataset. Objektový dataset je napojen na business objekt, položky datasetu jsou přímo mapované na položky objektu nebo kalkulované. Dataset má jednotné rozhraní pro napojení vizuálních komponent např. gridu. Položky memory datasetu jsou oproti objektovému datasetu plněny přímo z SQL.

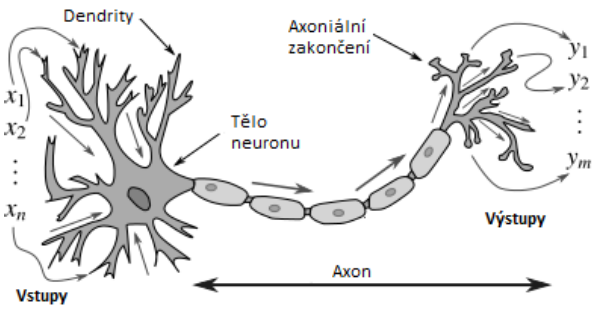
- **Skripting**

  ABRA Gen podporuje možnost skriptování. Skriptovací engine je založen na PascalScriptu, který je rozšířen tak, aby splňoval potřeby systému. Pomocí skriptingu je možné modifikovat fungování agend, business objektů a dalších funkcí systému. Místa, ve kterých lze takto změnit chování, jsou nazývána háčky. Háčky jsou metody volané v určitý okamžik, např. před uložením, uvnitř objektů, které umožní provedení příslušného kódu ze skriptingu.

Kompletní podrobné informace k fungování systému ABRA Gen je možné najít v helpu na adrese `https://help.abra.eu`.

2.2 Vícevrstvé neuronové sítě

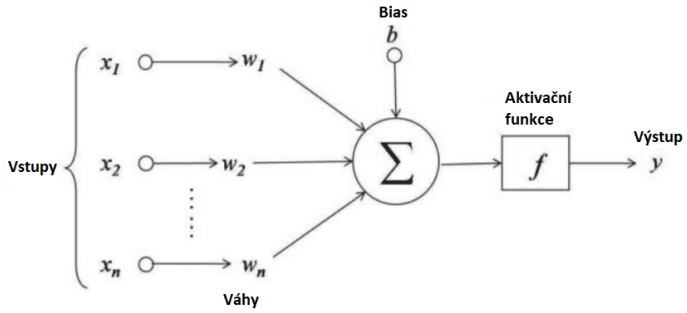
Základní myšlenky a principy fungování umělých neuronových sítí vzešly ze snahy napodobit biologické neurální systémy. Hlavní inspirací výzkumu se stal mozek skládající se z miliard vzájemně do sítě propojených buněk tzv. neuronů (obrázek 2.2). Neurony zajišťují zpracování a přenos chemických a elektrických signálů. Výběžky neuronů zvané dendrity přijímají signály z jiných neuronů a předávají je do těla neuronu ke zpracování. Zpracované signály jsou pak pomocí axonu a axoniálních zakončení předávány dalším neuronům v síti[3].



Obrázek 2.2: Biologický neuron (přeloženo z [4])

Perceptron

Umělý neuron označovaný jako perceptron (obrázek 2.3) je výpočetní model odvozený od chování biologických neuronů. Ve skutečném neuronu signály procházející podél axonů interagují multiplikativně s dendrity ostatních neuronů na základě synaptické síly v místě spojení neuronů. Dendrity pak přenášejí signály do těla buňky, kde jsou všechny sečteny. Pokud konečný součet přesáhne určitou prahovou hodnotu, potom dochází k vyslání výstupního signálu do axoniálních zakončení[3].



Obrázek 2.3: Perceptron (přeloženo z [5])

V perceptronu jsou synaptické síly reprezentovány vahami w, které jsou učitelné a řídí sílu a směr vlivu, excitační (pozitivní váha) nebo inhibiční (negativní váha), jednoho neuronu na druhý. Výpočetní model perceptronu lze popsat vztahem $y = f(\sum_{i=1}^{n} w_i x_i + b)$, který zohledňuje vliv všech vstupů x a jim odpovídajících vah w na celkový výstup y. Tento součet reprezentuje lineární hranici dělící vstupní prostor na dva podprostory. Bias b umožňuje posunutí dělicí hranice vlevo nebo vpravo od středu prostoru parametrů. Na součet je aplikována aktivační funkce f, která jej analogicky k chování biologického neuronu, transformuje na výstup[3].

Perceptron lze využít jako binární klasifikátor pro úlohy, které jsou lineárně separabilní, tj. lze je vyřešit pomocí jedné lineární hranice. Vyžaduje-li vyřešení úlohy hranic více, potom je třeba, aby každou hranici vypočítal samostatný perceptron a výstupy byly následně zkombinovány. Za tímto účelem vznikly vícevrstvé neuronové sítě, které popisuje následující sekce práce.

2.2.1 Architektura vícevrstvé sítě

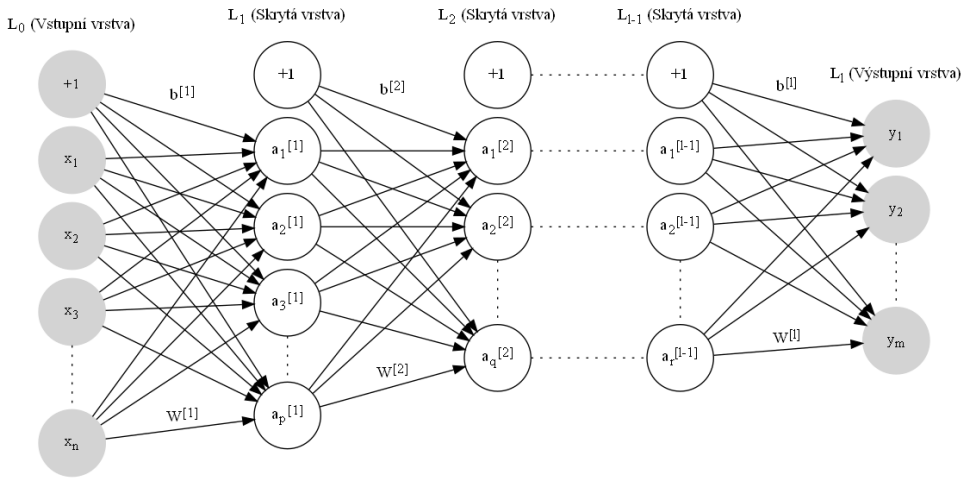
Vícevrstvý perceptron [6] - MLP (obrázek 2.4) je umělá neuronová síť složená z jednotlivých perceptronů uspořádaných do vrstev. Architektura vícevrstvé sítě je definována topologií uspořádání vrstev. Každá síť obsahuje právě jednu vstupní a výstupní vrstvu a alespoň jednu skrytou vrstvu.

Vrstvy

Vstupní vrstva slouží k načtení dat z datasetu tak, že každý perceptron odpovídá jednomu prvku vstupního vektoru. Perceptrony v této vrstvě neodpovídají přesně předchozí definici perceptronu, neboť neobsahují vektor vah, ale slouží pouze k přenosu vstupů do dalších vrstev.

Výstupní vrstva obsahuje počet perceptronů v závislosti na počtu výstupů řešené úlohy. Pro úlohy řešící regresi, tj. predikci hodnoty spojité proměnné na základě vstupů, se výstupní vrstva skládá z jediného perceptronu, který nevyužívá žádnou aktivační funkci. V případě klasifikačních problémů odpovídá počet perceptronů počtu klasifikovaných tříd a každý perceptron obvykle jako aktivační funkci aplikuje logistickou funkci. Výstup perceptronu potom udává pravděpodobnost dané třídy [6].

Mezi vstupní a výstupní vrstvou se vždy nachází alespoň jedna skrytá vrstva. Skryté vrstvy umožňují neuronové síti aproximovat libovolnou funkci více proměnných. S rostoucím počtem neuronů ve skrytých vrstvách lze dosáhnout přesnější aproximace. S rostoucím počtem neuronů však také roste doba, potřebná pro trénink takovéto sítě. V praxi je proto nutné najít kompromis mezi počtem vrstev, počtem neuronů, dobou učení a požadovanou přesností aproximace [7].



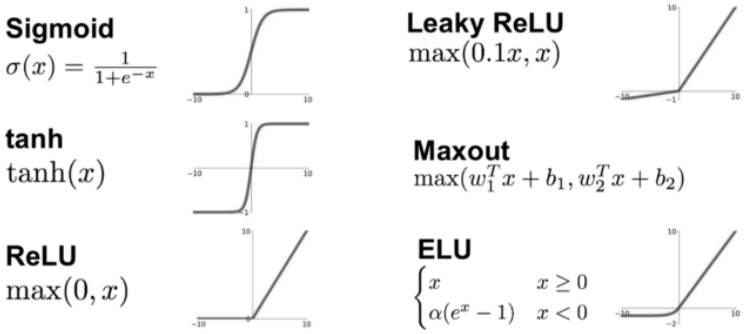
Obrázek 2.4: Neuronová síť se skrytými vrstvami

Bias

Bias je speciální perceptron viz obrázek 2.4), který je přidán do každé vrstvy, kromě vrstvy výstupní. Tento perceptron má vždy konstantní hodnotu 1. Jeho úlohou je umožnit posun grafu aktivační funkce doleva nebo doprava [8].

Aktivační funkce

Aktivační funkce v perceptronu slouží k rozhodnutí, zda má být perceptron aktivován, tzn. přispět svojí hodnotou do výpočetního modelu predikce. Aktivační funkce zásadním způsobem ovlivňují přesnost aproximace, výpočetní složitost trénovaného modelu a schopnost sítě konvergovat k řešení. Na obrázku 2.5 jsou zobrazeny grafy nejznámějších aktivačních funkcí [9].



Obrázek 2.5: Nejčastěji používané aktivační funkce (převzato z [10])

- **Sigmoid a tanh**

  Funkce Sigmoid, někdy též logistická funkce, má obor hodnot v intervalu $(0, 1)$. Díky tomu je často používána k normalizaci výstupu perceptronu a při určení pravděpodobnosti klasifikované třídy ve výstupní vrstvě. Tato funkce je díky exponenciální operaci výpočetně náročná. Možnou slabinou při použití je případ, kdy vstupní hodnota x je velmi malá nebo velká, neboť funkční hodnota funkce se v tomto případě prakticky nezmění, což může vést k pomalé konvergenci k řešení. Funkce tanh má obdobné vlastnosti, s tím rozdílem, že její obor hodnot je centrovaný v nule [9].

- **ReLU**

  Funkce ReLU je oproti předešlým funkcím výpočetně efektivnější a umožňuje rychlejší konvergenci k řešení. Pro vstupní hodnoty x menší nebo rovné nule, je gradient této funkce roven 0. To vede k problému, kdy není možné perceptrony s takovouto hodnotou dále učit pomocí zpětné propagace a celá síť degeneruje. Tento problém řeší rozšíření zvané Leaky ReLU, případně Parametric ReLU, kde je výpočet gradientu funkce $max(\alpha x, x)$ i pro záporné hodnoty nenulový a umožňuje učení pomocí zpětné propagace [9].

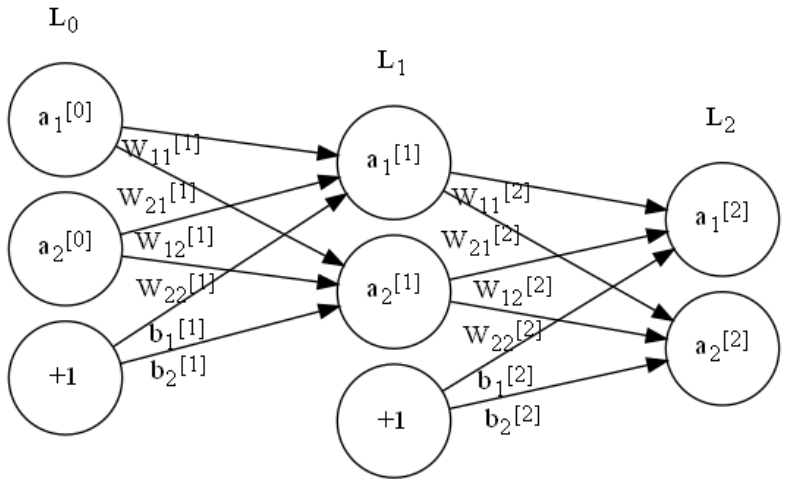
Dopředné šíření

Obrázek 2.4 zachycuje způsob propojení perceptronů v jednotlivých vrstvách. Každý perceptron je závislý na výstupech všech perceptronů z topologicky předcházející vrstvy. V rámci vrstvy neexistují žádné propojení a síť neobsahuje žádné cykly. Takováto neuronová síť se nazývá plně propojená. Dopředné šíření sítí je algoritmus popisující postupný tok dat ze vstupní do výstupní vrstvy, kde výstup perceptronů reprezentuje požadovanou predikci.

Algoritmus dopředného šíření [11] je popsán na příkladu jednoduché neuronové sítě (obrázek 2.6), která splňuje dříve popsané vlastnosti. Síť se skládá ze 3 vrstev. V prvním kroku algoritmu jsou inicializovány hodnoty perceptronů ve vstupní vrstvě L_0 hodnotami ze vstupního vektoru $\vec{x} = (x_1, x_2)$ tzn $a_1^{[0]} = x_1$, $a_2^{[0]} = x_2$. Hodnoty perceptronů v dalších vrstvách jsou vypočtený jako součet součinů všech perceptronů z předchozí vrstvy a jim odpovídajících vah. Na součet je na závěr aplikována aktivační funkce f.

$$a_1^{[1]} = f(w_{11}^{[1]} a_1^{[0]} + w_{12}^{[1]} a_2^{[0]} + b_1^{[1]}) \tag{2.1}$$
$$a_2^{[1]} = f(w_{21}^{[1]} a_1^{[0]} + w_{22}^{[1]} a_2^{[0]} + b_2^{[1]}) \tag{2.2}$$
$$a_1^{[2]} = f(w_{11}^{[2]} a_1^{[1]} + w_{12}^{[2]} a_2^{[1]} + b_1^{[2]}) \tag{2.3}$$
$$a_2^{[2]} = f(w_{21}^{[2]} a_1^{[1]} + w_{22}^{[2]} a_2^{[1]} + b_2^{[2]}) \tag{2.4}$$

Z rovnic je patrné, že dochází nejprve k výpočtu hodnot perceptronů ve vrstvě L_1 a následně ve vrstvě L_2. Algoritmus je nazván jako dopředné šíření, neboť výstupní hodnoty perceptronů jsou vypočteny na základě výstupů z předchozí vrstvy. Výpočet výstupní hodnoty perceptronu ve vrstvě l je obecně možné provést pomocí vzorce $a_i^{[l]} = f(\sum_{j=1}^{n} w_{ij}^{[l]} a_j^{[l-1]} + b_i^{[l]})$.



Obrázek 2.6: Příklad jednoduché neuronové sítě

Pro efektivní provedení výpočtu vrstvy L_1 je výhodné zapsat váhy a výstupy z vrstvy L_0 do matic tak, aby platilo:

$$A^{[1]} = f(W^{[1]} \cdot A^{[0]}) = f(\begin{bmatrix} w_{11}^{[1]} & w_{12}^{[1]} & b_1^{[1]} \\ w_{21}^{[1]} & w_{22}^{[1]} & b_2^{[1]} \end{bmatrix} \begin{bmatrix} a_1^{[0]} \\ a_2^{[0]} \\ 1^{[0]} \end{bmatrix}) \tag{2.5}$$

$$\begin{bmatrix} a_1^{[1]} \\ a_2^{[1]} \end{bmatrix} = f(\begin{bmatrix} w_{11}^{[1]} a_1^{[0]} + w_{12}^{[1]} a_2^{[0]} + b_1^{[1]} \\ w_{21}^{[1]} a_1^{[0]} + w_{22}^{[1]} a_2^{[0]} + b_2^{[1]} \end{bmatrix}) \tag{2.6}$$

Je zřejmé, že výpočet vrstvy L_1 pomocí maticového zápisu rovnice 2.6 odpovídá rovnicím 2.1 a 2.2. Za zmínku stojí umístění biasu do matice vah, díky čemuž je možné provést výpočet součtu v rámci jednoho maticového násobení. Obecně lze hodnoty ve vrstvě l získat jako $A^{[l]} = f(W^{[l]} \cdot A^{[l-1]})$, kde $W^{[l]}$ je matice vah a $A^{[l-1]}$ matice výstupů perceptronů ve vrstvě $l-1$.

2.2.2 Metody učení vícevrstvé sítě

Vícevrstvou síť můžeme označovat jako umělou inteligenci, díky její schopnosti učit se na základě vstupních dat. Učení neuronové sítě je proces postupné modifikace vah v jednotlivých vrstvách za účelem zpřesnění predikčních schopností modelu.

Inicializace vah

Před zahájením procesu učení je nutné inicializovat váhy perceptronů na výchozí hodnoty. Způsob provedení je závislý na použitých aktivačních funkcích a má významný vliv na schopnost a rychlost učení. Základní metody inicializace jsou inicializace vah nulami nebo náhodnými hodnotami.

Při inicializaci nulami je při výpočtu změny vah pomocí zpětné propagace tato změna stejná u všech perceptronů ve vrstvě a síť je tak redukována na lineární model. Jedná se o již dříve popsaný problém u aktivační funkce ReLU, který způsobuje degeneraci sítě. Alternativou je inicializace náhodnými hodnotami. Při použití aktivačních funkcí tanh a sigmoid však pro malé a velké hodnoty nastává problém s pomalou rychlostí konvergence, popsaný u aktivační funkce sigmoid.

K odstranění těchto nedostatků byly publikovány nové inicializační metody. Pro funkce ReLU lze použít metodu s názvem He. Metoda spočívá v inicializaci vah ve vrstvě l náhodnými hodnotami z normálního rozdělení se střední hodnotou 0 a směrodatnou odchylkou danou vztahem $\sigma = \sqrt{\frac{2}{\|l-1\|}}$, kde $\|l-1\|$ je počet nebiasových perceptronů vrstvy $l-1$. Obdobně lze použít Xavierovu inicializaci pro funkce sigmoid a tanh. Rozdíl je ve výpočtu směrodatné odchylky $\sigma = \sqrt{\frac{2}{\|l-1\|+\|l\|}}$, kde je odchylka závislá na počtu nebiasových perceptronů ve vrstvě $l-1$ i l.

Hodnoty biasů jsou zpravidla inicializovány nulami, neboť před samotným začátkem učení není žádoucí posun ze středu souřadného systému [12].

Učení s učitelem

Učení s učitelem je jednou z často používaných metod ve strojovém učení. Základním konceptem je rozdělení vstupního datasetu na trénovací a validační část. Data musí ke každému vstupu obsahovat i očekávaný výstup. Trénovací dataset slouží k postupnému vytrénování příslušného modelu tak, aby na vstup odpovídal očekávaným výstupem. Validační část datasetu umožňuje odhadnout jaká je úspěšnost modelu při predikci na neznámých datech.

Algoritmem na bázi učení s učitelem ve vícevrstvé neuronové síti je Zpětná propagace [13]. Algoritmus opakovaně upravuje váhy v jednotlivých vrstvách neuronové sítě tak, aby minimalizoval odchylku mezi výstupními hodnotami sítě a očekávanými výstupy. Pro výpočet celkové chyby predikce sítě lze použít vztah pro střední kvadratickou chybu $E = \frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$, kde $\hat{y}$ je očekávaný výstup, y predikce sítě a n počet perceptronů ve výstupní vrstvě. Minimalizaci funkce E vzhledem k vahám $w^{[l]}$ ve vrstvě l lze interpretovat jako výpočet gradientu $\nabla E_{w^{[l]}} = (\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_m})$. Gradient nám určuje jakým způsobem je třeba jednotlivé váhy změnit, aby chyba predikce sítě byla minimální. Nové hodnoty vah jsou potom dány vztahem $w_i = w_i - \epsilon\frac{\partial E}{\partial w_i}$. Hyperparametr ϵ tzv. rychlost učení ovlivňuje jakou rychlostí se model přizpůsobí řešenému problému. Obvykle nabývá hodnot z intervalu $(0, 1)$. Menší rychlost učení vyžaduje více tréninkových cyklů vzhledem k menším změnám vah při každé aktualizaci. Oproti tomu vyšší rychlost učení vede k větším změnám vah a vyžaduje méně tréninkových cyklů. Příliš velká rychlost učení může způsobit konvergenci modelu k neoptimálnímu řešení, zatímco příliš malá rychlost učení může způsobit neschopnost sítě konvergovat k řešení. V praxi je proto nutné pro každý řešený problém najít optimální hodnotu hyperparametru ϵ. Následující pseudokód zobrazuje celkový pohled na proces učení:

```
Vstup: Trénovací dataset

Inicializace vah
Do
  ForEach položku z~datasetu
    Výpočet výstupu pomocí dopředného šíření
    Úprava váh pomocí zpětné propagace
  EndFor
Until chyba sítě < požadovaná nebo překročen počet cyklů
```

Výpočet úpravy vah pomocí zpětné propagace je podrobněji vysvětlen na již výše popsaném příkladu neuronové sítě z obrázku 2.6. Pomocí dopředného šíření byly vypočteny výstupní hodnoty jednotlivých perceptronů. Rovnice popisující výpočet těchto hodnot lze vyjádřit ve tvaru vnější funkce f a vnitřní

funkce z. V našem příkladě je jako aktivační funkce f použita funkce sigmoid.

$$z_1^{[1]} = w_{11}^{[1]} a_1^{[0]} + w_{12}^{[1]} a_2^{[0]} + b_1^{[1]} \tag{2.7}$$
$$z_2^{[1]} = w_{21}^{[1]} a_1^{[0]} + w_{22}^{[1]} a_2^{[0]} + b_2^{[1]} \tag{2.8}$$
$$z_1^{[2]} = w_{11}^{[2]} a_1^{[1]} + w_{12}^{[2]} a_2^{[1]} + b_1^{[2]} \tag{2.9}$$
$$z_2^{[2]} = w_{21}^{[2]} a_1^{[1]} + w_{22}^{[2]} a_2^{[1]} + b_2^{[2]} \tag{2.10}$$

$$f_{a_1^{[1]}} = \frac{1}{1 + e^{-z_1^{[1]}}} \tag{2.11}$$
$$f_{a_2^{[1]}} = \frac{1}{1 + e^{-z_2^{[1]}}} \tag{2.12}$$
$$f_{a_1^{[2]}} = \frac{1}{1 + e^{-z_1^{[2]}}} \tag{2.13}$$
$$f_{a_2^{[2]}} = \frac{1}{1 + e^{-z_2^{[2]}}} \tag{2.14}$$

Celková chyba predikce se vypočte jako:

$$E = E_{a_1^{[2]}} + E_{a_2^{[2]}} = \frac{1}{2}(\hat{y}_{a_1^{[2]}} - f_{a_1^{[2]}})^2 + \frac{1}{2}(\hat{y}_{a_2^{[2]}} - f_{a_2^{[2]}})^2 \tag{2.15}$$

Cílem následujících výpočtů je určit jakým způsobem upravit jednotlivé váhy v síti, aby byla celková chyba co nejmenší. Nové hodnoty vah jsou vypočítávány postupně od výstupní vrstvy směrem k vrstvě vstupní. Nejprve je třeba spočítat jakým způsobem ovlivní váha $w_{11}^{[2]}$ celkovou chybu E. To lze zapsat jako parciální derivaci $\frac{\partial E}{\partial w_{11}^{[2]}}$. Tato parciální derivace nelze vypočítat přímo, ale je na ni třeba aplikovat řetězové pravidlo [14]. Pro výpočet parciální derivace pak platí následující vztah:

$$\frac{\partial E}{\partial w_{11}^{[2]}} = \frac{\partial E}{\partial f_{a_1^{[2]}}} \frac{\partial f_{a_1^{[2]}}}{\partial z_1^{[2]}} \frac{\partial z_1^{[2]}}{\partial w_{11}^{[2]}} \tag{2.16}$$

Nyní lze provést výpočet dílčích parciálních derivací:

$$\frac{\partial E}{\partial w_{11}^{[2]}} = (f_{a_1^{[2]}} - \hat{y}_{a_1^{[2]}})(f_{a_1^{[2]}}(1 - f_{a_1^{[2]}}))(f_{a_1^{[1]}}) \tag{2.17}$$

Nová hodnota váhy $w_{11}^{[2]}$ je vypočtena jako $w_{11}^{[2]} = w_{11}^{[2]} - \epsilon \frac{\partial E}{\partial w_{11}^{[2]}}$. Stejný postup je aplikován pro výpočet nových hodnot vah ve výstupní vrstvě a lze zapsat rovnicemi:

$$\frac{\partial E}{\partial w_{12}^{[2]}} = \frac{\partial E}{\partial f_{a_1^{[2]}}} \frac{\partial f_{a_1^{[2]}}}{\partial z_1^{[2]}} \frac{\partial z_1^{[2]}}{\partial w_{12}^{[2]}} = (f_{a_1^{[2]}} - \hat{y}_{a_1^{[2]}})(f_{a_1^{[2]}}(1 - f_{a_1^{[2]}}))(f_{a_2^{[1]}}) \tag{2.18}$$

$$\frac{\partial E}{\partial w_{21}^{[2]}} = \frac{\partial E}{\partial f_{a_2^{[2]}}} \frac{\partial f_{a_2^{[2]}}}{\partial z_2^{[2]}} \frac{\partial z_2^{[2]}}{\partial w_{21}^{[2]}} = (f_{a_2^{[2]}} - \hat{y}_{a_2^{[2]}})(f_{a_2^{[2]}}(1 - f_{a_2^{[2]}}))(f_{a_1^{[1]}}) \tag{2.19}$$

$$\frac{\partial E}{\partial w_{22}^{[2]}} = \frac{\partial E}{\partial f_{a_2^{[2]}}} \frac{\partial f_{a_2^{[2]}}}{\partial z_2^{[2]}} \frac{\partial z_2^{[2]}}{\partial w_{22}^{[2]}} = (f_{a_2^{[2]}} - \hat{y}_{a_2^{[2]}})(f_{a_2^{[2]}}(1 - f_{a_2^{[2]}}))(f_{a_2^{[1]}}) \tag{2.20}$$

Úprava vah ve skryté vrstvě je založena na stejném principu. Pro váhu $w_{11}^{[1]}$ se výpočet parciální derivace provede dle vztahu:

$$\frac{\partial E}{\partial w_{11}^{[1]}} = \frac{\partial E}{\partial f_{a_1^{[1]}}} \frac{\partial f_{a_1^{[1]}}}{\partial z_1^{[1]}} \frac{\partial z_1^{[1]}}{\partial w_{11}^{[1]}} \tag{2.21}$$

Oproti předchozímu případu výstup perceptronu ve skryté vrstvě přispívá do výstupu perceptronů ve výstupní vrstvě a tím i do celkové chyby sítě. V našem případě tedy perceptron $a_1^{[1]}$ přispívá do obou výstupních perceptronů $a_1^{[2]}$ a $a_2^{[2]}$. Parciální derivace $\frac{\partial E}{\partial f_{a_1^{[1]}}}$ musí zohlednit příspěvek následujícím způsobem:

$$\frac{\partial E}{\partial f_{a_1^{[1]}}} = \frac{\partial E_{a_1^{[2]}}}{\partial f_{a_1^{[1]}}} + \frac{\partial E_{a_2^{[2]}}}{\partial f_{a_1^{[1]}}} \tag{2.22}$$

$$\frac{\partial E}{\partial f_{a_1^{[1]}}} = \frac{E_{a_1^{[2]}}}{\partial f_{a_1^{[2]}}} \frac{\partial f_{a_1^{[2]}}}{\partial z_1^{[2]}} \frac{\partial z_1^{[2]}}{\partial f_{a_1^{[1]}}} + \frac{E_{a_2^{[2]}}}{\partial f_{a_2^{[2]}}} \frac{\partial f_{a_2^{[2]}}}{\partial z_2^{[2]}} \frac{\partial z_2^{[2]}}{\partial f_{a_1^{[1]}}} \tag{2.23}$$

Výsledný vztah pro výpočet změny váhy $w_{11}^{[1]}$ je dán kombinací rovnic 2.21 a 2.23, kde je již možné spočítat jednotlivé dílčí parciální derivace:

$$\frac{\partial E}{\partial w_{11}^{[1]}} = \left(\frac{E_{a_1^{[2]}}}{\partial f_{a_1^{[2]}}} \frac{\partial f_{a_1^{[2]}}}{\partial z_1^{[2]}} \frac{\partial z_1^{[2]}}{\partial f_{a_1^{[1]}}} + \frac{E_{a_2^{[2]}}}{\partial f_{a_2^{[2]}}} \frac{\partial f_{a_2^{[2]}}}{\partial z_2^{[2]}} \frac{\partial z_2^{[2]}}{\partial f_{a_1^{[1]}}} \right) \frac{\partial f_{a_1^{[1]}}}{\partial z_1^{[1]}} \frac{\partial z_1^{[1]}}{\partial w_{11}^{[1]}} =$$
$$= [(f_{a_1^{[2]}} - \hat{y}_{a_1^{[2]}})(f_{a_1^{[2]}}(1 - f_{a_1^{[2]}}))(w_{11}^{[2]}) + (f_{a_2^{[2]}} - \hat{y}_{a_2^{[2]}})(f_{a_2^{[2]}}(1 - f_{a_2^{[2]}}))(w_{21}^{[2]})]$$
$$(f_{a_1^{[1]}}(1 - f_{a_1^{[1]}}))(f_{a_1^{[0]}}) \tag{2.24}$$

Pro zbývající váhy ve skryté vrstvě je výpočet proveden analogicky dle rovnic:

$$\frac{\partial E}{\partial w_{12}^{[1]}} = \left(\frac{E_{a_1^{[2]}}}{\partial f_{a_1^{[2]}}} \frac{\partial f_{a_1^{[2]}}}{\partial z_1^{[2]}} \frac{\partial z_1^{[2]}}{\partial f_{a_1^{[1]}}} + \frac{E_{a_2^{[2]}}}{\partial f_{a_2^{[2]}}} \frac{\partial f_{a_2^{[2]}}}{\partial z_2^{[2]}} \frac{\partial z_2^{[2]}}{\partial f_{a_1^{[1]}}} \right) \frac{\partial f_{a_1^{[1]}}}{\partial z_1^{[1]}} \frac{\partial z_1^{[1]}}{\partial w_{12}^{[1]}} =$$
$$= [(f_{a_1^{[2]}} - \hat{y}_{a_1^{[2]}})(f_{a_1^{[2]}}(1 - f_{a_1^{[2]}}))(w_{11}^{[2]}) + (f_{a_2^{[2]}} - \hat{y}_{a_2^{[2]}})(f_{a_2^{[2]}}(1 - f_{a_2^{[2]}}))(w_{21}^{[2]})]$$
$$(f_{a_1^{[1]}}(1 - f_{a_1^{[1]}}))(f_{a_2^{[0]}}) \quad (2.25)$$

$$\frac{\partial E}{\partial w_{21}^{[1]}} = \left(\frac{E_{a_1^{[2]}}}{\partial f_{a_1^{[2]}}} \frac{\partial f_{a_1^{[2]}}}{\partial z_1^{[2]}} \frac{\partial z_1^{[2]}}{\partial f_{a_2^{[1]}}} + \frac{E_{a_2^{[2]}}}{\partial f_{a_2^{[2]}}} \frac{\partial f_{a_2^{[2]}}}{\partial z_2^{[2]}} \frac{\partial z_2^{[2]}}{\partial f_{a_2^{[1]}}} \right) \frac{\partial f_{a_2^{[1]}}}{\partial z_2^{[1]}} \frac{\partial z_2^{[1]}}{\partial w_{21}^{[1]}} =$$
$$= [(f_{a_1^{[2]}} - \hat{y}_{a_1^{[2]}})(f_{a_1^{[2]}}(1 - f_{a_1^{[2]}}))(w_{12}^{[2]}) + (f_{a_2^{[2]}} - \hat{y}_{a_2^{[2]}})(f_{a_2^{[2]}}(1 - f_{a_2^{[2]}}))(w_{22}^{[2]})]$$
$$(f_{a_2^{[1]}}(1 - f_{a_2^{[1]}}))(f_{a_1^{[0]}}) \quad (2.26)$$

$$\frac{\partial E}{\partial w_{22}^{[1]}} = \left(\frac{E_{a_1^{[2]}}}{\partial f_{a_1^{[2]}}} \frac{\partial f_{a_1^{[2]}}}{\partial z_1^{[2]}} \frac{\partial z_1^{[2]}}{\partial f_{a_2^{[1]}}} + \frac{E_{a_2^{[2]}}}{\partial f_{a_2^{[2]}}} \frac{\partial f_{a_2^{[2]}}}{\partial z_2^{[2]}} \frac{\partial z_2^{[2]}}{\partial f_{a_2^{[1]}}} \right) \frac{\partial f_{a_2^{[1]}}}{\partial z_2^{[1]}} \frac{\partial z_2^{[1]}}{\partial w_{22}^{[1]}} =$$
$$= [(f_{a_1^{[2]}} - \hat{y}_{a_1^{[2]}})(f_{a_1^{[2]}}(1 - f_{a_1^{[2]}}))(w_{12}^{[2]}) + (f_{a_2^{[2]}} - \hat{y}_{a_2^{[2]}})(f_{a_2^{[2]}}(1 - f_{a_2^{[2]}}))(w_{22}^{[2]})]$$
$$(f_{a_2^{[1]}}(1 - f_{a_2^{[1]}}))(f_{a_2^{[0]}}) \quad (2.27)$$

Stejně jako u dopředného šíření je výpočetně efektivnější provádět výpočet gradientu $\nabla E_{W^{[l]}}$ pomocí maticových operací [15]. Nechť matice $A^{[l]}$ je matice výstupů perceptronů, $f^{[l]}$ je aktivační funkce, $B^{[l]}$ je matice biasů a $W^{[l]}$ je matice vah ve vrstvě l, potom pro výpočet gradientu platí následující čtyři axiomy:

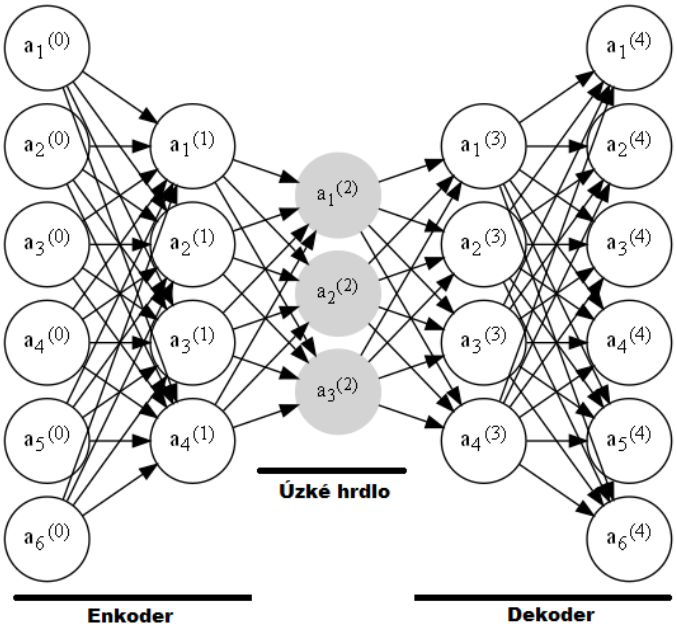
$$\delta^{[L]} = (f^{[L]})' \circ \nabla E_{A^{[L]}} \quad (2.28)$$
$$\delta^{[l]} = (f^{[l]})' \circ ((W^{[l+1]})^T \delta^{[l+1]}) \quad (2.29)$$
$$\nabla E_{W^{[l]}} = \delta^{[l]} (A^{[l-1]})^T \quad (2.30)$$
$$\nabla E_{B^{[l]}} = \delta^{[l]} \quad (2.31)$$

Axiomy vedou na použití tzv. pravidla delta, pomocí kterého lze při výpočtu gradientu rekurzivně využít již vypočtenou deltu z předchozí vrstvy. Rovnice 2.28 popisuje výpočet delty $\delta^{[L]}$ ve výstupní vrstvě L. Gradient $\nabla E_{A^{[L]}}$ lze vyjádřit jako $\nabla E_{A^{[L]}} = A^{[L]} - \hat{Y}$, kde první je matice výstupů perceptronů výstupní vrstvy a druhá je matice očekávaných výstupů. Pro všechny ostatní vrstvy je příslušná delta vypočtena rekurzivně na základě delty z předchozí vrstvy viz rovnice 2.29. Za použití takto spočtených delt lze již jednoduše dopočítat hledané gradienty $E_{W^{[l]}}$ a $E_{B^{[l]}}$ jak popisují rovnice 2.30 a 2.31. Nové hodnoty vah jsou tedy postupně dopočítávány od výstupní vrstvy směrem k vrstvě vstupní.

Učení bez učitele

Učení bez učitele je koncept strojového učení nad vstupním datasetem, ke kterému oproti učení s učitelem nejsou dopředu známé a člověkem definované výstupy. Cílem takovéhoto učení je rozdělit záznamy v datasetu do skupin dle podobnosti vzdáleností umístění ve stavovém prostoru atributů.

Architektura vícevrstvé neuronové sítě vhodná pro učení bez učitele je nazývána autoenkoder [16]. Autoenkoder využívá algoritmus učení pomocí zpětné propagace na datasetu bez očekávaných výstupů. Místo očekávaných výstupů jsou pak při výpočtu celkové chyby predikce použity hodnoty ze vstupu. Autoenkoder je tedy jednoduše vícevrstvá neuronová síť, která se snaží replikovat hodnoty ze vstupní vrstvy do vrstvy výstupní. Na obrázku (2.7) je zobrazen příklad této architektury.

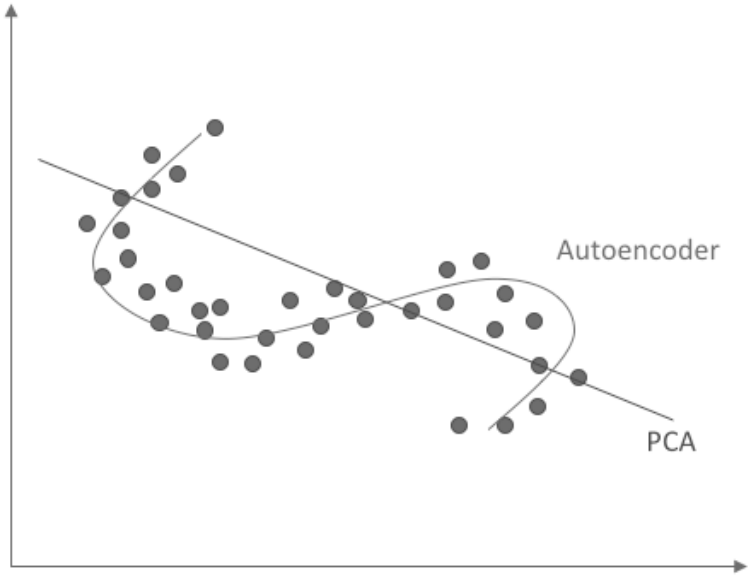


Obrázek 2.7: Architektura autoenkoderu

Autoenkoder je schematicky rozdělen na tři základní části enkoder, dekoder a úzké hrdlo. V části enkoderu probíhá komprese vstupních dat, zatímco dekoder se snaží komprimovaná data rekonstruovat do původní podoby. Enkoder a dekoder mají až na zrcadlové převrácení obvykle stejnou topologii, tj stejný počet vrstev, perceptronů, stejné aktivační funkce atd. Klíčovým místem v takovéto neuronové síti je úzké hrdlo. Úzké hrdlo omezuje množství informací, které mohou procházet celou sítí, a tím vynucuje naučenou kompresi vstupních

dat. Bez přítomnosti úzkého hrdla by si síť jednoduše zapamatovala vstupy a pouze je nezměněné přeposlala na výstup.

Díky schopnosti neuronových sítí aproximovat nelineární funkce je autoencoder ideální alternativou a zobecnění analýzy hlavních komponent (PCA). PCA se snaží najít dimenzionální redukci prostoru atributů, která by co nejvíce odpovídala původním datům. Autoencodery jsou schopny provést tuto dimenzionální redukci nelineárním způsobem jak je vidět na obrázku (2.8), neboť jsou schopné se naučit komplexní reprezentaci dat, kterou lze použít k popisu pozorování v nižší dimenzi a odpovídajícím způsobem ji dekódovat do původního vstupního prostoru.



Obrázek 2.8: Lineární vs nelineární dimenzionální redukce (převzato z [16])

Autoenkoder je architektura neuronové sítě schopná objevit závislosti v datech za účelem získání komprimované reprezentace vstupu. Existuje mnoho různých variant specializovaných architektur autoenkoderů s cílem zajistit, aby komprimovaná reprezentace představovala smysluplné atributy původního vstupu dat. Obvykle je autoenkoder schopný rekonstruovat pouze data podobná třídám pozorování, které model poznal během tréninku.

Posilované učení

Posilované učení je metoda učení umělé inteligence, jejímž cílem je vytrénovat model schopný uspokojivě provést posloupnost rozhodnutí. Trénovaný model je umístěn do potenciálně složitého prostředí, kde se pomocí jemu definovaných akcí snaží dosáhnout stanoveného cíle. Po provedení každé akce je model odměněn nebo penalizován. Jeho cílem je maximalizovat celkovou odměnu. Přestože programátor nastavuje zásady odměňování tj. pravidla prostředí, nedává modelu žádné rady ani návrhy, jak se v prostředí správně chovat. Trénink tak obvykle začíná zcela náhodnými pokusy a končí sofistikovanou taktikou a nadlidskými dovednostmi. Posilované učení je v současnosti jedna z nejefektivnějších metod jak naučit stroj řešit složité rozhodovací problémy, neboť může sbírat zkušenosti z velkého množství paralelních běhů.

Vícevrstvé neuronové sítě lze trénovat pomocí posilovaného učení ve spojení s genetickým algoritmem [17]. Genetický algoritmus je optimalizační algoritmus inspirovaný biologickou evolucí. Následující pseudokód popisuje využití genetického algoritmu pro trénování neuronové sítě.

```
NN: Neuronová síť
GA: Genetický algoritmus

GA.Vytvoření počateční populace vah pro NN
For i:=0 to početGenerací do
  ForEach jedince vah z~populace
    NN.Predikce akce na základě jedince
    Provedeni akce v~prostředí
    GA.Spočtení hodnoty fitness funkce jedince
  EndFor
  GA.Vytvoření nové populace
EndFor
```

Základní myšlenka učení spočívá ve snaze najít optimální hodnoty vah neuronové sítě pomocí genetického algoritmu. Při tvorbě populace je každý jedinec reprezentován náhodně nagenerovanými vahami neuronové sítě. Neuronová síť pak na základě vstupů prostředí a vah jedince predikuje akci, která by se měla vykonat. Tato akce je v prostředí následně provedena a jedinci je dle stavu prostředí po akci spočtena hodnota fitness funkce. Na základě jedinců s nejvyšší hodnotou fitness funkce je vytvořena nová generace. Během provádění jednotlivých generací algoritmu dochází k postupnému zlepšování hodnot vah jedinců a tím k procesu učení.

2.3 Předzpracování dat

Obrázek (2.9) zobrazuje jednotlivé fáze procesu strojového učení. Předzpracování dat tj. vytvoření datasetu, výběr vhodných atributů a jejich následná transformace zásadním způsobem ovlivňují úspěšnost a rychlost procesu učení. Je mu proto třeba věnovat zvýšený čas a pozornost. Tato sekce popisuje základní úskalí a techniky přípravy dat [18] pro strojové učení.

Obrázek 2.9: Fáze procesu strojového učení

- **Čištění dat**

  Jednou z prvních operací prováděných nad datasetem je čištění dat. Nejčastější chyby v datech zahrnují nesprávně zadané hodnoty, chybějící hodnoty, duplicity, nejednoznačné reprezentace, odlehlé hodnoty, špatné datové typy apod. Prvním krokem k vyčištění datasetu je detekce takovýchto chybných záznamů. Na základě detekovaných chyb je v zásadě možné provést následující operace. Odstranit nevalidní záznamy nebo odebrat celý atribut. Označit prázdné hodnoty jako chybějící. Dogenerovat chybějící nebo chybné hodnoty pomocí statistických nebo k tomu naučených modelů.

- **Volba atributů**

  Existují různé techniky pro výběr podmnožiny vstupních atributů predikce modelu. Irelevantní případně redundantní vstupní atributy mohou zpomalovat nebo zavádět algoritmy učení, což může mít za následek nižší prediktivní výkon. Je žádoucí vyvinout modely pouze s využitím dat, která jsou nutná pro predikci a upřednostňovat tak co nejjednodušší možný, dobře fungující model. Techniky výběru atributů lze dle principu rozdělit na tzv. filter a wrapper metody. Metody typu filter nejprve ocení přínos jednotlivých atributů a následně na základě tohoto ocenění vyberou podmnožinu nezávisle na predikčním modelu. Příkladem takovéto statistické metody je například výběr podmnožiny na základě vzájemné korelace atributů. Metody typu wrapper vytvoří několik podmnožin atributů a otestují je na predikčním modelu. Jako nejlepší hledaná podmno-

žina jsou nakonec zvoleny atributy s nejlepším prediktivním výkonem na validačním datasetu.

- **Datová transformace**

  Datové transformace se používají ke změně typu nebo rozdělení atributů. Typy atributů se dělí na číselné a kategorické. Podtypy číselných jsou celá a reálná čísla. Kategorické typy se rozlišují na typy nominální, ordinální a logické. Většina predikčních modelů vyžaduje na vstupu číselné atributy, je proto nezbytné provést transformaci kategorických atributů na číselné. Metoda, která tuto transformaci umožňuje se nazývá onehotencoding. Princip transformace spočívá ve vytvoření nového číselného atributu za každou možnou hodnotu kategorického atributu. Následně je každému záznamu nastavena hodnota 1 k novému atributu, který odpovídá původní hodnotě a ostatním atributům nastavena hodnota 0.

  Rozsah hodnot číselných atributů se v datové sadě může výrazně lišit. Některé modely strojového učení využívají pro své výpočty euklidovskou vzdálenost. Atributy s velkým rozsahem mají potom v modelu nepřiměřeně velkou váhu. Tento problém je potřeba odstranit přeškálováním rozsahů jednotlivých atributů. K tomu se využívají občas zaměňované techniky zvané normalizace a standardizace [19]. Max-Min normalizace umožňuje normalizovat hodnoty atributů dle vztahu $x_{norm} = \frac{x-min(x)}{max(x)-min(x)}$. Výsledné hodnoty jsou potom přeškálované v rozsahu min-max a při výpočtu vzdáleností nedominuje žádný konkrétní atribut. Nevýhodou při použití normalizace je případ, kdy atribut obsahuje odlehlé hodnoty, které způsobí nerovnoměrné rozložení hodnot při přeškálování. Standardizace oproti tomu není tolik citlivá na odlehlé hodnoty, ale výsledné přeškálované hodnoty nemají pevně daný rozsah. Pro standardizaci platí vztah $x_{stand} = \frac{x-E(x)}{\sigma(x)}$.

- **Refaktorizace atributů**

  Porozumění a rozklíčování atributů často vyžaduje spolupráci s odborníkem na danou oblast. Díky tomu lze pohlížet na atributy v širším kontextu. Na základě takovýchto znalostí lze v případě potřeby přidávat do datasetu nové logické atributy reprezentující stav, případně globální souhrnné statistiky jako je průměr, podíl nebo rozdíl původních atributů. Některé složené atributy jako je například datum a čas lze rozdělit na atributy elementární. Zajímavou technikou, která má experimentálně pozitivní efekt na predikci modelu, se v některých případech jeví použití druhých mocnin číselných atributů.

- **Dimenzionální redukce**

  Dimenzionalita dat je dána počtem vstupních atributů. Každý záznam z datasetu definuje bod v tomto prostoru. S rostoucím počtem dimenzí představuje dataset jen velmi řídké a pravděpodobně nereprezentativní vzorkování tohoto prostoru. Tento problém vede ke snaze vytvořit projekci dat do prostoru nižší dimenze, který si stále zachovává nejdůležitější vlastnosti původních dat. Této projekci se obecně říká dimenzionální redukce a je alternativou k již dříve popsaným metodám výběru nejvhodnější podmnožiny atributů. Nové atributy získané dimenzionální redukcí přímo nesouvisí s původními vstupními atributy, což ztěžuje interpretaci projekce. Nejčastěji používané techniky dimenzionální redukce jsou Analýza hlavních komponent (PCA), Lineární diskriminační analýza (LDA) a autoenkodery. Hlavním přínosem těchto technik je, že odstraňují lineární závislosti mezi vstupními atributy.

Kapitola 3

Návrh a implementace

3.1 Analýza požadavků

Na základě seznámení se zadáním byly identifikovány základní požadavky na funkcionalitu knihovny rozšiřující ABRA Gen o modul implementující vícevrstvé neuronové sítě. Dle zadání modul umožňuje vytvářet vícevrstvé neuronové sítě zvolené topologie. Uživatel má možnost si vytvářené sítě nakonfigurovat přesně dle potřeb řešené úlohy. Pomocí předdefinovaných funkcí lze ve skriptingu připravit data pro trénink a definovat proces učení každé síti přímo na míru.

Topologie sítě

Topologie definuje strukturu a chování sítě. Struktura je dána počtem vrstev, jejich uspořádáním a počty perceptronů v nich. Chování určuje jaké vstupy síť vyžaduje a jaké výstupy poskytuje. Při výběru topologie sítě modul podporuje dvě základní: vícevrstvý perceptron a autoenkoder.

- **MLP - Vícevrstvý perceptron**

  Vícevrstvý perceptron pokrývá metody učení s učitelem a je využitelný pro řešení klasifikačních a regresních úloh.

- **Autoenkoder**

  Autoenkoder umožňuje učení bez učitele a uplatní se při provádění dimenzionální redukce a rozdělování dat do klastrů.

Konfigurovatelné vrstvy

Jednotlivé vrstvy jsou plně konfigurovatelné. Uživatel má tak možnost vytvářet sítě odpovídající zvolené topologii. V každé vrstvě je možné nakonfigurovat následující vlastnosti:

- **Typy vrstev**

  Typ vrstvy definuje místo, chování a způsob užití vrstvy v topologii sítě.

- **Počet perceptronů**

  Počet perceptronů udává dimenzi vrstvy. Změnou počtu perceptronů ve vrstvě lze škálovat výkonnost sítě.

- **Aktivační funkce**

  Aktivační funkce je volitelná a aplikovatelná napříč celou vrstvou.

- **Inicializace vah**

  Inicializace vah perceptronů a biasů je prováděna na základě zvolených metod.

- **Rychlost učení**

  Hyperparametr rychlost učení ovlivňuje rychlost konvergence modelu k řešení.

Perzistence sítě a modelu

Nakonfigurovanou definici sítě je možné perzistentně uložit. Po inicializaci a případném trénování lze následně uložit hodnoty vah modelu.

Předzpracování dat

Podpora pro předzpracování dat před učením a predikcí modelu. Je možné vytvářet datasety se zvolenými atributy, provádět datovou transformaci, dimenzionální redukci a normalizaci dat.

Proces učení a predikce

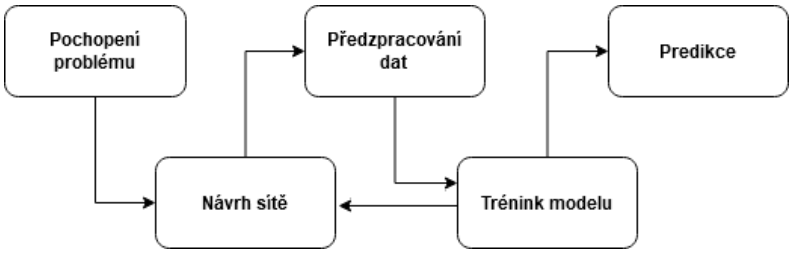
Proces učení je přizpůsobitelný potřebám řešené úlohy. Model je možné s předpřipravenými daty trénovat s volitelným počtem iterací případně ukončující podmínkou. Na základě provedeného tréninku je model schopen provádět predikci výstupu.

3.2 Návrh postupu užití

Souhrn požadavků z předchozí sekce vede k vytvoření předpokládaného postupu užití, který slouží k zpřesnění představy o procesu použití modulu a s ním souvisejících součástí.

Proces využití neuronové sítě v ABRA Gen

Navržený případ užití popisuje proces využití neuronové sítě z pohledu vývojáře systému ABRA Gen. Celý proces je rozdělen na pět na sebe logicky navazujících fází jak ukazuje obrázek 3.1. V obvyklém případě by měly být všechny tyto fáze vývojářem zohledněny a sloužit jako jakýsi návrhový vzor při vývoji řešení.



Obrázek 3.1: Fáze procesu využití neuronové sítě v ABRA Gen

- **Pochopení problému**

  První a nejdůležitější fází procesu je pochopení řešeného problému. Klíčovým aspektem, na který je třeba se zaměřit, je porozumění vstupním datům. Je třeba pochopit význam jednotlivých atributů dat a jejich relevanci pro řešení úlohy. Na základě provedené analýzy lze zvolit vhodnou metodu učení neuronové sítě a optimálně navrhnout topologii sítě.

- **Návrh sítě**

  Návrh sítě začíná volbou typu sítě a konfigurací jednotlivých vrstev. Počty perceptronů ve vstupní a výstupní vrstvě musí odpovídat počtu atributů vstupního datasetu resp. počtu očekávaných výstupů. Počet skrytých vrstev a jejich konfigurace není obecně pevně dána, ale závisí na složitosti problému a citu návrháře sítě. Jelikož má tato volba zásadní vliv na přesnost a rychlost konvergence modelu je obvyklým postupem úprava počtu vrstev a jejich konfigurace na základě dosažených výsledků při tréninku.

- **Předzpracování dat**

  Fáze předzpracování dat je proces tvorby a přípravy datasetu pro trénink modelu. Dataset je sestaven z atributů vybraných na základě strategie výběru atributů. Jednotlivé záznamy datasetu by měly obsahovat vyčištěná a kompletní data. Jelikož jsou na vstupu modelu očekávány číselné hodnoty atributů, je pro kategorické atributy třeba nejprve provést datovou transformaci. Pro lepší výkonnost modelu je vhodné hodnoty atributů normalizovat, případně standardizovat. V případě užití metody učení s učitelem je dataset nakonec rozdělen na tréninkovou a testovací část. Při rozdělení je žádoucí, aby obě části obsahovaly stejné rozdělení dat.

- **Trénink modelu**

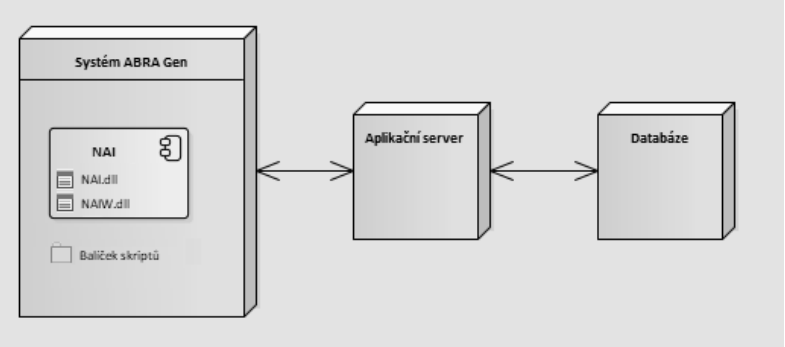
  Před zahájením tréninku je třeba inicializovat hodnoty vah sítě. Inicializace je provedena prvotně náhodnými hodnotami nebo načtením již dříve uložených vah sítě. Od tohoto okamžiku lze o síti mluvit jako o modelu. Trénink modelu probíhá v iteracích. V každé iteraci je proveden trénink nad všemi záznamy z tréninkového datasetu. Při správném průběhu tréninku by celková chyba predikce měla s každou další provedenou iterací klesat a model by tak měl konvergovat k řešení. Trénink je ukončen po provedení všech iterací případně dosažením požadované hodnoty chyby predikce. V případě metody učení s učitelem je zjištěna úspěšnost predikce proti validačnímu datasetu. Trénink případně následně pokračuje dokud není dosaženo požadované úspěšnosti. Je však třeba se vyvarovat efektu přeučení. V případě, že model nekonverguje k řešení nebo konverguje příliš pomalu, přichází v úvahu návrat zpět do fáze návrhu neuronové sítě a provedení nezbytných úprav v návrhu.

- **Predikce**

  Vytrénovaný model je použitelný pro predikci nad vstupními daty. Vstupní data je na vstup modelu třeba předávat ve stejném formátu a po provedení stejných operací předzpracování dat jako byly provedeny nad trénovacím datasetem. Přesnost predikce je nejlepší v případě, kdy vstupní data svojí povahou odpovídající datům, nad kterými byl model trénován.

3.3 Návrh a implementace modulu

Navržený modul `Nexus Artificial Inteligence` dále již jen zkráceně `NAI` rozšiřuje systém ABRA Gen o možnost využívat vícevrstvé neuronové sítě. Integraci modulu v rámci systému popisuje obrázek 3.2. Modul se skládá z knihoven `NAI.dll` a `NAIW.dll`. Prvně zmiňovaná knihovna implementuje veškerou business logiku neuronových sítí. Zde obsažené business objekty lze využít v rámci jiných modulů a jsou zpřístupněny ve skriptingu. Druhá knihovna obsahuje implementaci vizuální číselníkové agendy Neuronové sítě. Zároveň s modulem je vytvořen balíček skriptů, který přidává do skriptingu funkce pro předzpracování dat.

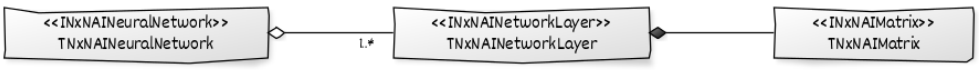


Obrázek 3.2: Integrace modulu NAI v systému ABRA Gen

Implementace modulu je stejně jako v systému ABRA Gen provedena v jazyce Delphi 10.3. Zvolená architektura při návrhu a jmenné konvence implementace odpovídají pravidlům a zvyklostem vyžadovaným pří vývoji tohoto systému.

3.3.1 Business logika

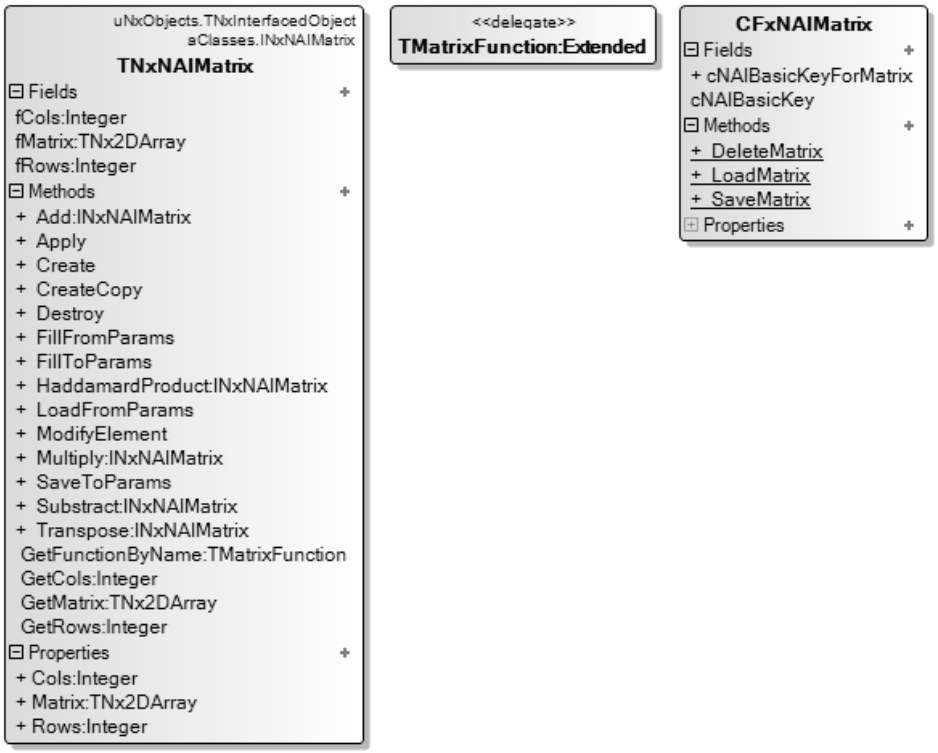
Dle principů objektového programování, které jazyk Delphi podporuje, je navržena hierarchie tříd implementujících požadovanou logiku fungování neuronové sítě (obr 3.3). Třída `TNxNAINeuralNetwork` reprezentuje business objekt neuronové sítě. Tento objekt agreguje vrstvy sítě viz třída `TNxNAINetworkLayer`. Jelikož je výpočetně výhodné provádět výpočet dopředného šíření a zpětné propagace pomocí maticových operací, jsou proto hodnoty vah perceptronů a biasu v jednotlivých vrstvách umístěny do matic viz. třída `TNxNAIMatrix`.



Obrázek 3.3: Diagram tříd business logiky

Matice a maticové operace

Výpočty v neuronové síti jsou prováděny pomocí maticových operací. Bylo proto třeba navrhnout třídu umožňující využívat matice a maticové operace tak, jak jsou matematicky definované. Výsledkem návrhu je třída `TNxNAIMatrix` umístěná v jednotce `oNAIMatrix.pas`. Obrázek 3.3 popisuje navržené rozhraní této třídy. Třída je potomkem třídy `TNxInterfacedObject` a implementuje interface `INxNAIMatrix`.



Obrázek 3.4: Návrh rozhraní třídy TNxNAIMatrix

.

Vlastnosti (properties) počet řádků `Cols` a počet sloupců `Rows` udávají rozměry matice. Tělo matice reprezentuje vlastnost `Matrix` implementovaná pomocí dvoudimenzionálního pole typu Extended. Nová instance třídy lze vytvořit pomocí konstruktoru `Create()` , který má jako parametry počet řádků a sloupců vytvářené matice. Hlubokou kopii matice implementuje kopírovací konstruktor `CreateCopy()` a uvolnění objektu zajišťuje destruktor `Destroy()`.

- **Operace mezi maticemi**

  Rozhraní třídy umožňuje provádět základní operace mezi dvěma maticemi. Jedná se o operace a příslušné funkce: sčítání `Add()`, odčítání `Substract()`, maticové násobení `Multiply()`, Hadamardův součin `HaddamardProduct()` a transpozici matice `Transpose()`.Společným rysem všech vyjmenovaných funkcí je princip, kde levým operandem je zdrojový objekt matice a pravým operandem matice předaná v parametru funkce. Samozřejmě s výjimkou transpozice, kde je transponována samotná matice zdrojového objektu. Po provedení operace vzniká vždy nová matice příslušných rozměrů a původní matice zůstávají nezměněny. Pro lepší představu následuje ukázka implementace kubického algoritmu maticového násobení.

```
function TNxNAIMatrix.Multiply
(const AMatrix: INxNAIMatrix):INxNAIMatrix;
var
  mSum: Extended;
  i, j, k: Integer;
begin
  Assert(fCols = AMatrix.Rows,Format(resInvalidDimensions,
    [fRows,fCols,AMatrix.Rows,AMatrix.Cols]));
  Result := TNxNAIMatrix.Create(fRows, AMatrix.Cols);
  for i:= 0 to fRows - 1 do
  begin
    for j := 0 to AMatrix.Cols - 1 do
    begin
      mSum := 0;
      for k~:= 0 to fCols - 1 do
      begin
        mSum := mSum + fMatrix[i][k] * AMatrix.Matrix[k][j];
      end;
      Result.Matrix[i][j] := mSum;
    end;
  end;
end;
```

  Před samotným provedením výpočtu je provedena kontrola, zda obě matice mají pro operaci kompatibilní rozměry. V případě zjištění problému je chyba odchycena a uživatel informován o chybných rozměrech matic.

- **Funkce aplikovatelné na matici**

  Kromě operací mezi maticemi je nutné umožnit provádět změny hodnot jednotlivých prvků matice. K tomu slouží procedura `ModifyElement()` , která umožňuje nastavit hodnotu prvku matice v parametrech zadaném řádku a sloupci na požadovanou hodnotu.

  Častým požadavkem na funkcionalitu matice je potřeba aplikovat nějakou funkci na všechny její prvky. K tomuto účelu byla navržena procedura `Apply()`.

```
procedure TNxNAIMatrix.Apply
(const AFunction: TNxMatrixFunctions; const AParam: Extended);
var
  i,j: Integer;
  mFunction: TMatrixFunction;
begin
  mFunction := GetFunctionByName(AFunction);
  for i := Low(fMatrix) to High(fMatrix) do
  begin
    for j := Low(fMatrix[i]) to High(fMatrix[i]) do
    begin
      fMatrix[i][j] := mFunction(fMatrix[i][j], AParam);
    end;
  end;
end;
```

  Procedura využívá neveřejnou funkci `GetFunctionByName()`, která mapuje funkci zadanou parametrem výčtového typu `TNxMatrixFunctions` na aplikovanou funkci s následujícím rozhraním.

```
TMatrixFunction = function(const AValue, AParam: Extended): Extended;
```

  Očekávaný parametr `AValue` nese hodnotu prvku matice, zatímco parametr `AParam` je volitelná konstanta, která je využita v případě, že to aplikovaná funkce vyžaduje. Obecně lze aplikovatelné funkce logicky rozdělit do několika skupin podle účelu využití. První skupinou jsou matematické funkce pro sčítání `Plus()`, odčítání `Minus()`, násobení `Mult()` a dělení `Divi()` matice skalárem.

```
function Mult(const AValue, AParam: Extended): Extended;
begin
  Result := AValue * AParam;
end;
```

Další skupinou jsou funkce, které inicializují prvky matice. Funkce `Zero()` a `One()` nastaví hodnoty prvků matice na nuly resp. jedničky. Pro inicializaci náhodnými hodnotami lze využít funkci `RandomNumber()`, která vrací náhodná čísla z intervalu <-1,1> z rovnoměrného rozdělení. Náhodná čísla z normálního rozdělení se střední hodnotou nula a směrodatnou odchylkou danou parametrem `AParam` lze pro inicializaci získat také alternativní funkcí `RandomFromNormal()`.

```
function RandomFromNormal(const AValue, AParam: Extended): Extended;
begin
  Result := RandG(0.0,AParam);
end;
```

Poslední skupinou jsou aktivační funkce využívané neuronovou sítí. Jedná se o funkce sigmoid `Sigmoid()`, tanh `Tanhx()`, parametrický ReLU `ReLU()` a jejich derivace `DSigmoid()`, `DTanhx()` a `DReLU()`.

```
function Sigmoid(const AValue, AParam: Extended): Extended;
begin
  Result := 1 / (1 + Exp(-1 * Avalue));
end;

function DSigmoid(const AValue, AParam: Extended): Extended;
begin
  Result := AValue * (1 - AValue);
end;
```

Do této skupiny patří také funkce `Identity()`, která nijak nemění vstupní hodnotu. Toho lze využít v případě, že aplikace aktivační funkce není chtěná. Funkce `Clip()` omezuje vstupní hodnotu parametru `AValue` v intervalu <`-AParam,AParam`>.

```
function Clip(const AValue, AParam: Extended): Extended;
begin
  Result := AValue;
  if AValue < -1*AParam then
  begin
    Result := -1*AParam;
  end;
  if AValue > AParam then
  begin
    Result := AParam;
  end;
end;
```

- **Perzistence matice**

  Matice je nutné umět perzistentně uložit. Jelikož třída `TNxNAIMatrix` není potomkem business objektu, nelze pro uložení využít perzistentní vrstvu. Perzistence je zajištěna pomocí alternativního mechanizmu využívajícího třídu `TNxParameters`. Tato třída umožňuje vytvářet stromovou strukturu parametrů rozličných datových typů. Pomocí procedury `SavePropertiesForCompany()` lze tyto parametry uložit do databáze. Pro uložení matice je implementována třídová procedura `SaveMatrix()`.

```
class procedure CFxNAIMatrix.SaveMatrix(
  const AMatrix: INxNAIMatrix; const AName: string;
  const AContext: INxContext);
var
  mPars: TNxParameters;
begin
  mPars := TNxParameters.Create;
  try
    AMatrix.SaveToParams(mPars);
    AContext.GetCompanyCache.SavePropertiesForCompany(
      cNAIBasicKeyForMatrix + AName, mPars)
  finally
    mPars.Free;
  end;
end;
```

  Pomocí procedury `SaveToParams()` je matice nejprve zapsána do struktury parametrů. Každý prvek matice je reprezentován parametrem typu Float se jménem složeným z čísla řádku, středníku a čísla sloupce.

```
...
AParams.NewFromDataType(
  dtFloat, IntToStr(i)+';'+IntToStr(j)).AsFloat := fMatrix[i][j];
...
```

  Takto připravená struktura parametrů je následně uložena do databáze. Pro uložení je třeba vytvořit unikátní klíč. Klíč se skládá z konstanty `cNAIBasicKeyForMatrix`, která reprezentuje třídu matice a unikátního názvu ukládané matice `AName`.

  Načtení probíhá pomocí inverzně fungující třídové procedury `LoadMatrix()`, která nejprve z databáze naplní strukturu parametrů a následně v proceduře `LoadFromParams()` načte prvky matice. Jméno každého parametru určuje řádek a sloupec prvku. Odstranění záznamů matice z databáze zajišťuje procedura `DeleteMatrix()`.

Vrstvy neuronové sítě

Neuronové sítě se skládají z vrstev. Vrstvy definují topologii sítě a na základě svého typu a konfigurace ovlivňují chování sítě. Pro tyto účely navržená třída `TNxNAINetworkLayer` umístěná v jednotce `oNAINetworkLayer.pas` implementuje vrstvy neuronových sítí v systému ABRA Gen. Rozhraní třídy je popsáno na obrázku 3.5. Třída dědí vlatnosti z řádkového business objektu `TNxRowBusinessObject` a implementuje interface `INxNAINetworkLayer`.

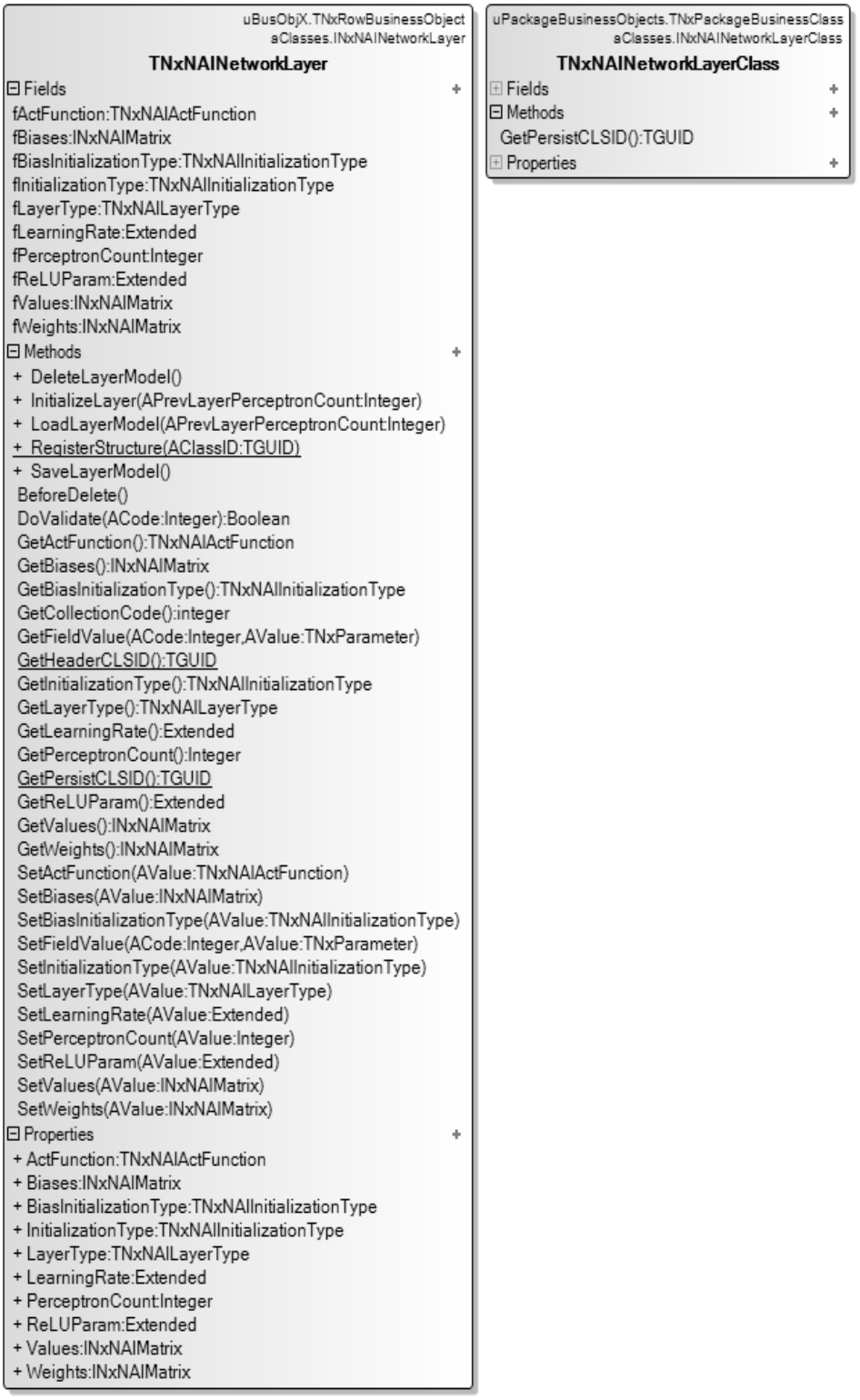
- **Vlastnosti vrstvy**

  Vrstvy v neuronové síti jsou rozděleny podle typu. Vlastnost `LayerType` je implementována pomocí ordinálního výčtového typu `TNxNAILayerType`. Vstupní vrstva slouží k načtení vstupních dat ze vstupu a neobsahuje matice vah. Je to vždy první vrstva v topologii sítě. Topologicky poslední vrstvou je vrstva výstupní. Její výstup odpovídá vypočtené predikcí sítě. Mezi vstupní a výstupní vrstvou je vždy alespoň jedna vrstva skrytá.

  Vlastnost `PerceptronCount` udává počet perceptronů ve vrstvě. Váhy potřebné pro výpočet hodnot perceptronů jsou uloženy v matici `Weights`. Počet řádků matice je roven počtu perceptronů vrstvy a počet sloupců odpovídá počtu perceptronů ve vrstvě předchozí. Matice `Biases` reprezentuje váhy biasu vůči všem perceptronům. Samotné hodnoty perceptronů jsou uloženy po řádcích v matici `Values`.

  Váhy perceptronů a biasu je třeba umět inicializovat vhodnými počátečními hodnotami. Vlastnosti `InitializationType` a `BiasInitializationType` nastavují metody inicializace dané výčtovým typem `TNxNAIInitializationType`. Implementováno je základní předvyplnění nulami a náhodnými čísly z rovnoměrného rozdělení v intervalu <-1,1>. Dále jsou také dostupné pokročilejší metody, konkrétně He a Xavierova inicializace.

  Každá vrstva má taktéž konfigurovatelnou aktivační funkci `ActFunction`, která je pak ve vrstvě aplikována. Výčtový typ `TNxNAIActFunction` pokrývá aktivační funkce sigmoid, tanh a parametriké ReLU. V případě použití parametrického ReLU je možné ještě nastavit jeho parametr `ReLUParam`. Speciálním případem je funkce identity, která zajistí, že aktivační funkce není ve vrstvě aplikována. Hyperparametr udávající rychlost učení lze nastavit pomocí vlastnosti `LearningRate`.



Obrázek 3.5: Návrh rozhraní třídy TNxNAINetworkLayer

- **Inicializace vrstvy**

  Při vytvoření jsou nakonfigurovány jednotlivé vlastnosti vrstvy. Inicializace vrstvy je závislá na vrstvě předchozí. Je proto třeba provádět až v okamžiku, kdy jsou všechny vrstvy sítě definovány. Prvotní inicializaci lze provést pomocí procedury `InitializeLayer()`, která vytvoří nové objekty matic `Weights`, `Biases` a `Values`. Následně jsou první dvě uvedené matice naplněny hodnotami dle nakonfigurované inicializační metody vrstvy.

  Hodnoty z matic vah ve vrstvě je možné perzistentně ukládat a načítat. K načtení slouží procedura `LoadLayerModel()`.

```
procedure TNxNAINetworkLayer.LoadLayerModel(
  const APrevLayerPerceptronCount: Integer);
var
  mContext: INxContext;
begin
  mContext := NxGetContext(ObjectSpace);
  fValues := TNxNAIMatrix.Create(fPerceptronCount,1);
  fBiases := TNxNAIMatrix.Create(fPerceptronCount,1);
  fWeights := TNxNAIMatrix.Create(
              fPerceptronCount, APrevLayerPerceptronCount);

  CFxNAIMatrix.LoadMatrix(fBiases,'Biases'+OID,mContext);
  CFxNAIMatrix.LoadMatrix(fWeights,'Weights'+OID,mContext);
end;
```

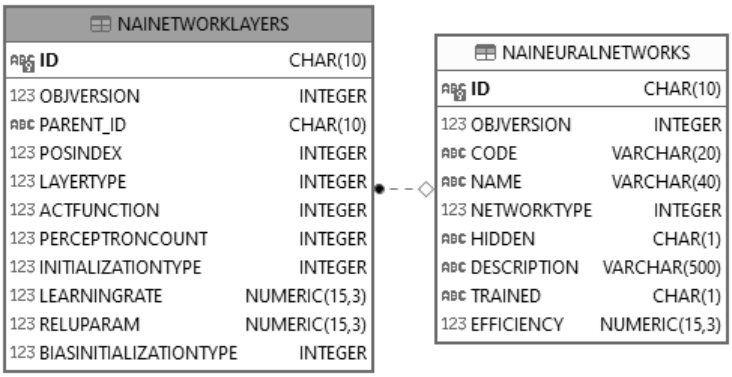
  Obdobně jako u procedury zajišťující prvotní inicializaci jsou nejprve vytvořeny objekty matic. Za zmínku stojí fakt, že pro vytvoření objektu matice `Weights` je třeba znát počet perceptronů v předcházející vrstvě. Následně jsou do připravených instancí načteny uložené hodnoty vah. Při identifikaci vah patřících ke konkrétní vrstvě je využit název matice skládající se z řetězce se jménem vlastnosti a unikátního ID objektu vrstvy.

  Ukládání matic s vahami implementuje procedura `SaveLayerModel()`. Odmazání nepotřebných uložených záznamů lze provést pomocí procedury `DeleteLayerModel()` .

- **Perzistence neuronové sítě a jejích vrstev**

  Třída `TNxNAINetworkLayer` je potomkem řádkového business objektu a dědí tak od něj vybrané metody a vlastnosti. Současně s tím je třída `TNxNAINeuralNetwork` potomkem hlavičkového business objektu, který si drží kolekci řádkových objektů. V tomto případě tak má neuronová síť kolekci vrstev. Kolekce je plněna na základě atributu Parent_ID, který je roven ID hlavičkového objektu. Pořadí vrstev v kolekci určuje atribut Posindex. Hlavičkový objekt zodpovídá za uložení všech řádkových objektů a uložení je proto provedeno v jedné transakci.

  Pro perzistenci sítě a jejich vrstev bylo rozšířeno databázové schéma o nové tabulky viz obrázek 3.6. Aplikační server pro načtení, uložení a smazání objektů využívá příslušné SQL dotazy z repozitoře.



Obrázek 3.6: Rozšíření databázového modelu

Procedura `RegisterStructure()` mapuje atributy z tabulek do záznamů `TNxBusinessFieldDescRecord`. Tyto záznamy obsahují položky se jménem odpovídajícího atributu tabulky, kódem, datovým typem apod.

Při načtení business objektu jsou hodnoty odpovídajících záznamů přiřazeny do příslušných vlastností objektu procedurou `SetFieldValue()`. Při ukládaní jsou naopak hodnoty vlastností zapsány do záznamů v proceduře `GetFieldValue()`. Funkce `DoValidate()` otestuje před uložením, že hodnoty jednotlivých vlastností objektu jsou validní, tj splňují požadovaná integritní omezení. Příkladem může být například validace na povolený rozsah vlastnosti `LearningRate`.

Neuronová síť

Hlavním účelem návrhu neuronových sítí je umožnit provádět predikci pomocí vytrénovaného modelu. Z instance sítě zvolené topologie se inicializací vah stává predikční model. Navržená třída `TNxNAINeuralNetwork` umístěná v jednotce `oNAINeuralNetwork.pas` poskytuje rozhraní popsáné na obrázku 3.7. Třída dědí vlatnosti z hlavičkového business objektu `TNxHeaderBusinessObject` a implementuje interface `INxNAINetworkLayer`.

- **Vlastnosti sítě**

  Objekt neuronové sítě je charakterizován vlastnostmi kód `Code` a jméno `Name`. Textový popis účelu a případu použití sítě reprezentuje vlastnost `Description`.

  Vlastnost `NetworkType` umožňuje zvolit typ sítě z výčtového typu `TNxNAINetworkType`. Podporovaný je typ topologie vícevrstvého perceptronu - MLP, který využívá techniku učení s učitelem pro predikci řešení úloh vyžadující klasifikaci nebo regresi. Druhou možností je topologie autoenkoderu, která je založena na učení bez učitele. Lze jí uplatnit k dimenzionální redukci nebo jako vstup algoritmů hierarchického klastrování.

  Kolekce vrstev sítě je dostupná pomocí vlastnosti `Layers`. Hlavičkový objekt neuronové sítě je zodpovědný za správu řádkových objektů vrstev uložených v této kolekci. K evidenci, zda existují inicializované váhy modelu sítě slouží vlastnost `Trained`. Dosažená úspěšnost predikce sítě je dostupná ve vlastnosti `Efficiency`.

- **Inicializace sítě**

  Inicializací vah instance sítě vzniká predikční model. K tomuto účelu slouží procedura `InitializeNetwork()`, která prvotně inicializuje matice vah ve všech vrstvách sítě. Funkce `LoadNetworkModel()` umožňuje načíst již dříve vytrénované a uložené hodnoty vah modelu. Uložení a smazání vah zajišťují komplementární procedury `SaveNetworkModel()` a `DeleteNetworkModel()`. Společnou charakteristikou všech zde vyjmenovaných metod je, že je lze volat nad již uloženým objektem neuronové sítě. V případě, že dojde ke změně topologie sítě, konkrétně změní se počet perceptronů v některé z vrstev, potom jsou při uložení takovéto sítě odstraněny uložené matice vah, neboť jejich hodnoty v rámci modelu nedávají žádný smysl a model je třeba inicializovat a vytrénovat od začátku.

Obrázek 3.7: Návrh rozhraní třídy TNxNAINeuralNetwork

- **Predikce modelu**

  Hlavním účelem vytvoření modelu je predikce výstupu na základě vstupních dat. K tomu slouží funkce neuronové sítě `Predict()`, která provádí výpočet predikce pomocí algoritmu dopředného šíření. Vstupní data jsou do funkce předána pomocí parametru `AParams`, který je instancí již dříve zmíněné třídy `TNxParameters`. Tato stromová struktura obsahuje seznam parametrů s názvem `Inputs`. Jednotlivé prvky seznamu jsou hodnoty atributů vstupního záznamu datasetu a jsou pomocí procedury `FillFromParams()` načteny do matice hodnot perceptronů ve vstupní vrstvě.

```
...
//načtení vstupů do vstupní vrstvy
mLayer := Layers[0].BusinessObject as INxNAINetworkLayer;
mLayer.Values.FillFromParams(AParams.ParamByName('Inputs').AsList);

//Výpočet hodnot percetronů ve vrstvách
for i := 1 to Layers.Count - 1 do
begin
  mPrevLayer := Layers[i-1].BusinessObject as INxNAINetworkLayer;
  mLayer := Layers[i].BusinessObject as INxNAINetworkLayer;
  mSum := mLayer.Weights.Multiply(mPrevLayer.Values);
  mSum := mSum.Add(mLayer.Biases);
  mSum.Apply(MapActFunction(mLayer.ActFunction,False),mLayer.ReLUParam);
  mLayer.Values := mSum;
end;
...
```

  Po načtení hodnot perceptronů do vstupní vrstvy přichází na řadu určení hodnot perceptronů v ostatních vrstvách. Algoritmus dopředného šíření provádí výpočet postupně od první skryté vrstvy, až k vrstvě výstupní. Váhy perceptronů z aktuálně zpracovávané vrstvy jsou vynásobeny hodnotami perceptronů z předchozí vrstvy. Následně jsou přičteny váhy biasu a aplikována zvolená aktivační funkce.

  Výsledek predikce je vrácen pomocí parametru funkce `AParams`, který obsahuje seznam výstupních parametrů s názvem `Outputs` . Naplnění tohoto seznamu zajišťuje procedura `FillToParams()`. Pokud je typ sítě vícevrstvý perceptron, potom jsou výstupem hodnoty perceptronů z výstupní vrstvy. V případě autoenkoderu jsou jako výstup predikce vráceny hodnoty perceptronů z úzkého hrdla, tj topologicky prostřední skryté vrstvy.

- **Trénink modelu**

  Trénink modelu spočívá v nalezení optimálních vah perceptronů. Funkce `Train()` využívá algoritmu zpětné propagace. Algoritmus začíná zavoláním funkce `Predict()` z předchozí sekce, díky čemuž dojde k napočítání hodnot perceptronů ve všech vrstvách. Vstupní data jsou do funkce opět předána pomocí parametrů `AParams`. Krom seznamu vstupních parametrů s názvem `Inputs` je v parametrech obsažen ještě další seznam `Tagets`, který obsahuje očekávané výstupní hodnoty predikce. Tyto hodnoty jsou po řádcích načteny do matice `mTargets`. V případě, že je síť typu autoenkoder, jsou jako očekávané hodnoty výstupu použity hodnoty perceptronů ve vstupní vrstvě. Na základě hodnot perceptronů ve výstupní vrstvě a očekávaných výstupů je v návratové hodnotě funkce vrácena střední kvadratická chyba predikce.

```
...
//vypocet matice delta ve vrstvach
for i := Layers.Count - 1 downto 1 do
begin
  mPrevLayer := Layers[i-1].BusinessObject as INxNAINetworkLayer;
  mLayer := Layers[i].BusinessObject as INxNAINetworkLayer;
  if i = Layers.Count - 1 then  //vypocet delty v~L-te vrstve
  begin
    mErrors := mLayer.Values.Substract(mTargets);
    mTmp := TNxNAIMatrix.CreateCopy(mLayer.Values);
    mTmp.Apply(MapActFunction(mLayer.ActFunction,True),mLayer.ReLUParam);
    mDeltas := mTmp.HaddamardProduct(mErrors);
  end
  else
  begin
    mNextLayer := Layers[i+1].BusinessObject as INxNAINetworkLayer;
    mTmp := mNextLayer.Weights.Transpose;
    mDeltas := mTmp.Multiply(mDeltas);
    mTmp :=  TNxNAIMatrix.CreateCopy(mLayer.Values);
    mTmp.Apply(MapActFunction(mLayer.ActFunction,True),mLayer.ReLUParam);
    mDeltas := mDeltas.HaddamardProduct(mTmp);
end;
...
```

Zpětná propagace probíhá postupně směrem od výstupní vrstvy k vrstvě vstupní. Výpočet je prováděn pomocí pravidla delta, které je popsáno v kapitole Analýza axiomy 2.28 - 2.31 a umožňuje pro určení gradientu použít předpočítanou matici delta z předchozí vrstvy. Delta matice ve

výstupní vrstvě (axiom 2.28) je určena pomocí gradientu střední kvadratické chyby, vypočtené jako rozdíl matice hodnot perceptronů ve výstupní vrstvě a matice očekávaných výstupů. Matice hodnot perceptronů ve výstupní vrstvě je po aplikaci derivace aktivační funkce následně po složkách vynásobena maticí tohoto gradientu. Delty v ostatních vrstvách (axiom 2.29) se spočtou jako součin matice delta vypočtené v předchozí vrstvě a transponované matice vah z předchozí vrstvy. Matice s tímto součinem je následně vynásobena po složkách hodnotami perceptronů z aktuálně zpracovávané vrstvy po aplikaci derivace aktivační funkce.

Gradient chyby predikce v závislosti na vahách biasu je roven matici delta (axiom 2.31). Gradient chyby predikce v závislosti na vahách perceptronů (axiom 2.30) se určí jako součin matice delta s transponovanou maticí hodnot perceptronů z následující vrstvy. Nové hodnoty vah se potom pro oba případy vypočtou jako rozdíl původních hodnot vah a gradientu chyby skalárně vynásobeného rychlostí učení.

```
...
//výpočet nových hodnot vah
mGradients := TNxNAIMatrix.CreateCopy(mDeltas);
mGradients.Apply(mfMultiply,mLayer.LearningRate);
mGradients.Apply(mfClip,1);
mLayer.Biases := mLayer.Biases.Substract(mGradients);
mTmp := mPrevLayer.Values.Transpose;
mGradients := mDeltas.Multiply(mTmp);
mGradients.Apply(mfMultiply,mLayer.LearningRate);
//gradient clipping
mGradients.Apply(mfClip,1);
mLayer.Weights := mLayer.Weights.Substract(mGradients);
...
```

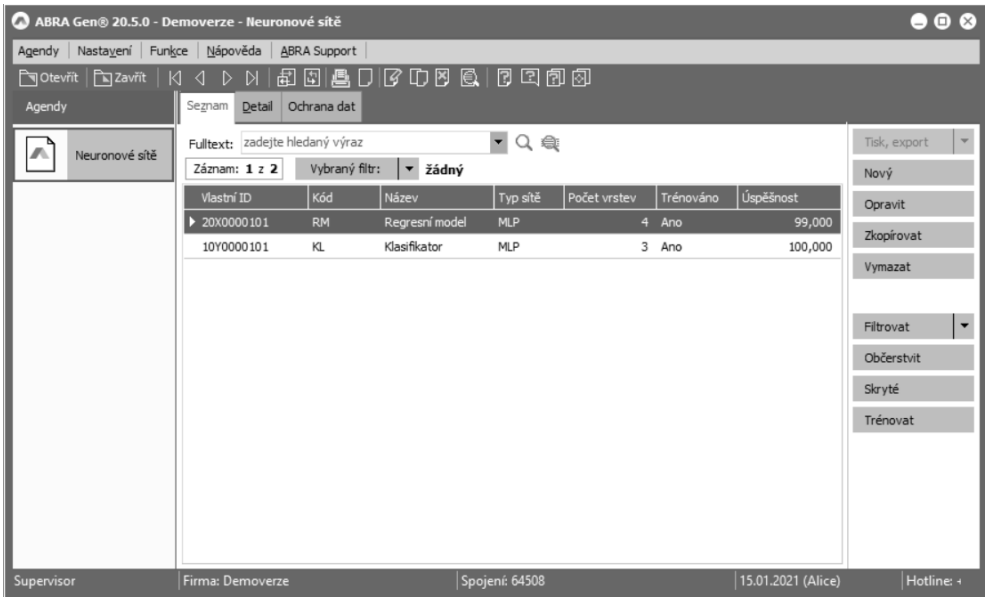
Během procesu učení, může docházet k situacím, kdy některá z upravovaných hodnot vah je měněna příliš prudce a znemožňuje tak dosažení konvergence modelu. Algoritmus proto využívá techniku klipování gradientu, která omezí maximální absolutní hodnotu změny vah.

3.3.2 Číselníková agenda Neuronové sítě

Grafické uživatelské rozhraní systému ABRA Gen se skládá z agend, které uživateli umožňují ovládat specifickou část systému. Pro usnadnění vytváření a konfigurace neuronových sítí byla navržena číselníková agenda Neuronové sítě. Tato agenda obsahuje seznam neuronových sítí a zpřístupňuje jejich vlastnosti. Implementace je uložena v jednotce `fNAINeuralNetwork.pas` a její definicí formuláře `fNAINeuralNetwork.dfm`. Číselníkovou agendu je možné otevřít jako tzv. malý číselník při výběru hodnot do číselníkových položek. Plné zobrazení agendy zajišťuje velký číselník, který obsahuje dvě pro číselníkovou agendu standardní záložky Seznam a Detail.

Záložka Seznam

Výchozím místem po otevření agendy Neuronové sítě je záložka Seznam viz obrázek 3.8. Hlavní vizuální část vyplňuje grid, ve kterém každý řádek odpovídá jednomu business objektu neuronové sítě. Sloupce gridu poté zobrazují jednotlivé vlastnosti objektů. Na objekt z aktuálně vybraného řádku gridu je možné aplikovat některou z funkcí umístěných v panelu funkcí na pravé straně okna aplikace. Je tak možné vytvořit novou neuronovou síť a opravit, zkopírovat, smazat, nebo trénovat síť stávající. Díky implementaci skrývání záznamů je v agendě při mazání neuronové sítě dostupná možnost síť pouze skrýt a umožnit ji v případě potřeby opět obnovit.

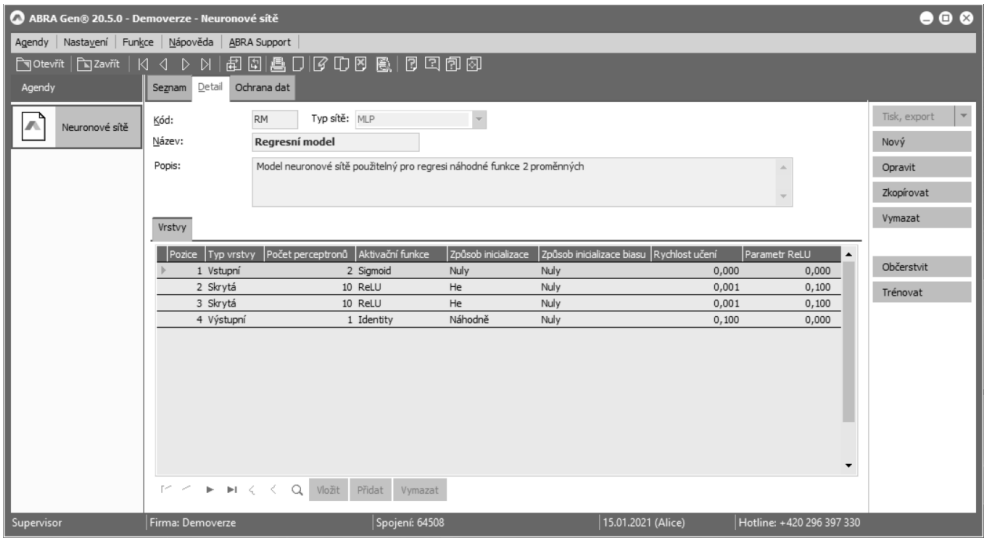


Obrázek 3.8: Návrh GUI záložky Seznam

Záložka Detail

Přechodem na záložku Detail (obr. 3.9) lze zobrazit podrobné informace o aktuálně vybrané neuronové sítí. Funkce Nový, Opravit a Zkopírovat přepnou formulář do editačního režimu. V horní části formuláře lze nastavit vlastnosti hlavičkového objektu neuronové sítě. Důležitý je zejména typ sítě, který ovlivňuje způsob vyhodnocení validace vrstev sítě při ukládání objektu.

Ve spodní polovině formuláře je umístěna záložka Vrstvy, na které je možné konfigurovat vrstvy neuronové sítě. Každý řádek zde umístěného gridu odpovídá business objektu vrstvy neuronové sítě. Pomocí pod gridem přidruženého navigátoru lze jednotlivé vrstvy přidávat, mazat a případně měnit jejich pořadí. Sloupce gridu odpovídají jednotlivým vlastnostem vrstvy. Při ukládání je za správné uložení a zajištění validace vrstev zodpovědný hlavičkový business objekt.



Obrázek 3.9: Návrh GUI záložky Detail

Akční funkce Trénovat umožňuje provést trénink zvolené sítě z vizuální části aplikace v případě, že je pro danou síť ve skriptingu definován proces jejího učení. Podrobnosti jakým způsobem definovat proces učení je vysvětlen v následující sekci zabývající se skriptováním.

3.3.3 Rozšíření funkcionality skriptingu

Každý problém řešený pomocí neuronových sítí vyžaduje unikátní přístup z pohledu výběru a předzpracování dat a navržení vhodného procesu učení. Je proto nutné zajistit vývojové prostředí vhodné pro takovýto individuální přístup a umožňující provedení zmíněných požadavků. V systému ABRA Gen lze takovéto funkcionality doprogamovat pomocí skriptingu.

Vytažení do skriptingu

K používání business objektů ze skriptingu bylo třeba vytvořit skriptingové obálky, které zpřístupňují zvolenou část jejich rozhraní. Objekt neuronové sítě je zpřístupněn pomocí obálky umístěné v jednotce `ufsoNAINeuralNetwork`. Vrstvy neuronové sítě pak obsahuje jednotka `ufsoNAINetworkLayer`. Následující kód zobrazuje rozhraní metod a vlastností neuronové sítě přístupných ze skriptingu.

```
...
with AddClass(TNxNAINeuralNetwork, 'TNxHeaderBusinessObject',
  resClass_TNxNAINeuralNetwork_Description) do //DoNotLocalize
  begin
    AddMethod('procedure InitializeNetwork;', _InitializeNetwork ...
    AddMethod('procedure Predict(AParams: TNxParameters);', _Predict, ...
    AddMethod('function Train(AParams: TNxParameters): Extended;', ...
    AddMethod('procedure SaveNetworkModel;', _SaveNetworkModel, ...
    AddMethod('function LoadNetworkModel: Boolean;', _LoadNetworkModel, ...
    AddProperty('Code', 'String', GetProp, SetProp, ...
    AddProperty('Name', 'String', GetProp, SetProp, ...
    AddProperty('NetworkType', 'TNxNAINetworkType', GetProp, SetProp, ...
    AddProperty('Description', 'String', GetProp, SetProp, ...
    AddProperty('Layers', 'TNxCustomBusinessMonikerCollection', ...
    AddProperty('Trained', 'Boolean', GetProp, SetProp, ...
    AddProperty('Efficiency', 'Extended', GetProp, SetProp, ...
  end;
...
```

Takto rozšířené skriptingové rozhraní umožňuje editaci stávajících a vytváření nových business objektů neuronových sítí. Je možné konfigurovat jednotlivé vrstvy sítě stejně jako je tomu v GUI agendy Neuronových sítí. Při implementaci procesu učení ve skriptingu pak lze využít metody pro inicializaci modelu a metodu neuronové sítě určenou pro jeho trénink. Samozřejmostí je i provádění predikce.

Háčky

Chování business objektů, agend a dalších funkcí systému je možné dodefinovat pomocí tzv. háčků. Háčky jsou registrované metody skriptingu volané v určitý okamžik uvnitř jiných metod objektů systému. Jejich účelem je provedení definovaného skriptingového kódu, který pozmění původní chování metod. V agendě Neuronové sítě byl vytvořen háček `Do_Train_Hook`, který je volaný uvnitř akční funkce Trénovat. Díky tomu lze ve skriptingu naprogramovat proces učení pro aktuálně aktivní business objekt neuronové sítě v agendě.

Balíček skriptů

Skriptingová řešení jsou v systému ABRA Gen umístěna v takzvaných balíčcích skriptů, v rámci kterých lze implementovat chování jednotlivých háčků a vytvářet knihovny podpůrných funkcí. Za účelem využití funkcionality neuronových sítí ve skriptingu je připraven balíček `NeuralNetwork.xml`. Balíček je rozdělen na tři části.

První část obsahuje implementaci výše zmíněného háčku `Do_Train_Hook`. Je zde navržen princip zajištění tréninkového procesu pro aktuálně aktivní neuronovou síť. V praxi to znamená, že je na základě ID sítě zavolána unikátní funkce definující tréninkový proces pro tuto konkrétní síť. Tréninkové funkce jsou umístěné v druhé části balíčku v knihovně `Networks`, kde je třeba je pro každou síť dodefinovat.

Třetí část balíčku je tvořena knihovnou `NewtworkFunc`, která obsahuje podpůrné funkce pro vytvoření datasetu a předzpracování dat. Základním předpokladem pro získávání dat pro trénink je využití dat z relační databáze. Prvotní výběr atributů vytvářeného datasetu je tak proveden na úrovni SQL dotazu. K uchování dat po načtení z databáze a další datové transformace se předpokládá využití paměťového datasetu. Následující řádky kódu popisují způsob načtení dat do datasetu pomocí SQL dotazu.

```
mSQLQuery := 'SELECT * FROM TABLE T';
mDataset := TMemoryDataset.Create(nil);
AOS.SQLSelect2(mSQLQuery,mDataset);
```

Po provedení těchto operací obsahuje dataset atributy s názvem a datovým typem odpovídajícím SQL dotazu. Pro usnadnění další práce s datasetem lze využít následující funkce předzpracování dat a datové transformace.

V případě, že je některý z atributů kategorického typu, je zpravidla výhodné provést datovou transformaci založenou na principu onehotencoding. V knihovně k tomu slouží funkce `OneHotEncodeField()`, která na základě v parametrech

předaného datasetu, jména transformovaného atributu a seznamu očekávaných kategorii založí nové atributy a naplní dataset příslušnými daty. Původní atribut je z datasetu odstraněn. Funkce si umí poradit se situací, kdy je kategorický datový typ transformovaného atributu typu řetezec i celé číslo, neboť v systému ABRA Gen jsou často výčtové datové typy v databázi ukládány jako celá čísla.

Pro standardizaci hodnot atributu lze využít proceduru `StandardizeField()`. Hodnoty zvoleného atributu jsou následně přepočítány na základě střední hodnoty a směrodatné odchylky.

Obdobně existuje i funkce `NormalizeField()` pro provedení MinMax normalizace. Hodnoty vybraného atributu jsou přeškálovány v rozsahu zadaného minima a maxima. Minimální a maximální hodnotu atributu z datasetu lze určit pomocí funkce `GetMinFieldValue()` resp. `GetMaxFieldValue()`.

```
//Min Max normalizace fieldu z~datasetu
procedure NormalizeField(ADataset: TMemoryDataset;
  const AFieldName: String; AMin, AMax: Extended);
begin
  ADataset.First;
  while not ADataset.Eof do
  begin
    ADataset.Edit;
    ADataset.FieldByName(AFieldName).AsFloat :=
      (ADataset.FieldByName(AFieldName).AsFloat - AMin) / (AMax - AMin);
    ADataset.Post;
    ADataset.Next;
  end;
end;
```

Normalizované hodnoty atributů lze přeškálovat zpět na původní hodnoty pomocí funkce `RevertNormalization()`. To je třeba například u výstupních atributů pokud chceme zobrazit predikci v původním škálování.

Při využití metody učení s učitelem je také třeba umět rozdělit dataset na tréninkovou a testovací část. Procedura `SplitDataset()` automaticky rozdělí vstupní dataset na dva nové datasety se zadaným počtem záznamů.

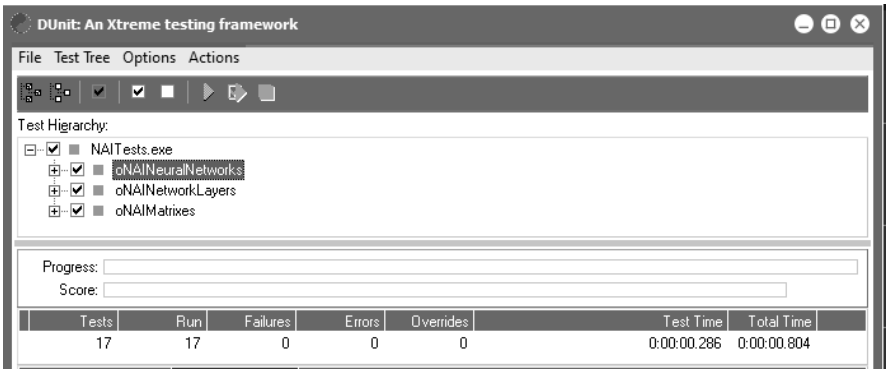
Kapitola 4

Testování

Testování funkcionality navrženého modulu probíhalo po celou dobu implementace. Proces testování lze rozdělit do tří základních oblastí testování. Základní funkcionalita business logiky byla otestována automaticky pomocí jednotkových testů. Vizuální část modulu reprezentovaná agendou Neuronové sítě, byla otestována ručně. Jednalo se zejména o proklikání všech funkcí grafického uživatelského rozhraní a otestování správného rozložení prvků na formuláři. Poslední část testování se zabývala použitelností modulu jako celku a zároveň se snaží prakticky ověřit správnost navrženého procesu využití neuronové sítě v systému ABRA Gen.

4.1 Jednotkové testy

Jednotkové testy pokrývají oblast testování business logiky neuronových sítí. Jak již jejich název a obrázek 4.1 napovídá testy testují funkcionalitu z jednotek `oNAIMatrixes`, `oNAINetworkLayers` a `oNAINeuralNetworks`. Uvedené testy se skládají z dílčích testovacích scénářů.



Obrázek 4.1: Průběh jednotkových testů

Jedním příkladem za všechny je testovací scénář `TestMultiply`, který automaticky testuje funkci `Multiply()` pro maticové násobení z třídy `TNxNAIMatrix`. Test začíná vytvořením matice se dvěma řádky a třemi sloupci. Prvky matice jsou následně inicializovány na hodnoty rovné součtu svých řádkových a sloupcových indexů. Pomocí maticové transpozice je vytvořena transponovaná kopie této matice. Obě matice jsou následně vynásobeny pomocí testované funkce pro maticové násobení. V závěru testu je provedena kontrola, zda má výsledná matice vzniklá násobením odpovídající rozměry a její prvky správné hodnoty.

```
//test správnosti výpočtu maticového násobení
procedure TNxNAIMatrixesTests.TestMultiply;
var
  i,j: Integer;
  mNAIMatrixTrans, mNAIOutMatrix: INxNAIMatrix;
begin
  fNAIMatrix := TNxNAITestingMatrix.Create(2,3);
  for i := 0 to fNAIMatrix.Rows - 1 do
  begin
    for j := 0 to fNAIMatrix.Cols - 1 do
    begin
      fNAIMatrix.Matrix[i][j] := i+j;
    end;
  end;
  mNAIMatrixTrans := fNAIMatrix.Transpose;
  mNAIOutMatrix := fNAIMatrix.Multiply(mNAIMatrixTrans);
  CheckEquals(mNAIOutMatrix.Rows, 2, cCountRowsNotMatch');
  CheckEquals(mNAIOutMatrix.Cols, 2, cCountColsNotMatch);
  CheckEquals(mNAIOutMatrix.Matrix[0][0], 5, cValueNotMatch);
  CheckEquals(mNAIOutMatrix.Matrix[0][1], 8, cValueNotMatch);
  CheckEquals(mNAIOutMatrix.Matrix[1][0], 8, cValueNotMatch);
  CheckEquals(mNAIOutMatrix.Matrix[1][1], 14, cValueNotMatch);
...
```

Testovací scénáře pokrývají nejdůležitější funkce z tříd umístěných v daných jednotkách, u kterých má význam takovéto testování provádět.

4.2 Modelové případy užití

Testování použitelnosti implementovaného modulu se snaží pomocí modelových případů užití ověřit správnost navrženého procesu využití neuronových sítí v systému ABRA Gen. Následující dva případy užití řeší předpokládané nejčastější typy úloh a slouží jako návrhový vzor postupu řešení pro další vývojáře. Zdrojový kód pro přípravu dat a trénink modelů je dostupný ve skriptingové knihovně `Networks`.

Klasifikace stavu dokončeného výrobku

- **Pochopení problému**

  Výrobní firma produkuje automobily typu kabriolet. V poslední době se i přes veškerou snahu o testování potýká s nespolehlivostí uchycení a seřízení motorů uvnitř karoserie. Jednou z operací technologického postupu, podle které výroba probíhá, je osazení a seřízení motoru. Pracovník, který v kabrioletu provedl montáž motoru, odepíše do systému ABRA Gen pracovní lístek s detaily montáže. Zároveň s tím jsou do systému pomocí IoT čidel zaznamenány hodnoty použitých momentů síly na šrouby, kterými je motor v karoserii uchycen. Pokud se po čase ukáže, že byl s motorem nějaký problém, vznikne k pracovnímu lístku, v rámci kterého byl zamontován, záznam o neshodě. Nejčastějšími příčinami vzniku neshody jsou uvolnění motoru, způsobené nedostatečně utaženými šrouby a chybné seřízení motoru, zapříčiněné nevyvážené utaženými šrouby. Je proto třeba odhalit možné nesprávně zamontované motory již během procesu výroby, aby se předešlo nežádoucímu chování u zákazníka.

  Na základě popsané situace lze úlohu definovat jako klasifikační problém. Po provedení montáže motoru je třeba předpovědět, v jakém ze tří stavů se motor nachází. Motor muže být v pořádku, špatně seřízen nebo nedostatečně utažen. Případy, kdy byl motor po zamontování v chybovém stavu jsou evidovány pomocí záznamů o neshodách. V ostatních případech se předpokládá že byl v pořádku. Pro každou montáž motoru jsou pomocí IoT senzorů $M1$ až $M4$ zaznamenané čtyři číselné hodnoty momentů utažení šroubů. Na základě znalosti těchto vstupů a k nim očekávaných výstupů stavu montáže lze trénovat predikční model neuronové sítě pomocí metody učení s učitelem.

- **Návrh sítě**

  Klasifikační problémy jsou řešitelné pomocí neuronové sítě typu vícevrstvý perceptron. Tato síť je vytvořena pomocí GUI agendy Neuronové

sítě. Vstupní vrstva obsahuje čtyři perceptrony, každý odpovídá jednomu z IoT senzorů. Výstupní vrstva bude mít pro každý predikovaný stav motoru jeden perceptron, celkově tedy tři. Aby výstup každého perceptronu odpovídal pravděpodobnosti daného stavu, je v této vrstvě použita aktivační funkce sigmoid. Vhodným způsobem inicializace vah perceptronů je potom Xavierova metoda. Rychlost učení je zde zvolena jako jedna desetina. Pro řešení této úlohy je zpočátku použita jedna skrytá vrstva s deseti perceptrony. Jako aktivační funkci se díky rychlosti výpočtu nabízí aplikovat funkci ReLU s parametrem jedna desetina. Inicializace vah perceptronů se pro tuto aktivační funkci provádí metodou He. Váhy biasů ve všech vrstvách jsou inicializovány nulami.

- **Předzpracování dat**

  Ve fázi předzpracování dat je třeba připravit dataset pro trénink modelu neuronové sítě. K tomu se hodí využít připravený háček akční funkce Trénovat v agendě Neuronových sítí. Ve skriptingu je implementována funkce zajišťující předzpracování dat a trénink modelu sítě z předchozího odstavce. Pomocí SQL dotazu do databáze je naplněn dataset daty pro trénink. Dataset ve výchozím stavu obsahuje vstupní atributy momentů $M1$ až $M4$, které jsou typu reálných čísel a ordinální celočíselný atribut $TARGET$, který reprezentuje jeden z predikovaných stavů zamontování motoru. Protože automobily nejsou skutečně vyráběny a neexistují tak naměřené hodnoty IoT senzorů, jsou záznamy datasetu uměle vytvořeny. Aby nebylo třeba data na několika místech složitě importovat do systému, je jejich definice umístěna přímo v těle skriptu, odkud jsou načtena do datasetu. Naplněný dataset je následně upravován pro trénink. Příprava spočívá v normalizaci vstupních atributů $M1$ až $M4$, což urychlí konvergenci modelu. Dále je provedena datová transformace ordinálního atributu $TARGET$ pomocí metody onehotencoding, jejímž výsledkem je samostatný výstupní atribut pro každý stav montáže. Na závěr je dataset rozdělen na tréninkovou a testovací část.

- **Trénink modelu**

  Trénink navazuje ve skriptingu na předzpracování dat. Váhy modelu neuronové sítě jsou buď prvotně inicializovány nebo načteny dříve uloženými hodnotami. Proces tréninku probíhá v iteracích. V každé iteraci jsou postupně všechny záznamy z tréninkového datasetu předány přes parametry do funkce zajišťující trénink modelu sítě. Na základě jednotlivých výstupů je pak spočtena střední kvadratická chyba predikce modelu v iteraci. Velikost této chyby by měla s každou další provedenou iterací klesat. Pokud chyba neklesá nebo klesá příliš pomalu, potom je třeba zvážit následující postup. Model mohl díky nepříznivě vygenerované startovací kombinaci vah uváznout v lokálním minimu. Je proto

třeba vyzkoušet, zda se schopnost konvergence nezlepší novou inicializací vah a opakovat proces učení. Pokud tento postup nepomůže, je třeba se vrátit do fáze návrhu topologie neuronové sítě a upravit počty skrytých vrstev a neuronů v nich, případně změnit použité aktivační funkce nebo upravit hodnoty rychlostí učení ve vrstvách.

- **Predikce**

  Po dokončení tréninku je otestována dosažená schopnost predikce modelu proti testovací části datasetu. V případě, že model nemá požadovanou úspěšnost, lze pokračovat v tréninku modelu v dalších iteracích. Je však třeba dát pozor na případ, kdy přílišným trénikem nad trénovacími daty dojde k efektu přeučení. Model je pak příliš fixován na tréninková data a ztrácí schopnost generalizace.

Regrese neznámé funkce

- **Pochopení problému**

  V systému Abra Gen byly během uplynulého období zaznamenávány hodnoty číselných veličin x a y. Na konci období byla každé dvojici známa výsledná hodnota z závislá na kombinaci hodnot obou proměnných. V následujícím období chceme předpovídat hodnoty z v okamžiku získání nových hodnot veličin x a y, aniž by bylo nutné čekat na skutečné hodnoty na konci období.

  Ze zadání úlohy vyplývá, že je třeba provést regresi neznámé funkce. Jedná se o funkci dvou proměnných x a y. Pro trénink jsou dostupná data z minulého období, u kterých známe očekávaný výstup z. Vhodnou strategií tréninku neuronové sítě je tedy učení s učitelem.

- **Návrh sítě**

  V agendě Neuronové sítě je třeba pro řešení regresní úlohy vytvořit novou neuronovou síť typu vícevrstvý perceptron. Topologie sítě je tvořena pěti vrstvami. Vstupní vrstva má dva perceptrony, každý odpovídá jedné ze vstupních proměnných. Výstupní vrstva musí mít v tomto případě jeden perceptron a nebude zde aplikována aktivační funkce. Způsob inicializace vah perceptronů bude náhodný a váhy biasu inicializujeme nulami. Rychlost učení nastavíme na jednu setinu. Počet skrytých vrstev je závislý na složitosti hledané funkce a požadované přesnosti modelu. Ve výchozím stavu proto přidáme do sítě 3 skryté vrstvy s deseti až padesáti perceptrony. Jako aktivační funkci použijeme ve všech vrstvách ReLU s parametrem jedna desetina. Způsob inicializace vah perceptronů

zvolíme v závislosti na vybrané aktivační funkci metodou He. Biasy inicializujeme opět nulami. Rychlosti učení budou mít hodnotu jedna tisícina.

- **Předzpracování dat**

  Předzpracování dat je stejně jako v předchozím případě provedeno v háčku akční funkce Trénovat. Jelikož se jedná o modelový příklad, trénink nebude prováděn na reálně naměřených datech, ale na datech náhodně vygenerovaných. Vygenerovaný dataset obsahuje tisíc záznamů a obsahuje vstupní atributy x a y a očekávaný výstup z. Pro lepší konvergenci modelu jsou hodnoty všech atributů normalizovány. Normalizovaný dataset je následně rozdělen na tréninkovou a testovací část.

- **Trénink a predikce modelu**

  Trénink i predikce probíhají opět ze skriptingu za stejných podmínek a s použitím stejných technik jako v předchozím případě užití.

Závěr

Řešením diplomové práce je knihovna rozšiřující ABRA Gen o modul implementující vícevrstvé neuronové sítě. Teoretická část práce seznamuje čtenáře se systémem ABRA Gen a popisuje základní principy fungování vícevrstvých neuronových sítí a techniky předzpracování dat vhodné pro jejich učení. Všechny tyto poznatky byly následně využity při návrhu a implementaci modulu.

Na základě požadavků a předpokládaného postupu užití byl navržen a implementován výsledný modul NAI. Modul umožňuje navrhovat vícevrstvé sítě vybraných topologií. Sítím je možné přidávat vrstvy s konfigurovatelným počtem perceptronů, aktivačními funkcemi a způsoby inicializace vah. Model vzniklý inicializací navržené sítě je možné trénovat a použít pro predikci. Modul podporuje perzistenci vytvořených neuronových sítí a vytrénovaných vah modelu. Součástí řešení je balíček skriptů obsahující funkce pro předzpracování dat ve skriptingu, které pokrývají zejména oblast datové transformace, standardizace a normalizace.

Navržený a implementovaný modul umožňuje využívat vícevrstvý perceptron pro řešení klasifikačních a regresních úloh s využitím metody učení s učitelem. Při učení bez učitele je podporována topologie sítě autoenkoder, která je vhodná pro dimenzionální redukci a jako případný vstup algoritmů hierarchického klastrování. V případě potřeby je také možné sítě trénovat metodou posilovaného učení.

Základní funkcionalita implementovaného modulu byla otestována pomocí jednotkových testů. Pro otestování celkové využitelnosti modulu a správnosti navrženého konceptu byly ve skriptingu implementovány dva modelové případy užití, které zároveň slouží jako vzorové příklady užití.

V rámci řešení se podařilo splnit všechny cíle, které vyplynuly ze zadání této diplomové práce. V současnosti je tak modul připraven pro integraci do dalších modulů systému ABRA Gen, zejména do modulů Výroby a Kapacitního

plánování. Modul je z povahy návrhu možné dále rozšiřovat o nové funkcionality a vylepšení. Zároveň se taktéž nabízí příležitost provést zrychlení výpočtů logiky modulu pomocí paralelizace maticových výpočtů.

Literatura

[1] Popis systému ABRA Gen. [online], [cit. 2020-10-24]. Dostupné z: https://help.abra.eu/cs/21.0/G3/Content/Part20_Zakladni_popis_systemu/system_rozcestnik.htm

[2] Výkladový slovník. [online], [cit. 2020-10-24]. Dostupné z: https://help.abra.eu/cs/21.0/G3/Content/Part60_Vykladovy_slovnik/vykladovy_slovnik.htm

[3] CS231n Convolutional Neural Networks for Visual Recognition. [online], [cit. 2020-10-24]. Dostupné z: https://cs231n.github.io/neural-networks-1/#bio

[4] Biological neuron model. [online], [cit. 2020-10-24]. Dostupné z: https://en.wikipedia.org/wiki/Biological_neuron_model

[5] Dhruvil Karani: Artificial Intelligence - Accelerate the Power with Neural Networks. [online], July 2, 2019, [cit. 2020-10-24]. Dostupné z: https://dimensionless.in/artificial-intelligence-accelerate-the-power-with-neural-networks/

[6] John McGonagle, José Alonso García, Saruque Mollick: Feedforward Neural Networks. [online], [cit. 2020-10-24]. Dostupné z: https://brilliant.org/wiki/feedforward-neural-networks/

[7] Michael A. Nielsen: A visual proof that neural nets can compute any function. [online], ©2015, [cit. 2020-10-24]. Dostupné z: http://neuralnetworksanddeeplearning.com/chap4.html

[8] Neural Network Bias: Bias Neuron, Overfitting and Underfitting. [online], [cit. 2020-10-24]. Dostupné z: https://missinglink.ai/guides/neural-network-concepts/neural-network-bias-bias-neuron-overfitting-underfitting/

[9] 7 Types of Neural Network Activation Functions: How to Choose? [online], [cit. 2020-10-24]. Dostupné z: https://missinglink.ai/guides/neural-network-concepts/7-types-neural-network-activation-functions-right/

[10] Shruti, J.: Introduction to different activation functions for deep learning. [online], 2018, [cit. 2020-10-24]. Dostupné z: https://medium.com/@shrutijadon10104776/survey-on-activation-functions-for-deep-learning-9689331ba092

[11] Unsupervised Feature Learning and Deep Learning Tutorial. [online], [cit. 2020-10-24]. Dostupné z: http://ufldl.stanford.edu/tutorial/supervised/MultiLayerNeuralNetworks/

[12] Saurabh Yadav: Weight Initialization Techniques in Neural Networks. [online], November 9, 2018, [cit. 2020-10-24]. Dostupné z: https://towardsdatascience.com/weight-initialization-techniques-in-neural-networks-26c649eb3b78

[13] Simeon Kostadinov: Understanding Backpropagation Algorithm. [online], August 8, 2019, [cit. 2020-10-24]. Dostupné z: https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd

[14] Chain rule - Wikipedia. [online], 2001-, [cit. 2020-10-24]. Dostupné z: https://en.wikipedia.org/wiki/Chain_rule

[15] Hind Sellouk: Matrix Based Back-propagation. [online], July 24, 2018, [cit. 2020-10-24]. Dostupné z: https://medium.com/@hindsellouk13/matrix-based-back-propagation-fe143ce2b2df

[16] Jeremy Jordan: Introduction to autoencoders. [online], March 19, 2018, [cit. 2020-10-24]. Dostupné z: https://www.jeremyjordan.me/autoencoders/

[17] Fatima Ezzahra Jarmouni: Train Neural Networks Using a Genetic Algorithm in Python with PyGAD. [online], September 25, 2020, [cit. 2020-10-24]. Dostupné z: https://heartbeat.fritz.ai/train-neural-networks-using-a-genetic-algorithm-in-python-with-pygad-862905048429

[18] Jason Brownlee: Tour of Data Preparation Techniques for Machine Learning. [online], June 19, 2020, [cit. 2020-10-24]. Dostupné z: https://machinelearningmastery.com/data-preparation-techniques-for-machine-learning/

[19] Clare Liu: Data Transformation: Standardization vs Normalization. [online], April, 2020, [cit. 2020-10-24]. Dostupné z: https://www.kdnuggets.com/2020/04/data-transformation-standardization-normalization.html

Příloha **A**

Seznam použitých zkratek

ERP Enterprise resource planning

CRM Customer relationship management

BI Business intelligence

IoT Internet of things

MLP Multilayer perceptron

ReLU Rectified linear unit

PCA Principal component analysis

LDA Linear discriminant analysis

NAI Nexus artificial inteligence

GUI Graphical User Interface

Příloha B

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `DP_Jelinek_Tomas_2020.pdf` ..... text práce ve formátu PDF