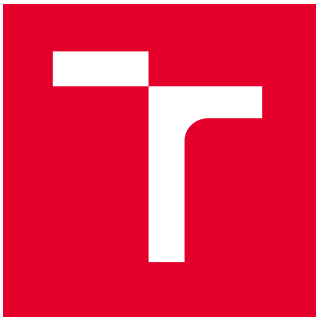


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

NÁSTROJ PRO OPTIMALIZACI STRUKTURY NEURONOVÝCH SÍTÍ

NEURAL NETWORK STRUCTURE OPTIMIZATION TOOL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. Daniel Štark**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. Eva Holasová**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Informační bezpečnost**

Ústav telekomunikací

Student: Bc. Daniel Štark ***ID:*** 211814

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Nástroj pro optimalizaci struktury neuronových sítí

POKYNY PRO VYPRACOVÁNÍ:

Cílem diplomové práce je vytvořit nástroj, který bude provádět návrh a optimalizaci struktury neuronové sítě z pohledu jejích jednotlivých parametrů (např. počtu vrstev, typu vrstev, aktivačních funkcí, počtu neuronových vrstev atd.). Výběr vhodné struktury bude prováděn na základě získaných a porovnaných metrik. V rámci práce bude navržen, vytvořen a následně otestován systém pro automatický návrh struktury neuronové sítě a jejich parametrů. Na základě tohoto systému bude možné provést modifikaci vybrané struktury pro dosažení nejlepšího výsledku. Struktura by měla být testována na omezeném počtu epoch. Efektivita výsledného řešení bude následně ověřena na veřejně dostupných datových sadách obsahující anomálie a průmyslové protokoly, jejichž klasifikace bude v rámci neuronové sítě prováděna a testována/srovnávána. Toto testování/srovnání bude provedeno pomocí odlišných metod zpracování dat.

DOPORUČENÁ LITERATURA:

[1] ABDEL-NASSER SHARKAWY. Principle of Neural Network and Its Main Types: Review [online]. 2020, 7, 8-19. ISSN 2409-5761. Doi:10.15377/2409-5761.2020.07.2

[2] KNAPP, Eric D. a Joel Thomas LANGILL. Industrial Network Security [online]. 2015. Doi:10.1016/C2013-0-06836-3

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 19.5.2023

Vedoucí práce: Ing. Eva Holasová

doc. Ing. Jan Hajný, Ph.D.
předseda rady studijního programu

ABSTRAKT

Tato diplomová práce se zabývá optimalizací struktur umělých a konvolučních neuronových sítí. V teoretické části práce jsou mimo jiné popsány hyperparametry, které tyto struktury tvoří. Dále jsou popsány metriky, pomocí kterých lze struktury ohodnotit. Praktickým výstupem práce je nástroj, který na základě uživatelského nastavení dokáže pro daný dataset automaticky vygenerovat struktury neuronových sítí, otestovat je, a pro ty nejlepší z nich vypsat přehlednou zprávu. Nástroj je naimplementován v jazyce Python, s využitím knihoven TensorFlow a Keras. Součástí praktické části práce je kromě podrobného popisu zdrojového kódu nástroje také jeho testování na dobře známých ukázkových datasetech a na datasetu vyjadřujícím provoz v průmyslové síti během probíhajících kybernetických útoků.

KLÍČOVÁ SLOVA

strojové učení, klasifikace, neuronová síť, optimalizace struktury, Python, TensorFlow, Keras

ABSTRACT

This thesis deals with optimizing the structures of artificial and convolutional neural networks. The hyperparameters, from which these structures are comprised of, are described in the theoretical part of this thesis. In addition, it explains the metrics used for evaluation of these structures. The practical outcome of this thesis is a tool capable of automatically generating neural network structures for a given dataset based on user-defined configuration. The tool also automatically tests the generated structures and creates reports which summarize the performace of the best generated structures. The tool is implemented using Python language, with utilization of TensorFlow and Keras libraries. In addition to providing a detailed source code description, the practical part of the thesis includes testing the tool on well-known datasets, as well as a dataset simulating traffic of an industrial network under ongoing cyber attack.

KEYWORDS

machine learning, classification, neural network, structure optimization, Python, TensorFlow, Keras

ŠTARK, Daniel. *Nástroj pro optimalizaci struktury neuronových sítí*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2023, 77 s. Diplomová práce. Vedoucí práce: Ing. Eva Holasová

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Daniel Štark
VUT ID autora:	211814
Typ práce:	Diplomová práce
Akademický rok:	2022/23
Téma závěrečné práce:	Nástroj pro optimalizaci struktury neuronových sítí

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Velice děkuji vedoucí diplomové práce paní Ing. Evě Holasové za její odborné vedení, přínosné konzultace a pozitivní rozpoložení. Velmi si vážím času, který strávila pomáháním mi s touto prací.

Obsah

Seznam obrázků

Seznam výpisů

Úvod

Strojové učení je oblastí umělé inteligence, která se zabývá způsoby, jak pomocí počítačů řešit problémy, které se dají jen velmi obtížně rozložit na jednotlivé, počítačem zpracovatelné instrukce. Řešení těchto problémů je paradoxně pro člověka mnohdy zcela automatické a intuitivní. Mezi jednu z technik strojového učení se řadí neuronové sítě, které jsou hlavním tématem této práce. Neuronové sítě jsou v současnosti často skloňovaným tématem jak v průmyslu, tak i akademické sféře. I běžný člověk se s nimi setkává na denní bázi – doporučují mu příspěvky na sociálních sítích, překládají pro něj texty mezi světovými jazyky, jsou i součástí algoritmů, které jeho smartphone využívá k rozpoznání jeho obličeje.

Jak již napovídá název, neuronové sítě jsou způsobem reprezentace informace, který je inspirován pochody v lidských mozcích. Základními jednotkami sítě jsou neurony. Informace jsou v rámci neuronové sítě distribuovaně rozloženy do sil propojení mezi jednotlivými neurony. Způsob, jakým jsou neurony propojeny, kolik jich je a jaký význam mají jednotlivá propojení, je definován pomocí tzv. hyperparametrů neuronové sítě, neboli její struktury. Zvolení vhodné struktury neuronové sítě pro daný typ problému velmi ovlivňuje schopnost sítě tento problém řešit. Vhodná struktura je typicky hledána člověkem pomocí aplikování jeho znalostí a zkušeností s určitou dávkou heuristiky. Cílem této práce je navrhnout a vytvořit nástroj, který toto hledání alespoň urychlí. Tento nástroj by měl jeho uživateli poskytnout možnost své zkušenosti s problematikou využít, zároveň by však měl alespoň částečně pracovat s jeho možnou zaujatostí a vyzkoušet tedy i ty struktury, které by uživatel mohl předem zavrhnout. Čím více však bude nástroj kompenzovat zaujatost jeho uživatele, tím více možných struktur bude muset být vytvořeno a otestováno, a tím se zvýší i doba běhu nástroje. Nástroj tedy musí také dát uživateli možnost ovlivnit míru heuristiky, se kterou bude pracovat, a tím i ovlivnit výslednou dobu běhu.

Práce je rozdělena na tři kapitoly – kapitola 1 se zabývá teoretickými poznatky z oblasti strojového učení, které jsou samozřejmě platné i pro neuronové sítě. Dále stručně představuje problematiku datasetů a metrik. Kapitola 2 se zabývá přímo neuronovými sítěmi a jejich principem. Představeny jsou neurony, vrstvy, druhy neuronových sítí, aktivační funkce a samotné trénování. Kapitola 3 pak popisuje praktický výstup této práce. Popsán je samotný problém hledání architektur neuronových sítí, poté je představen návrh programu, který tento problém má řešit. Následně jsou stručně popsány datasety, které byly použity při vytváření a testování nástroje. Nejdůležitější a nejobsáhlejší sekcí kapitoly 3 je pak popis samotné implementace nástroje obsahující četné výpisy zdrojového kódu. Poslední částí práce je komentář k výsledkům testování nástroje na zmíněných datasetech.

1 Strojové učení

V této kapitole jsou nejprve popsány vybrané poznatky z oblasti strojového učení, které je nutné zmínit pro pochopení problematiky neuronových sítí. Strojové učení je jeden ze základních principů umělé inteligence. Účelem strojového učení je tedy umožnit počítačům najít řešení i těch problémů, které se dají jen těžko řešit pomocí přesně daných neměnných instrukcí. Příkladem problému, který je velmi obtížné řešit algoritmicky, může být například rozpoznávání obrazů nebo řeči.

Algoritmus strojového učení je speciální druh programu, který se dokáže učit z jeho vstupních trénovacích dat a na základě nich vrátit další algoritmus nebo funkci. Tento výstup se označuje jako model. Pomocí modelu je možné aplikovat zkušenosti, získané během učení ze vstupních dat, při zpracovávání dat jiných. Mezi běžné techniky strojového učení se řadí například lineární regrese, k-nejbližších sousedů, rozhodovací stromy a samozřejmě neuronové sítě [1].

1.1 Metody strojového učení

V závislosti na charakteru vstupních dat a samotném učícím procesu se v rámci strojového učení uvažují tři různé přístupy [1]:

- učení s učitelem,
- učení bez učitele,
- zpětnovazební učení.

Učení s učitelem

V rámci metody zvané učení s učitelem představuje vstupní data množina uspořádaných dvojic:

$$(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n),$$

kde $x_i \in X$ (prostoru vstupů) a $y_i \in Y$ (prostoru výstupů). Tato množina uspořádaných dvojic se nazývá (trénovací) dataset. Algoritmus strojového učení hledá takovou funkci (model) f, která přiřazuje prvkům z prostoru X prvky z prostoru Y. Tato hledaná funkce je aproximací neznámé funkce, která přesně charakterizuje daný problém ve skutečnosti. Typicky jsou vstupy x_i tzv. feature[1] vektory[2] vstupní entity, zatímco y_i popisují požadovaný výstup, například přiřazení správných labelů dané entitě nebo její zařazení do třídy [1, 2].

[1] Features (příznaky) lze chápat jako numericky vyjádřené vlastnosti dané vstupní entity.

[2] Běžně se lze setkat i s jiným tvarem vstupních dat, například trojdimenzionálními tenzory pro barevné obrázky.

Mezi typické problémy řešené pomocí učení s učitelem se řadí například [1]:

- klasifikace: v rámci klasifikačních problémů je prostor výstupů Y tvořen množinou obsahující k kategorií. Algoritmus strojového učení má za úkol nalézt funkci f, která přiřadí vstupu (typicky feature vektoru) x jednu nebo více z daných k kategorií.
- Regrese: problémem regrese rozumíme úlohu, ve které má nalezená funkce f předpovědět spojitou proměnnou y závislou na vstupu x.
- Predikce struktury: v některých úlohách je výstup y značně složitější než vektor labelů vyjadřující příslušnost entity k vybraným třídám (kategoriím) nebo reálné číslo v případu regrese. Výstupem y může být i například sekvence slov v rámci překladače.

Nevýhodou učení s učitelem je nutnost existence onoho učitele, to znamená člověka, který označí vstupy x_i správným výstupem y_i a tím připraví dataset. Aplikovatelnost učení s učitelem je tedy značně omezena dostupností datasetů pro daný druh problému. Tato práce se bude zaměřovat výhradně na řešení problému klasifikace pomocí učení s učitelem.

Učení bez učitele

Vstupní data při učení bez učitele jsou pouze feature vektory (popřípadě jiný tvar dat) x_i, které nejsou označeny zamýšleným výstupem y_i. Strojové učení bez učitele se používá k nalezení skrytých pravidelností mezi vstupními daty. Úlohy, které se pomocí učení bez učitele dají řešit, jsou například clustering (roztřídění dat do skupin na základě podobností) nebo dimenzionální redukce (vyjádření stejných dat pomocí feature vektorů menších rozměrů). Velkou výhodou tohoto přístupu oproti učení s učitelem je výrazně větší dostupnost vstupních dat, nemusí totiž prvně být označena. Jistým kompromisem mezi učením s učitelem a bez učitele je tzv. kombinované učení, které pracuje jak s označenými, tak neoznačenými daty [1].

Zpětnovazební učení

Modely zpětnovazebního učení pracují s tzv. agentem, který má za úkol vybrat co nejlepší posloupnost akcí na základě stavu okolí. Prostřednictvím akcí agent ovlivňuje nadcházející stavy svého okolí. Zpětnovazební učení pracuje s omezeným a zpožděným feedbackem, na rozdíl od učení s učitelem, kde po každém vyhodnocení vstupu x_i je algoritmus hned informován o tom, jaký je odpovídající správný výstup y_i. Algoritmus zpětnovazebního učení musí nejprve učinit několik vyhodnocení, až poté se dozví jejich správnost (vyjádřenou pomocí tzv. reward). Tímto způsobem se algoritmus učí jaká sekvence rozhodnutí v jakých situacích přinese co největší reward. Hlavními vlastnostmi zpětnovazebního učení jsou tedy přístup pokus – omyl,

zpoždění zpětné vazby a rozhodování se na základě celé situace, ne pouze dílčích podproblémů. Proces zpětnovazebního učení se dá obecně popsat pomocí následujících opakujících se kroků:

1. agent zjistí stav okolí,
2. agent vybere akci na základě stavu okolí a svých získaných zkušeností (tzv. policy),
3. okolí přejde do dalšího stavu ohodnoceného pomocí reward,
4. agent na základě zjištěné reward upraví svou policy.

Zpětnovazební učení se obecně využívá v rozhodovacích problémech, ve kterých je důležité sledovat změny v okolí a reagovat na ně, například v ovládání robotů v prostoru nebo hraní her jako šachy, Go a podobně [1, 3, 4].

V tabulce 1.1 jsou přehledně porovnány tři výše popsané základní metody strojového učení.

Tab. 1.1: Srovnání jednotlivých základních přístupů strojového učení.

Metody strojového učení		
Učení s učitelem	Učení bez učitele	Zpětnovazební učení
Vstupní data jsou označena očekávaným správným výstupem	Vstupní data nejsou označena	Vstupní data pouze vymezují možnosti agenta a jeho okolí
Řešení problémů regrese a klasifikace	Řešení problémů clusteringu, redukce dimenzí apod.	Řešení rozhodovacích problémů, kde je nutná opakovaná interakce s okolím
Hledání závislostí mezi vstupními features a požadovanými výstupy	Hledání struktur ve vstupních datech	Hledání sekvencí akcí, které pro daný stav okolí zajistí největší reward

1.2 Učení s učitelem

Jak již bylo zmíněno v předešlém textu, učení s učitelem se zaměřením na klasifikaci je hlavním paradigmatem strojového učení této práce. Tato sekce navazuje na předešlý text a rozšiřuje ho o další detaily.

1.2.1 Formální definice

Definice 1 *Algoritmus učení s učitelem je počítačový program, který hledá funkci f z prostoru funkcí F, která minimalizuje ztrátovou funkci l nad trénovacím datasetem D [1]:*

$$\min_{f \in F} \frac{1}{|D|} \sum_{(x,y) \in D} l(f(x), y), \tag{1.1}$$

kde $|D|$ je počet párů (x, y) v trénovacím datasetu D, F je prostor obsahující funkce, které zobrazují prostor vstupů X na prostor výstupů Y, a l je ztrátová (loss) funkce vyjadřující rozdíly mezi predikcemi funkce f a labely y [1].

Prostor F definuje jaký druh funkce (modelu) se má natrénovat na datasetu D. Příklady druhů těchto funkcí mohou být například lineární funkce, rozhodovací stromy nebo neuronové sítě. Správně zvolený druh funkcí musí být dostatečně komplexní na to, aby dokázal přesně aproximovat hledanou funkci charakterizující řešený reálný problém. Pokud je k řešení náročného a značně nelineárního problému zvolen málo komplexní druh modelu, bude docházet k tzv. underfittingu, to znamená, že model nebude schopen se naučit všechny skryté vztahy mezi vstupy a výstupy. Naopak pokud bude zvolený druh modelu příliš komplexní pro daný problém, bude docházet k tzv. overfittingu. Při overfittingu se výsledný model příliš přizpůsobí vzorcům v trénovacím datasetu a ztratí schopnost generalizace, tedy schopnost správně vyhodnocovat i předem neviděná data. K ovládání komplexity modelu slouží techniky tzv. regularizace [1].

Ztrátová funkce l porovnává predikce modelu $f(x)$ s učitelem označeným labelem (očekávaným výstupem) y. Výstup l přímo vyjadřuje správnost predikce. Příkladem ztrátové funkce může být například MSE (Mean Squared Error) [1]. Čím nižší je výstup ztrátové funkce l, tím přesněji funkce f aproximuje hledanou neznámou funkci, která charakterizuje daný reálný problém. Samotný proces učení lze tedy chápat jako hledání parametrů modelu, pro které bude ztrátová funkce l minimální.

1.2.2 Dataset a validace

Kvalita i kvantita použitého datasetu D, na kterém se model natrénuje, má velký vliv na praktickou využitelnost tohoto modelu. Mezi vlastnosti popisující vhodnost datasetu pro trénování patří například:

- velikost – počet párů (x, y),
- přesnost – správnost labelů y pro dané vstupy x,
- vyváženost – možné klasifikační třídy jsou zastoupeny rovnoměrně,
- čistota dat – obsah duplikátů, anomálií, chybějících příznaků...
- stejné měřítko příznaků (normalizovanost).

Aby bylo možné ověřit míru správnosti predikcí daného modelu v průběhu učení a taktéž po dokončení učení, jsou data z datasetu typicky rozdělena na data trénovací, validační a testovací. Trénovací data jsou použita během samotného učení, tedy k hledání takových parametrů modelu, pro které je ztrátová funkce l minimální. Validační data se využívají k posouzení predikčních schopností modelu na neviděných datech během trénování. Z predikcí na validačních datech se taktéž vypočítává ztrátová funkce l[3]. Velké rozdíly mezi hodnotami ztrátové funkce l pro trénovací a validační data znamená, že dochází k přetrénování a model je pro danou úlohu příliš komplexní. Testovací data pak slouží k posouzení predikčních schopností modelu po dokončení trénování. Na základě predikcí na testovacích datech jsou vypočteny metriky ohodnocující kvalitu výsledného modelu [1].

Existuje několik způsobů, jak rozdělit dataset na trénovací a validační data a provádět následnou validaci během trénování, mezi základní patří například [5]:

- testování externí validací (tzv. holdout) – v rámci této metody je dataset náhodně rozdělen na trénovací a validační podmnožinu v předem definovaném poměru. Model je natrénován na trénovacích datech a pak je správnost jeho výsledků ověřena na datech validačních. Nevýhodou tohoto přístupu je, že velká část dat nebude použita při trénování, dosahované výsledky modelu tedy mohou být ovlivněny tím, jak byl dataset rozdělen.
- k-násobná křížová validace (k-fold cross-validation) – dataset je rozdělen do k podmnožin, poté je vždy jedna podmnožina použita k validaci a zbylých $k-1$ k trénování. Tento proces se opakuje k-krát a pokaždé je zvolena jiná kombinace podmnožin, aby se každá podmnožina použila právě jednou k validaci a $(k-1)$-krát k trénování. Výsledná efektivnost modelu je pak průměrem efektivností všech z k iterací. Díky tomuto přístupu jsou všechna data modelu použita jak ke trénování, tak k validaci. Nevýhodou je vyšší časová náročnost.
- leave-one-out cross-validation – jedná se o speciální případ křížové validace, kdy $k = |D|$. V jedné iteraci se tedy pro validaci použije pouze jeden pár (x, y).

1.2.3 Metriky

V rámci učení s učitelem existuje několik metrik popisujících kvalitu modelem předpovězených predikcí a tím i kvalitu modelu jako takového. Pro modely určené ke klasifikaci se používají jiné metriky, než pro modely určené k výpočtu regrese. V následujícím textu jsou uvedeny ty nejpoužívanější metriky pro oba typy úloh.

[3]Na základě hodnoty ztrátové funkce l pro validační data však nejsou upravovány parametry modelu.

Metriky pro regresi

MSE, která se využívá i jako ztrátová funkce l, je rovněž standardní metrikou pro modely určené k řešení problému regrese. Pro N predikcí y' a k nim příslušným skutečným hodnotám y je MSE definována jako:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - y_i')^2. \tag{1.2}$$

Mezi vlastnosti MSE patří relativně velká citlivost na chyby (odchylky se značně projevují na výsledku), způsobená umocňováním rozdílů mezi y a y'. Funkce MSE je rovněž diferencovatelná, může tedy poskytovat i informace o tom, jestli jsou predikce modelu y' větší nebo menší než skutečné hodnoty y [6, 7].

Metrika MAE (Mean Absolute Error) je velice podobná MSE, rozdíly mezi predikcí y' a skutečnosti y však nejsou umocňovány na druhou:

$$MAE = \frac{1}{N}\sum_{i=1}^{N}(y_i - y_i'). \tag{1.3}$$

Výhodou MAE může být snazší reprezentativnost chyby pro lidského pozorovatele oproti MSE (výsledek přímo vyjadřuje o kolik se průměrně liší predikce od skutečnosti, jednotky jsou zachovány). Další dobrou vlastností MAE je její robustnost, jednotlivé odchylky se neprojevují tak významně na výsledné hodnotě. Nevýhodou této funkce je však její nediferencovatelnost [6, 7].

Další velmi často používaná metrika je RMSE (Root Mean Squared Error), která je definovaná jako odmocnina z MSE:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - y_i')^2}. \tag{1.4}$$

RMSE je diferencovatelná a poměrně odolná proti odchylkám. Rovněž poskytuje výsledky ve stejné jednotce jako ve které jsou vyjádřeny predikce modelu y' a skutečné hodnoty y, výsledky RMSE jsou tedy snadno interpretovatelné. Tyto vlastnosti dělají z RMSE dobrý kompromis mezi dvěma výše zmíněnými metrikami [6, 7].

Metriky pro klasifikaci

Základní metrikou pro modely určené ke klasifikaci je tzv. celková správnost (accuracy). Accuracy je definována jako poměr počtu správných predikcí a celkového počtu predikcí:

$$accuracy = \frac{N_{\text{správných}}}{N_{\text{celkem}}}. \tag{1.5}$$

Schopnost objektivně ohodnotit model této metriky však prudce klesá s nevyvážeností datasetu. Pokud se zastoupení jednotlivých labelů v datasetu významně liší, celková správnost podává velmi zkreslené výsledky [8].

Tab. 1.2: Matice záměn (confusion matrix).

		Predikce	
		+	−
Realita	+	TP	FN
	−	FP	TN

Pro binární klasifikaci (rozdělení do dvou vzájemně se vylučujících skupin) existují další metriky odvozené z tzv. matice záměn (confusion matrix). Matice záměn[4] rozlišuje 4 druhy predikcí [8]:

- TP (True Positive) – model správně předpověděl pozitivní třídu,
- TN (True Negative) – model správně předpověděl negativní třídu,
- FP (False Positive) – model předpověděl pozitivní třídu, když měl předpovědět negativní,
- FN (False Negative) – model předpověděl negativní třídu, když měl předpovědět pozitivní.

Pro větší názornost je matice záměn vyobrazena v tab. 1.2. V následujícím textu budou dvoupísmenné zkratky druhů predikcí vyjadřovat i počet predikcí daného druhu. Pro binární klasifikaci můžeme pomocí matice záměn celkovou správnost (accuracy) také definovat jako [8]:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \tag{1.6}$$

Další metrikou pro binární klasifikaci, která je odvozená z matice záměn, je tzv. přesnost (precision). Precision je definována vztahem:

$$precision = \frac{TP}{TP + FP}. \tag{1.7}$$

Tato metrika vyjadřuje percentil kolik predikcí pozitivní třídy bylo ve skutečnosti správných. Například pokud model s precision rovnou 0,8 označí email jako spam, tak se na 80 % jedná o spam [9].

Úplnost (recall) je další metrikou vyjadřující kvalitu binárního klasifikátoru. Je definována jako:

$$recall = \frac{TP}{TP + FN}. \tag{1.8}$$

Recall vyjadřuje percentil kolik případů pozitivní třídy bylo modelem správně predikováno. Například model s recallem rovným 0,7 dokáže 70 % spamu správně označit jako spam. Typicky platí, že větší recall je kompenzován nižší precision a naopak.

[4]Matici záměn lze zobecnit i pro n tříd.

Dobře natrénovaný model tedy musí najít rovnováhu mezi těmito dvěma vzájemně soupeřícími metrikami [9].

Další metrikou pro evaluaci klasifikace je tzv. F1 skóre. F1 skóre vychází z přesnosti a úplnosti, jedná se o harmonický průměr těchto dvou metrik:

$$F1 = \frac{2}{\frac{1}{precision} + \frac{1}{recall}}. \tag{1.9}$$

Vysoké F1 skóre vyjadřuje, že jak přesnost, tak úplnost jsou relativně vysoké [9].

Jak již bylo zmíněno, výše popsané metriky lze zobecnit i na klasifikační problém o více třídách. V takovém případě se přesnost, úplnost a F1 skóre počítají pro každou třídu zvlášť. Celkovou přesnost, úplnost a F1 skóre lze pak vypočítat zprůměrováním.

2 Neuronové sítě

Tato kapitola se zabývá neuronovými sítěmi, technikou strojového učení, na kterou je zaměřena tato práce. Jsou zde popsány základní myšlenky, na bázi kterých neuronové sítě pracují. Dále jsou zde představeny neurony, základní stavební bloky neuronových sítí, a jejich struktura. Rovněž jsou představeny základní druhy neuronových sítí podle jejich architektury. Také je věnována pozornost aktivačním funkcím a jejím druhům. Na konci kapitoly je popsán algoritmus gradient descent, který slouží k natrénování neuronových sítí.

Umělou neuronovou sítí lze rozumět výpočetní model inspirovaný pochody v lidských mozcích během řešení problémů. Tyto pochody se dají velice zjednodušeně popsat jako vzájemnou výměnu informací mezi jednotlivými neurony pomocí synapsí, tedy propojení mezi neurony [10].

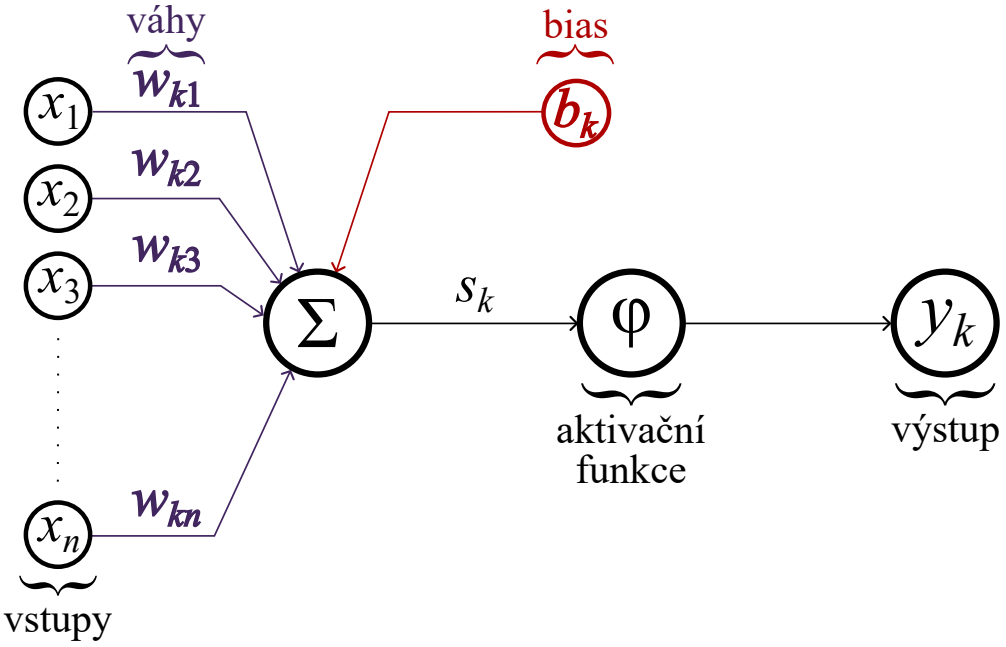
Tato propojení jsou charakterizována svojí silou (neboli vahou), která popisuje jak moc informace uložená v neuronu na začátku synapse ovlivňuje informaci uloženou v neuronu na jejím konci. Váhy jednotlivých propojení lidský mozek zjišťuje a upravuje na základě již zažitých zkušeností. Dá se tedy říct, že ve vahách propojení mezi neurony, které jsou spjaty s řešením konkrétního problému, jsou distribuovaně uloženy informace, které se mozek naučil při řešení úkolu podobného. Na základě těchto distribuovaně uložených informací dokáže mozek vyřešit problémy efektivně a správně [11].

Tato výměna informací probíhá mezi mnoha neurony současně, je tedy silně paralelizována. To umožňuje mozkům vyřešit i poměrně náročné problémy, jako rozpoznávání obrazu (hledání známého prvku na neznámé scéně), v řádech 100–200 milisekund [11]. Na základě těchto poznatků lze umělou neuronovou síť definovat jako:

Definice 2 *Umělá neuronová síť je masivně paralelizovaný, distribuovaný procesor, složený z jednoduchých výpočetních jednotek přirozeně uzpůsobených k ukládaní a aplikování na základě zkušeností získaných informací [12].*

2.1 Neuron

Jednoduché výpočetní jednotky, ze kterých je umělá neuronová síť složena, se nazývají umělé neurony. Základní struktura umělého neuronu je zobrazena na obr. 2.1. V následujícím textu budou vysvětleny všechny vyobrazené části této struktury.



Obr. 2.1: Struktura neuronu. Inspirováno [11].

Vstupy neuronu

Vektor proměnných $x_1, x_2, \ldots, x_n$ představuje vstupy neuronu k. Tyto vstupy mohou být přímo vhodně vyjádřené features vstupních dat v případě, že neuron k se nachází ve vstupní vrstvě (vrstvy budou představeny v dalších podkapitolách), nebo výstupy jiných neuronů.

Váhy

Vstupy x_n vstupují do neuronu k pomocí spojení (synapsí), které jsou charakterizovány synaptickými váhami:

$$w_{k1}, w_{k2}, \ldots, w_{kn} \in \mathbb{R}. \tag{2.1}$$

Synaptické váhy rovněž tvoří vektor o velikosti n. Váhové koeficienty w_{kn} jsou parametry neuronové sítě, jejichž ideální hodnoty jsou hledány v rámci trénování sítě. Platí, že každý vstup x_n neuronu k je vynásoben hodnotou jemu příslušné váhy w_{kn}. Všechny násobky jsou následně sečteny. Váhy v umělé neuronové síti dosahují reálných hodnot, tedy jak kladných, tak i záporných, na rozdíl od lidských mozků, kde spojení jsou buďto kladná nebo neexistující (nulová) [11].

Bias

Bias b_k je dalším reálným parametrem neuronové sítě, jehož ideální hodnota je hledána během trénování sítě. Princip fungování biasu je podobně jako váhy inspirován lidským mozkem, kde se lze setkat s tzv. prahy. Práh je atribut neuronu, který přímo ovlivňuje, jak snadno/obtížně je daný neuron možné aktivovat. Čím větší je práh, tím větší musí být součet všech vstupů do neuronu, aby došlo k jeho aktivaci.

Bias b_k lze chápat jako obdobu prahu, která je přičítána k součtu všech vstupů x_n vynásobených příslušnými vahami w_{kn}. Bias je tedy na vstupech nezávislý parametr, který může výrazně ovlivnit hodnotu S_k, která vstupuje do aktivační funkce ϕ. Jedná se tedy o konstantní vstup, charakteristický pro neuron k, jehož aplikaci lze vnímat jako afinní transformaci výsledku sumy. Pro vstup S_k aktivační funkce ϕ, tzv. vnitřního potenciálu neuronu, platí [11]:

$$S_k = \sum_{i=1}^{n} w_{ki}\, x_i + b_k. \tag{2.2}$$

Pro zjednodušení tohoto vztahu se dá bias b_k nahradit novým vstupem x_0 a synaptickou váhou w_{k0} na nultých pozicích příslušných vektorů, pro které platí:

$$x_0 = 1, \tag{2.3}$$

$$w_{k0} = b_k. \tag{2.4}$$

Zjednodušený vztah pro výpočet vnitřního potenciálu neuronu S_k tedy bude [13]:

$$S_k = \sum_{i=0}^{n} w_{ki}\, x_i. \tag{2.5}$$

Aktivační funkce

Aktivační funkce ϕ neuronu k definuje způsob, jakým je vnitřní potenciál neuronu S_k převeden na samotný výstup neuronu y_k. Vnitřní potenciál S_k může teoreticky nabývat jakékoliv reálné hodnoty, hodnota výstupu y_k je však typicky omezena na předem určený interval. Zvolením vhodné aktivační funkce lze zajistit nelinearitu neuronové sítě, což je klíčová vlastnost potřebná pro řešení komplexnějších úkolů. Dále je velice důležité, aby aktivační funkce byla diferencovatelná, neboť při trénování neuronové sítě dochází k derivování dané funkce [14]. Konkrétní druhy používaných aktivačních funkcí jsou představeny v sekci 2.3.

Shrnutí

Umělý neuron je základní prvek neuronové sítě, na který lze pohlížet jako na funkci většího počtu proměnných. Princip fungování neuronu k lze tedy shrnout rovnicí:

$$y_k = \phi\left(\sum_{i=1}^{n} w_{ki}\, x_i + b_k\right), \tag{2.6}$$

kde y_k je výstup neuronu k, ϕ je aktivační funkce, w_{ki} je synaptická váha spojená se vstupem x_i a b_k je bias neuronu k. V případě nahrazení biasu nultým vstupem a nultou váhou, jak bylo ukázáno ve vztahu 2.5, lze tento vztah zjednodušit na:

$$y_k = \phi\left(\sum_{i=0}^{n} w_{ki}\, x_i\right). \tag{2.7}$$

Neuron nejprve aplikuje své natrénované zkušenosti obsažené ve vahách a biasu na všechny své vstupy, tak vypočítá svůj vnitřní potenciál, který vstupuje do aktivační funkce. Výstupem neuronu je výstup této aktivační funkce.

2.2 Architektura neuronových sítí

Již samotné neurony dokáží aproximovat jednoduché funkce a tím řešit jednoduché problémy (např. binární operaci *AND*), avšak na aproximaci složitějších funkcí jsou potřeba celé sítě na sebe navazujících neuronů. V rámci neuronových sítí jsou umělé neurony uspořádány do jedné nebo více vrstev.

2.2.1 Dopředná neuronová síť

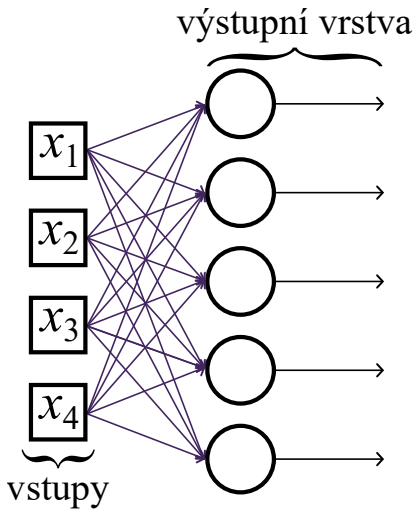
Dopřednou neuronovou sítí rozumíme síť, mezi jejímiž vrstvami (a tedy i neurony) neexistují žádné cykly. Platí, že vstup každého jednoho neuronu je nezávislý na svém vlastním výstupu. Informace uvnitř sítě se tedy šíří pouze jedním směrem (dopředu) [15].

Jednovrstvá dopředná síť

Jednovrstvá dopředná síť je nejjednodušší typ architektury neuronové sítě. Je složena z vrstvy vstupní a vrstvy výstupní. Ve vstupní vrstvě se však nenachází neurony, ale pouze vstupy, takže tam neprobíhají žádné matematické operace. Proto se tato architektura označuje jako jednovrstvá, i když teoreticky má vrstvy dvě. Tato jednoduchá architektura je vyobrazena na obr. 2.2. Uvedená jednovrstvá dopředná síť má 4 vstupy x_n a 5 výstupů, například by tedy mohla třídit vstupní entity do 5 skupin podle 4 parametrů. Všechny výstupy vstupní vrstvy jsou propojeny se všemi neurony výstupní vrstvy. Taková vrstva, jejíž všechny neurony jsou závislé na všech výstupech předešlé vrstvy, se nazývá hustě propojená [11].

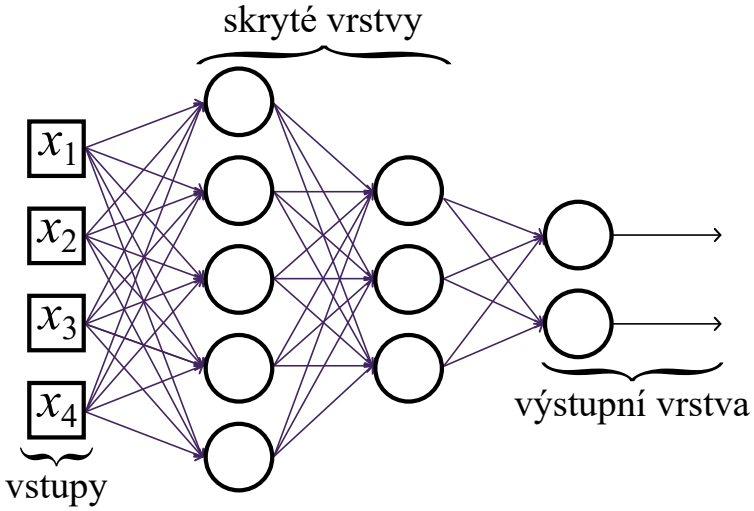
Vícevrstvá dopředná síť

Vícevrstvá dopředná síť se vyznačuje přítomností alespoň jedné tzv. skryté vrstvy. Skrytou vrstvou lze chápat takovou vrstvu, která není vstupní ani výstupní. Nazývá



Obr. 2.2: Jednovrstvá dopředná neuronová síť. Inspirováno [11].

se skrytá, protože přímo neinteraguje se vstupy nebo výstupy neuronové sítě, pracuje pouze s mezivýsledky. Obecně platí, že vícevrstvé sítě dokáží aproximovat složitější funkce než jejich jednovrstvé protějšky. Vícevrstvé dopředné sítě jsou nejběžnější architekturou neuronových sítí, používají se na široké spektrum úkolů. Na obr. 2.3 je jako příklad vícevrstvé dopředné sítě vyobrazena 4-5-3-2 síť [11, 15].



Obr. 2.3: Vícevrstvá dopředná neuronová síť. Inspirováno [15].

2.2.2 Konvoluční neuronové sítě

V rámci předešlého textu se uvažovaly neuronové sítě, které jsou složené pouze z tzv. plně propojených (dense) vrstev. To znamená, že každý jeden neuron je syna-

pticky spojen se všemi neurony předešlé vrstvy. Tento typ neuronových sítí se často označuje jako ANN (Artificial Neural Network). Sítě typu ANN dosahují dobrých výsledků při práci s textovými daty, avšak zpracovávání obrazových dat pomocí nich může být problematické, a to hlavně ze dvou důvodů [16]:

- počet neuronů ANN velmi rychle roste s rozměry obrazových vstupů, mnoho neuronů znamená mnoho parametrů, které je třeba optimalizovat,
- ANN nedokáží pracovat v kontextu celého prostoru obrazových vstupních dat, zaměřují se na jednotlivé pixely samostatně, ne nutně na obrazce. Jsou tedy velmi náchylné na přesnou polohu objektu, který například mají detekovat.

Zejména tyto problémy řeší tzv. konvoluční neuronové sítě, označované zkratkou CNN (Convolutional Neural Network). Konvoluční neuronové sítě jsou schopné v obrazovém vstupu detekovat určité příznaky (feature extraction) nezávisle na jejich poloze. Příznaky jsou extrahovány hierarchicky, například výstup první vrstvy může být množina hran, výstup druhé vrstvy pak geometrické obrazce, výstup třetí například už části hledaných objektů a tak dále. Pro sítě CNN je typické použití tzv. konvolučních a pooling vrstev, které se principem odlišují od plně propojených vrstev [17].

Konvoluční vrstva

V rámci konvoluční vrstvy jsou neurony synapticky spojeny pouze s vybranými neurony předešlé vrstvy, neurony jsou tedy zaměřené na nějakou konkrétní oblast. To znatelně snižuje počet vah a biasů a zrychluje konvergenci sítě k lokálnímu minimu při trénování. Počet vah může být dále redukován pomocí tzv. weight sharingu (jedna váha je sdílena skupinou spojení). Konvoluční vrstva aplikuje na vstupní data výpočetní operaci nazývanou konvoluce. Konvoluce probíhá postupným procházením vstupních dat pomocí filtrů (kernelů). Právě tyto filtry hledají ve vstupních datech příznaky. Výstup konvoluce, tzv. příznaková mapa, je vypočítána postupnou aplikací těchto filtrů na části vstupní matice o odpovídající velikosti [17].

Pooling vrstva

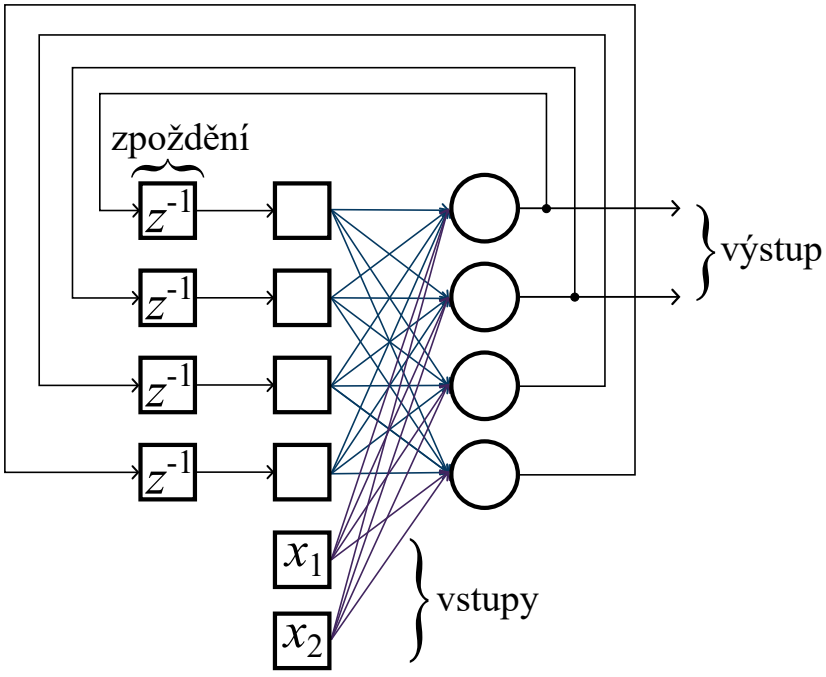
Pooling vrtvy slouží k redukci rozměrů příznakových map jejich podvzorkováním. To má za následek zvýšení odolnosti proti overfittingu a další snížení počtu parametrů sítě [18].

2.2.3 Rekurentní neuronová síť

Rekurentní neuronové sítě (RNN, Recurrent Neural Network) se od těch dopředných liší tím, že na alespoň jedné z jejich skrytých vrstev existuje zpětnovazební cyklus,

tedy synaptické spojení sama na sebe. Platí, že vstup neuronů napojených na zpětnovazební cyklus je závislý na předešlých výstupech těchto neuronů. Rekurentní sítě tedy oproti dopředným mají nějakou paměť, do které ukládají své minulé výpočty. Nazývají se rekurentní, protože postupně aplikují tu stejnou operaci (taktéž určenou vahami, biasy a aktivačními funkcemi) na každý jeden element vstupního vektoru. Díky zpětnovazební smyčce je výsledek této operace závislý jak na právě zpracovávaném elementu, tak i na elementech předchozích [19]. Příklad rekurentní neuronové sítě je vyobrazen na obr. 2.4 [11].

Rekurentní sítě se například využívají v případech, kdy počet vstupů a výstupů je proměnlivý pro různé páry vektorů vstupů a výstupů. Takový případ může být třeba překládání textu (to, že překlad desetiznakové věty A má 8 znaků neznamená, že překlad desetiznakové věty B bude mít taktéž 8 znaků). Nevýhodou rekurentních sítí je menší efektivita trénovacích algoritmů [15, 19].



Obr. 2.4: Rekurentní neuronová síť. Bloky z^{-1} značí zpoždění o jednu iteraci. Inspirováno [11].

2.3 Druhy aktivačních funkcí

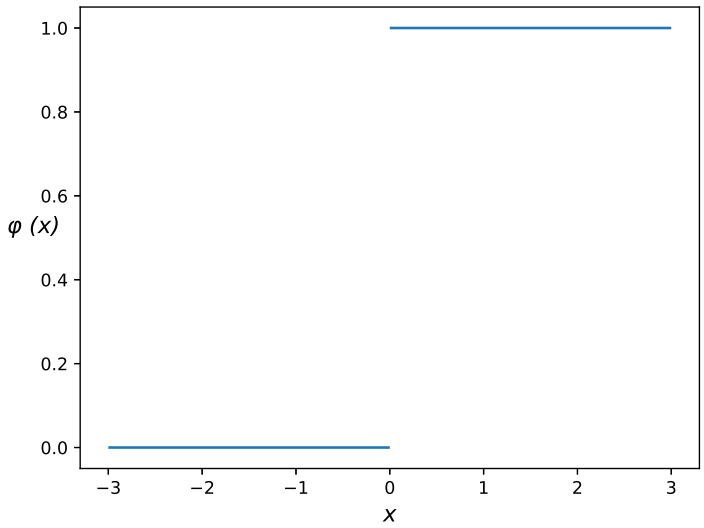
Jak bylo zmíněno v kapitole 2.1, aktivační funkce ϕ neuronu k slouží k výpočtu hodnoty výstupu neuronu y_k z jeho aktivačního potenciálu S_k. Výběr aktivačních funkcí

pro jednotlivé typy vrstev neuronové sítě je stěžejní pro schopnost dané sítě dosahovat přesných výsledku v rozumném čase. Platí, že v rámci jedné vrstvy všechny neurony využívají stejnou aktivační funkci. Aktivační funkce výstupní vrstvy závisí na požadovaném formátu výstupů. Ve vstupní vrstvě se nenachází neurony jako takové, o aktivačních funkcích pro tuto vrstvu tedy nemá smysl uvažovat. Aktivační funkce musí být diferencovatelné, aby bylo možné danou neuronovou síť trénovat algoritmem gradient descent. Tento algoritmus bude představen v sekci 2.4 [14]. V následujícím textu jsou uvedeny nejběžnější druhy aktivačních funkcí.

Skoková aktivační funkce

Skoková aktivační funkce je základním primitivním druhem aktivační funkce. Pokud je vstup funkce větší než nějaký stanovený práh t, výstupem bude hodnota y_1, jinak y_2. Tuto aktivační funkci využívají tzv. perceptrony, což jsou jednoduché neuronové sítě složené z jediného neuronu, vhodné na aproximaci pouze lineárně separabilních funkcí, například jíž zmíněného binárního *AND*. Graf skokové aktivační funkce je vyobrazen na obr. 2.5. Další nevýhodou skokové aktivační funkce je, že je nespojitá, nemá tedy derivaci [11]. Skokovou aktivační funkci lze definovat jako:

$$\phi(x) = \begin{cases} c_1, \text{pro } x \geq t, \\ c_2, \text{pro } x < t. \end{cases} \tag{2.8}$$



Obr. 2.5: Graf skokové aktivační funkce s $c_1 = 1$, $c_2 = 0$ a $t = 0$.

Lineární aktivační funkce

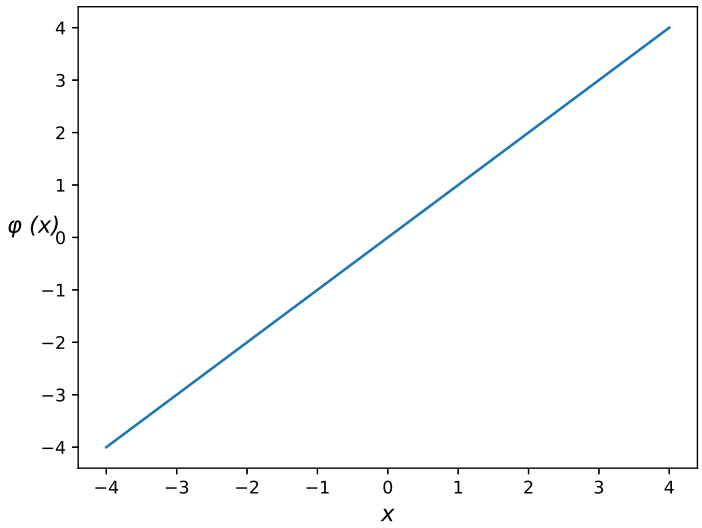
Výstup lineární aktivační funkce je přímo úměrný jejímu vstupu. Tato funkce není vhodná pro skryté vrstvy ze dvou různých důvodů:

- její derivace je konstantní, tedy nezávislá na vstupu. To opět znemožňuje použití algoritmu zpětné propagace.
- Všechny vrstvy neuronové sítě by se při použití lineární aktivační funkce sloučily do jedné, protože i při vyšším počtu skrytých vrstev by výstupy sítě stejně byly jen lineárně závislé na jejich vstupech. Vícevrstvá neuronová síť by tedy degradovala na jednovrstvou [20].

Lineární aktivační funkce však nachází využití ve výstupních vrstvách v podobě tzv. identity:

$$\phi(x) = x. \tag{2.9}$$

Výstupem neuronu využívající aktivační funkci identity je tedy přímo jeho aktivační potenciál. Tato aktivační funkce se využívá ve výstupní vrstvě neuronových sítí určených k výpočtu predikce numerických hodnot (regrese) [14].



Obr. 2.6: Graf identity, aktivační funkce používané ve výstupních vrstvách.

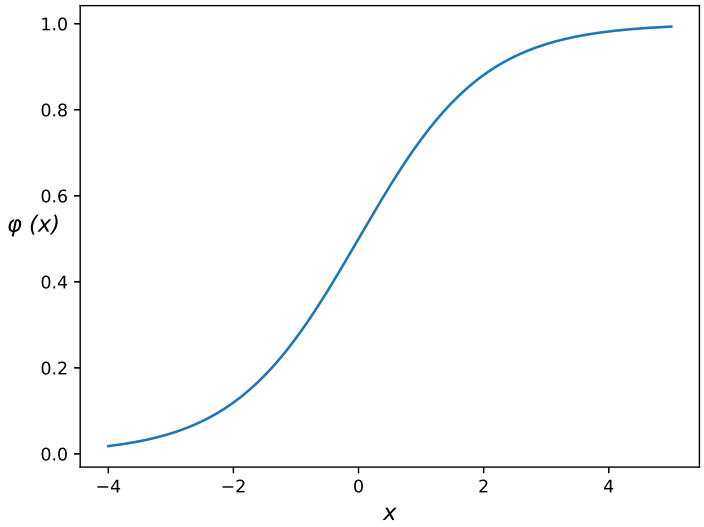
Sigmoida

Vstupem sigmoidní aktivační funkce může být jakékoliv reálné číslo, výstupem pak je hodnota v rozmezí (0, 1). Jedná se o rostoucí funkci, čím vyšší číslo je jejím

vstupem, tím více se bude výstup blížit k 1. Sigmoida je dána předpisem:

$$\phi(x) = \frac{1}{1 + e^{-x}}. \tag{2.10}$$

Graf této aktivační funkce je vyobrazen na obr. 2.7. Sigmoidu lze použít jak ve skrytých, tak výstupních vrstvách neuronových sítí, v případě skrytých vrstev vícevrstvých dopředných sítí však existují funkce, které obecně dosahují lepších výsledků. Problémem sigmoid v rámci skrytých vrstev je fakt, že jejich použití výrazně snižuje efektivitu trénovacího algoritmu pokud se v síti nachází větší množství skrytých vrstev (tzv. problém mizících gradientů). Sigmoidy se však uplatňují ve výstupních vrstvách sítí, které mají za úkol roztřídit vstupy buďto do dvou vzájemně se vylučujících skupin (binární klasifikace) nebo do několika vzájemně se nevylučujících skupin (vícelabelová klasifikace) [14].



Obr. 2.7: Graf sigmoidní aktivační funkce.

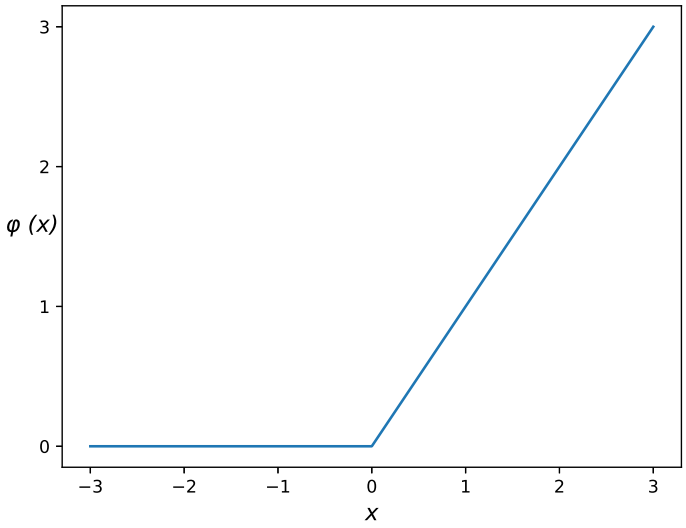
ReL

Funkce ReL (Rectified Linear Activation Function) je v současnosti standardně používanou aktivační funkcí skrytých vrstev dopředných neuronových sítí. Prvek, který tuto funkci aplikuje, se nazývá ReLU (Rectified Linear Unit), ovšem v literatuře se toto označení často používá i přímo pro danou funkci. Vstupem funkce ReL je opět jakékoliv reálné číslo x, výstupem je pak 0, pokud x je záporné, nebo samotné číslo x, pokud je nezáporné. Funkce ReL je tedy dána vztahem:

$$\phi(x) = \begin{cases} 0, \text{pro } x < 0, \\ x, \text{pro } x \geq 0, \text{tedy} \end{cases} \tag{2.11}$$

$$\phi(x) = max(0, x). \tag{2.12}$$

Graf funkce ReL je vyobrazen na obr. 2.8. Adaptace ReL jako aktivačních funkcí dopředných sítí okolo roku 2010 může být vnímána jako významný milník v oblasti strojového učení, protože od té doby je možné prakticky konstruovat i velmi hluboké dopředné neuronové sítě spjaté s termínem deep learning. To bylo se sigmoidami nebo hyperbolickým tangensem obtížné, právě kvůli již zmíněnému problému mizejících gradientů [21].



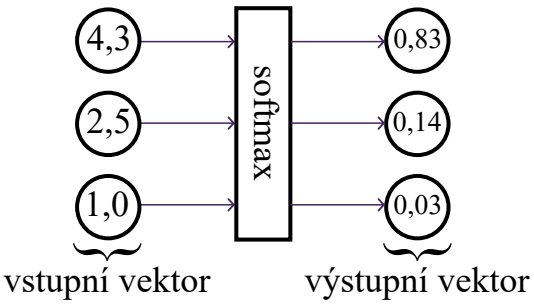
Obr. 2.8: Graf ReL, nyní nejpoužívanější aktivační funkce v dopředných sítích.

Softmax

Aktivační funkce softmax se využívá ve výstupních vrstvách neuronových sítí, které mají roztřídit vstupní entity do více vzájemně se vylučujících skupin (tzv. vícetřídní klasifikace). Vstupem funkce softmax je vektor několika reálných hodnot. Výstupem pak je stejně velký vektor pravděpodobností, jejichž součet je roven 1. Pravděpodobnosti jsou proporcionální relativním velikostem hodnot v rámci vstupního vektoru. Funkce softmax tedy převádí aktivační potenciály na pravděpodobnost, že konkrétní vstupní entita patří do příslušné skupiny. Funkce softmax je dána vztahem:

$$\phi_i(\vec{x}) = \frac{\mathrm{e}^{x_i}}{\sum_{j=1}^{K} \mathrm{e}^{x_j}}, \tag{2.13}$$

kde x_i je i-tý prvek vstupního vektoru $\vec{x}$ a $\phi_i(\vec{x})$ je pravděpodobnost příslušná x_i. Dále K je celkový počet vzájemně se vylučujících tříd [22]. Ukázka výpočtu funkce softmax se nachází na obr. 2.9.



Obr. 2.9: Ukázka výpočtu funkce softmax. Zaokrouhleno pro přehlednost.

Shrnutí aktivačních funkcí

Aktivační funkce jsou velmi důležitým prvkem neuronových sítí. Výběr aktivačních funkcí neuronů ve skrytých vrstvách přímo ovlivňuje schopnost se učit dané neuronové sítě. Volba aktivační funkce v rámci výstupní vrstvy je stěžejní pro to, aby neuronová síť mohla vůbec řešit konkrétní typ problému [14].

2.4 Trénování neuronové sítě a gradient descent

Trénováním neuronové sítě rozumíme problém nalezení co nejlepších hodnot pro váhy a biasy dané sítě. Jak již bylo zmíněno v sekci 1.2, kvalita hodnot vah a biasů je vyjádřena pomocí loss funkce (popřípadě cost funkce). Během trénování neuronové sítě jsou tedy hledány ty hodnoty vah a biasů, pro které je hodnota cost funkce minimální. Trénování neuronové sítě lze tedy chápat jako úlohu minimalizace funkce. Parametry modelu jsou na začátku učení inicializovány náhodně.

Zdaleka nejběžnějším algoritmem využívaným k trénování neuronových sítí je tzv. gradient descent. Gradient descent je metoda minimalizace cost funkce $C(\Phi)$ závislé na parametrech modelu Φ (tedy na vahách a biasech) spočívající v postupném měnění těchto parametrů ve směru opačném ku gradientu cost funkce $\nabla_\Phi C(\Phi)$ ve vztahu k těmto parametrům. Velikost jednotlivých změn v parametrech Φ po každém výpočtu gradientu je vyjádřena pomocí tzv. learning rate η. Dalším důležitým parametrem gradient descentu je počet tzv. epoch, epochu lze chápat jako jeden kompletní průchod celého trénovacího datasetu algoritmem učení. Podle množství dat z datasetu, které jsou použity pro výpočet gradientu před jediným krokem, lze gradient descent roztřídit na tři varianty [23]:

- batch gradient descent,
- stochastický gradient descent,
- mini-batch gradient descent.

Batch gradient descent

Batch gradient descent pracuje s gradienty pro kompletně celý dataset D. Pro jedinou změnu parametrů Φ je tedy potřeba vypočítat gradient cost funkce $C(\Phi)$ pro celý dataset. Platí tedy, že jedné epoše odpovídá právě jeden krok. Nové parametry po každém kroku lze vyjádřit jako:

$$\Phi = \Phi - \eta \cdot \nabla_{\Phi} C(\Phi, D). \tag{2.14}$$

Kvůli nutnosti vypočítat gradient pro celý dataset pro provedení jediného kroku může být batch gradient descent velmi pomalý až neproveditelný, pokud se celý dataset nevejde do paměti stroje. Výhodou batch gradient descentu je však jistota, že při dostatečném počtu epoch nakonec vždy nalezne lokální minimum cost funkce $C(\Phi)$ (pro konvexní cost funkce s jistotou nalezne minimum globální) [23].

Stochastický gradient descent

Stochastický gradient descent se vyznačuje tím, že parametry Φ jsou upravovány po každém páru (x_i, y_i) z trénovacího datasetu D, gradient je vypočítáván pro každý tento pár. Pro parametry Φ tedy po každém kroku platí:

$$\Phi = \Phi - \eta \cdot \nabla_{\Phi} C(\Phi, (x_i, y_i)). \tag{2.15}$$

Výhodou stochastického gradient descentu oproti batch gradient descentu je jeho obvykle vyšší rychlost. Další vlastností této metody je poměrně velké kolísání cost funkce $C(\Phi)$ způsobené častými a mnohdy velmi proměnlivými změnami parametrů Φ. Díky této vlastnosti se dokáže stochastický gradient descent na rozdíl od výše zmíněné metody dostat i do oblastí funkce s jiným, potenciálně lepším lokálním minimem. Tato kolísavost však také komplikuje konvergenci přesně do lokálního minima [23].

Mini-batch gradient descent

Kompromisem mezi dvěma výše zmíněnými metodami je tzv. mini-batch gradient descent. V rámci této metody je trénovací dataset D rozdělen na mini-batche obsahující n párů (x, y). Parametry Φ jsou změněny po výpočtu gradientu pro celou mini-batch:

$$\Phi = \Phi - \eta \cdot \nabla_{\Phi} C(\Phi, (x_{[i,i+n)}, y_{[i,i+n)})). \tag{2.16}$$

Díky tomu je oproti stochastickému gradient descentu snížena proměnlivost změn, což vede k obecně stabilnější konvergenci. Mini-batch gradient descent je také dostatečně rychlý, díky existenci vysoce optimalizovaného softwaru. Z těchto důvodů se jedná o nejčastěji používaný algoritmus pro trénování neuronových sítí. Využití této metody však do problematiky přidává další proměnnou, kterou je třeba optimalizovat, a to velikost mini-batche n [23].

Shrnutí trénování neuronových sítí

Při trénování neuronové sítě dochází k hledání co nejlepších hodnot pro váhy a biasy dané sítě. S rostoucí kvalitou hodnot těchto parametrů klesá hodnota cost (loss) funkce. Zdaleka nejpoužívanější metodou trénování neuronových sítí je tzv. mini-batch gradient descent, který upravuje váhy a biasy po každých n trénovacích párech. Důležitými hyperparametry, které ovlivňují trénovaní neuronových sítí, jsou tedy velikost mini-batchů n a počet epoch [23].

3 Praktická část

Praktickým výstupem této práce je program, který pro určitý vstupní dataset navrhne co nejlepší strukturu neuronové sítě, na základě uživatelem definovaných možných hyperparametrů. Tento úkol obvykle vykonává člověk na základě svých zkušeností s touto problematikou. Výsledný program má tomuto člověku pomoct tak, že mu na samotném začátku práce navrhne výchozí bod ve formě relativně dobré struktury, se kterou pak může dále pracovat. Nástroj umožňuje uživateli využít své zkušenosti stanovením možných hodnot, kterých hyperparametry můžou dosahovat. Nástroj pak vytvoří různé variace těchto hyperparametrů, a tím tedy vytvoří i možné architektury. Takto je částečně kompenzována možná zaujatost uživatele – nástroj vytvoří i ty struktury, které by uživatel mohl předem zavrhnout.

3.1 Hledání architektury neuronové sítě

Problém hledání architektury spočívá v optimalizaci dílčích hyperparametrů neuronové sítě pro daný problém. Tyto hyperparametry zahrnují například:

- počet vrstev sítě,
- počet neuronů ve vrstvách,
- druhy aktivačních funkcí ve vrstvách,
- druh použité cost funkce při učení,
- velikost mini-batchů,
- optimizer provádějící gradient descent,
- parametry specifické pro CNN, například velikost kernelů.

Výsledný program pracuje s ANN a CNN. Parametry, které program optimalizuje, byly z důvodu rychlejšího běhu omezeny na počet vrstev, počet neuronů ve vrstvách a druhy použitých aktivačních funkcí v případě ANN, v případě CNN se optimalizuje počet konvolučních vrstev, počet filtrů v konvolučních vrstvách, rozměry kernelu v konvolučních vrstvách a rozměry poolingu v pooling vrstvách.

Problém hledání architektur neuronových sítí se dá rozložit na tři dílčí komponenty [24]:

- prostor hledání – obecně definuje jaké architektury lze během hledání nalézt. Prostor hledání je vhodné zmenšit omezením nějakých hyperparametrů na předem stanovené hodnoty, aby hledání bylo jednodušší. Toto omezení však může negativně ovlivnit kvalitu nalezených architektur na základě zaujetí člověka, který toto omezení určil.
- Vyhledávací strategie – popisuje jakým způsobem je prostor hledání procházen. Prostor hledání může být exponenciálně velký až nekonečný, je potřeba vyvážit dobu hledání a kvalitu nalezených výsledků.

- Strategie posouzení výsledků – popisuje způsob, jakým je určena vhodnost nalezené architektury. Jednoduchý způsob je provést standardní trénování a testování na neviděných datech. To však může být zdlouhavé pro větší množství testovaných architektur.

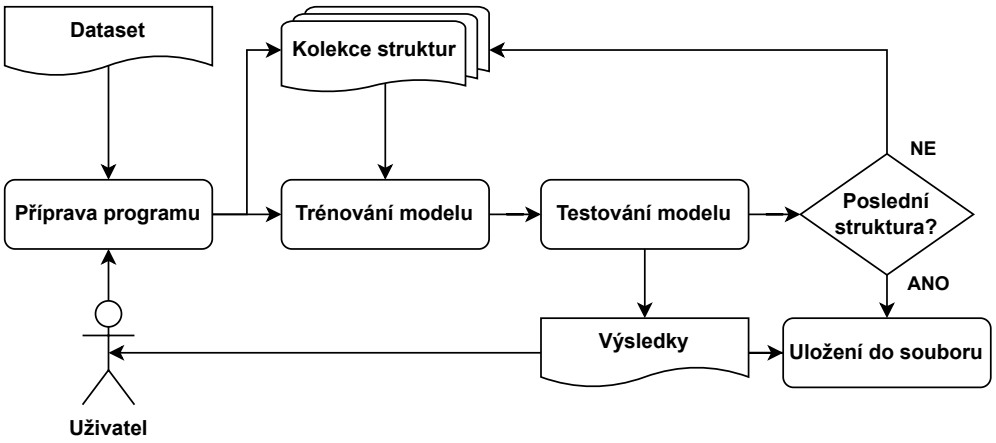
Ve výsledném programu diplomové práce je prostor hledání definován na základě uživatelem zvolených možných hodnot, kterých můžou výše vyjmenované hyperparametry dosahovat. Na základě těchto hodnot je následně vygenerován list struktur, jejichž vstupní a výstupní vrstvy se přizpůsobí danému datasetu. Strategie hledání je projití celého tohoto prostoru. Výsledky, kterých jednotlivé architektury dosahují, jsou posouzeny jejich natrénováním a následným výpočtem klasifikačních metrik (celková správnost, přesnost, úplnost, F1 skóre) na testovacích datech. Architektury jsou následně seřazeny podle uživatelem zvolené metriky.

3.2 Návrh programu

Na obr. 3.1 je vyobrazen návrh programu, který je implementován v rámci praktické části diplomové práce. Program pracuje v následujících krocích:

1. příprava programu – v rámci přípravy programu uživatel interaguje s konfiguračním souborem, do kterého zadá mimo jiné cestu k datasetu, pro který si přeje nalézt co nejlepší strukturu neuronové sítě, a jeho vlastnosti. Uživatel také má možnost si zvolit provedení základního předzpracování datasetu ve formě rozdělení na trénovací, validační a testovací data nebo standardizace hodnot feature vektorů x, popřípadě převedení zamýšlených výstupů y na one-hot vektory. Úloha předzpracování datasetu je však obtížně automatizovatelná, program tedy předpokládá, že náročnější předzpracování provede sám uživatel. Dále uživatel zadá informace o úloze jako takové, tedy počet klasifikačních tříd a podobně. Uživatel také určí možné hodnoty, kterých můžou optimalizovávané hyperparametry dosahovat. Zadány jsou také hyperparametry, které po dobu běhu programu zůstavají konstantní z důvodu úspory času. Uživatel také nastaví v jaké formě chce obdržet výsledky.
2. Vygenerování kolekce struktur – na základě uživatelského nastavení z konfiguračního souboru je vygenerován list struktur, které se liší v optimalizovávaných hyperparametrech. Struktury jsou například počtem neuronů vstupní a výstupní vrstvy přizpůsobené datasetu na základě informací, které o tomto datasetu byly uživatelem zadány.
3. Trénování modelu – během tohoto kroku se vybere jedna architektura z kolekce a je následně natrénována na trénovacích datech. Trénovaní může využívat multithreadingu, pokud tuto možnost uživatel specifikuje během konfigurace.

4. Testování modelu – po natrénovaní modelu je struktura sítě na testovacích datech ohodnocena pomocí příslušných metrik (celkové správnosti, přesnosti, úplnosti a F1 skóre). Informace o struktuře a její hodnocení jsou následně vloženy do tabulky výsledků. Poté je k natrénování vybrána další struktura z kolekce a krok 3 se opakuje.
5. Zobrazení výsledků a uložení do souboru – pokud se v kolekci nenachází žádná další nevyzkoušená architektura, program vytvoří pro n (zvoleno uživatelem) nejlepších architektur zprávy s výsledky a zobrazí je uživateli. Program také může m (taktéž zvoleno uživatelem) nejlepších natrénovaných struktur uložit do uživatelem zvolené složky, spolu s jejich zprávou s výsledky. Následně se program zastaví. Struktury jsou řazeny podle hodnoty jedné z metrik, kterou uživatel vybere.



Obr. 3.1: Návrh výsledného programu.

3.3 Použitý software

V této sekci je stručně představen hlavní software použitý během vývoje programu pro optimalizaci struktury neuronových sítí.

Python 3

Python je vysokoúrovňový hybridní programovací jazyk, který se vyznačuje poměrně jednoduchým syntaxem a ohraničováním bloků jejich odsazením. Dalšími vlastnostmi Pythonu jsou dynamický type checking a správa paměti vykonávaná

garbage collectorem. Jedná se o interpretovaný jazyk, obecně je tedy možné ho považovat za pomalejší než jeho kompilované protějšky. Python patří v současnosti k nejpopulárnějším programovacím jazykům [25].

Python je jazykem zvoleným pro implementaci praktické části této práce z důvodu existence velmi optimalizovaných machine learning knihoven TensorFlow a Keras, které rovněž disponují přehlednou dokumentací a aktivní komunitou. Dalším důvodem volby tohoto jazyka je osobní preference autora.

TensorFlow 2

TensorFlow je svobodná a open-source state-of-the-art knihovna zaměřená na strojové učení. Knihovna TensorFlow je vyvíjena společností Google. Díky množství vysokoúrovňových API (Application Programming Interface) je možné s touto knihovnou pracovat relativně rychle a intuitivně. TensorFlow je napsána primárně v Pythonu a C++. Mezi základní funkcionality TensorFlow patří:

- výpočet numerických operací na multidimenzionálních tenzorech,
- podpora výpočtů pomocí grafických karet,
- automatická diferenciace,
- vytváření, trénování a export modelů.

V rámci TensorFlow je rovněž zakomponována kolekce veřejných datasetů, které je možné pohodlně naimportovat a použít [26].

Keras

Keras je v Pythonu napsané, vysokoúrovňové API knihovny TensorFlow, které se zabývá primárně hlubokým učením. Keras byl vyvinut s důrazem na možnost rychlého experimentování a omezení zátěže kladené na uživatele. Hlavními datovými strukturami Kerasu jsou vrstvy a modely, základním modelem pak je sekvenční model, tedy lineární list po sobě jdoucích vrstev neuronové sítě. Keras obsahuje i další prvky důležité pro vytváření neuronových sítí, jako aktivační funkce, optimizery gradient descentu, loss funkce a podobně [27].

Google Colab

Google Colab dovoluje uživatelům zadarmo používat Jupyter Notebooky běžící na cloudu Googlu. Mezi výhody Colabu patří podpora knihoven strojového učení, které jsou na cloudu už připraveny (mezi nimi je i TensorFlow a Keras). Další výhodou Colabu oproti klasickému IDE (Integrated Development Environment) je možnost rychle spouštět a upravovat kód na jakémkoli zařízení bez nutnosti cokoliv importovat. Google rovněž nabízí omezený přístup k možnosti zrychlit náročné výpočty pomocí připravených grafických karet. Google Colab byl použit při raném vývoji, následně

od něj bylo upuštěno z důvodu omezeného výpočetního výkonu propůjčovaném Googlem.

PyCharm

PyCharm je integrované vývojové prostředí pro jazyk Python. Toto IDE je vyvíjeno českou společností JetBrains. Jedná se o populární volbu mezi Python vývojáři díky svému intuitivnímu ovládání a množství užitečných nástrojů pro tvorbu a debugování kódu. V rámci implementace byla použita edice PyCharm Professional.

NumPy

NumPy je jeden ze základních vědeckých nástrojů používaných v jazyce Python. Jedná se o knihovnu zaměřenou na optimalizované operace s vícedimenzionálními poli. NumPy je vhodný zejména na zpracovávání velkého množství strukturovaných dat. Pro účely této práce byla tato knihovna přímo využita hlavně pro práci s datasety [28].

Ukázkové datasety

Pro testování programu vytvořeného v rámci diplomové práce bylo zvoleno pět známých ukázkových datasetů. Těmito datasety jsou:

- MNIST (Modified National Institute of Standards and Technology) fashion dataset [29] – obrázky 28x28 celkem 10 druhů oblečení,
- Iris flower dataset [30] – 4 údaje o celkem 3 druzích květin,
- Palmer penguins dataset [31] – 4 údaje o celkem 3 druzích tučňáků,
- MNIST handwritten digits dataset [32] – obrázky 28x28 celkem 10 druhů číslic,
- Titanic dataset [33] – 13[1] údajů o jednotlivých pasažérech a informace o tom, jestli přežili potopení.

Pro práci s datasety byly okrajově využity nástroje scikit-learn a pandas.

DNP3 Intrusion detection dataset

Nástroj byl také otestován na datasetu DNP3 (Distributed Network Protocol 3) Intrusion detection. Tento dataset obsahuje 103 features vyjadřujících informace o tocích průmyslového protokolu DNP3 během probíhajícího simulovaného útoku. Labely pak vyjadřují jeden z devíti možných druhů útoku na danou průmyslovou síť. Dataset obsahuje 5994 párů (x, y) [34].

[1] Jen 8 z nich je však relevantních k šanci na přežití, jen tyto jsou použity.

3.4 Implementace

Kód výsledného programu diplomové práce se skládá z několika Python modulů a dvou textových konfiguračních souborů. Těmito soubory jsou:

- *conf.ini* – konfigurační soubor pro ANN,
- *cnn_conf.ini* – konfigurační soubor pro CNN,
- *arch_generator.py* – Python modul obsahující třídu *ArchGen*, která slouží k načtení datasetu a vygenerování ANN architektur na základě *conf.ini*,
- *cnn_arch_generator.py* – Python modul obsahující třídu *CArchGen*, která dědí z *ArchGen* a generuje architektury CNN,
- *tools.py* – Python modul, ve kterém jsou implementovány nejrůznější funkce, které program využívá. Obsaženy jsou například funkce pro správně čtení nastavení z *conf.ini* a *cnn_conf.ini*, funkce pro samotné natrénování vygenerovaných struktur, funkce pro vygenerování výsledných reportů o konkrétní architektuře, a podobně.
- *dataset_playground.py* – Python modul s funkcemi, které byly použity pro předzpracování datasetů, na kterých byl nástroj testován,
- *main.py* – funkce main, která využívá výše zmíněné moduly a slouží ke spuštění nástroje.

V následujících sekcích je na výpisech zdrojového kódu podrobně vysvětleno, jak každý z těchto souborů funguje.

3.4.1 Konfigurační soubory

Pro práci s konfiguračními soubory je použit modul *configparser.py*, který je obsažen ve standardní Python knihovně. Tento modul pracuje s textovými soubory ve formátu *.ini*. Strukturálně jsou *.ini* soubory složeny ze sekcí, které obsahují klíče. Každý klíč má svou hodnotu, kterou *configparser* dokáže přečíst a uložit do proměnné v rámci Python programu [35].

Jak již bylo zmíněno, konfigurační soubory jsou celkem dva – jeden pro umělé neuronové sítě a druhý pro konvoluční neuronové sítě. Tyto soubory se od sebe liší pouze v sekcích *hyperpars*, ve kterých uživatel zadává možné hodnoty optimalizovávaných hyperparametrů. V následujícím textu jsou vypsány a popsány jednotlivé sekce konfiguračního souboru *conf.ini* a sekce *hyperpars* souboru *cnn_conf.ini*.

Obecné nastavení

Výpis 3.1 obsahuje sekci *learning_settings*, pomocí které uživatel nastavuje obecné parametry definující úlohu a také hyperparametry, které zůstávají konstantní po dobu běhu programu. Klíči v rámci sekce *learning_settings* tedy jsou:

- *loss_function* – ztrátová funkce, která se má během učení minimalizovat. Podporovány jsou v Kerasu implementované ztrátové funkce [36]. Například ztrátová funkce *categorical_crossentropy* se používá pro klasifikaci do několika tříd[2].
- *epochs* – Počet trénovacích epoch pro každou architekturu.
- *output_neurons* – Počet neuronů výstupní vrstvy architektur.
- *batch_size* – Velikost mini-batche.
- *output_activation* – Aktivační funkce ve výstupní vrstvě architektur. Podporovány jsou v Kerasu implementované aktivační funkce [37].
- *optimizer* – Algoritmus pro provádění gradient descentu. Podporovány jsou v Kerasu naimplementované optimizery [38].
- *multithreading* – Možnost zapnout (pomocí True) nebo vypnout (pomocí False) multithreading pro trénování modelů.
- *number_of_threads* – Počet vláken při zapnutém multithreadingu.
- *input_shape* – Rozměry dat, které vstupují do vstupní vrstvy. Jako příklad je uveden jednorozměrný vektor o délce 784.

Výpis 3.1: Sekce obecných nastavení učení.

```
[learning_settings]
loss_function = categorical_crossentropy
epochs = 10
output_neurons = 10
batch_size = 128
output_activation = softmax
optimizer = adam
multithreading = True
number_of_threads = 4
input_shape = (784)
```

Dataset

Výpis 3.2 obsahuje sekci *dataset*, která slouží k zadání cesty k datasetu, jeho rozdělení na trénovací, validační a testovací data, popřípadě k provedení základního předzpracování. Klíči v rámci sekce *dataset* tedy jsou:

- x – cesta ke vstupům,
- y – cesta k zamýšleným výstupům,
- *train_part* – velikost trénovací části datasetu,
- *validation_part* – velikost validační části datasetu,

[2] I pro binární klasifikaci je však očekávána *categorical_crossentropy*, aby bylo možné správně vyhodnotit výstupy modelu.

- *test_part* – velikost testovací části datasetu,
- *y_do_one_hot* – převedení zamýšlených výstupů y do one-hot vektorů (True/False),
- *y_do_scaling* – provedení standardizace pro vstupy x.

Výpis 3.2: Sekce nastavení pro dataset.

```
[dataset]
x = Y:/PythonProjekty/Datasets/MNIST_digits/x.csv
y = Y:/PythonProjekty/Datasets/MNIST_digits/y.csv

x_shape = (70000, 784)

train_part = 0.6
validation_part = 0.2
test_part = 0.2

y_do_one_hot = True
x_do_scaling = True
```

Hyperparametry ANN

Výpis 3.3 obsahuje sekci *hyperpars*, která slouží k zadání hodnot, kterých můžou optimalizovávané hyperparametry ANN dosahovat. Klíče v rámci sekce *hyperpars* odpovídají optimalizovávaným hyperparametrům:

- *number_of_hidden_layers* – možné počty skrytých vrstev architektur,
- *layer_range* – pokud je hodnota tohoto klíče True, tak možné počty skrytých vrstev budou interval [1, n], kde n je nejvyšší číslo z hodnoty klíče *number_of_hidden_layers*.
- *number_of_neurons* – Možné počty neuronů ve skrytých vrstvách struktur.
- *activation_funcs* – Možné aktivační funkce ve skrytých vrstvách. Podporovány jsou aktivační funkce implementované v Kerasu [37].

Výpis 3.3: Sekce nastavení optimalizovávaných hyperparametrů ANN.

```
[hyperpars]
number_of_hidden_layers = 2, 3
layer_range = False
number_of_neurons = 16, 32, 64
activation_funcs = relu, sigmoid
```

Hyperparametry CNN

Jak již bylo zmíněno, konfigurační soubory pro ANN a CNN se liší pouze v sekci *hyperpars*, tedy v této sekci. Výpis 3.4 obsahuje sekci *hyperpars* pro konvoluční sítě. Klíče této sekce odpovídají hyperparametrům, které jsou optimalizovávány pro CNN:

- *number_of_conv_layers* – možné počty konvolučních vrstev architektur,
- *layer_range* – viz předchozí text,
- *number_of_filters* – možné počty filtrů v konvolučních vrstvách struktur.
- *kernel_size* – Možné rozměry kernelů filtrů konvolučních vrstev,
- *pooling_size* – možné rozměry poolingu v max pooling vrstvách (tyto vrstvy jsou přidány za každou konvoluční vrstvu)
- *activation_funcs* – možné aktivační funkce v konvolučních vrstvách [37].

Výpis 3.4: Sekce nastavení optimalizovávaných hyperparametrů CNN.

```
[hyperpars]
number_of_conv_layers = 3, 4
layer_range = False
number_of_filters = 32, 64
kernel_size = (3, 3), (5, 5)
pooling_size = (2, 2)
activation_funcs = relu
```

Zobrazení výsledků

Ve výpise 3.5 je vyobrazena sekce *visualize_results*, která slouží uživateli k nastavení způsobu, jakým si přeje do konzole vypsat výsledky po natrénování a otestování struktur. Tato sekce obsahuje celkem tři klíče:

- *print_top* – určuje kolik nejlepších struktur bude po otestování vypsáno do konzole. O tom, které struktury jsou nejlepší, je rozhodnuto na základě jedné uživatelem vybrané metriky.
- *generate_confusion_matrix* – Dává uživateli možnost vygenerovat matici záměn pro každou strukturu pomocí knihovny *scikit-learn* a zobrazit ji spolu s výsledky [39].
- *generate_classification_report* – Dává uživateli možnost vygenerovat hlášení o výsledcích klasifikace pomocí knihovny *scikit-learn*. Toto hlášení obsahuje tabulku s metrikami (přesnost, úplnost F1 skóre) pro každou klasifikační třídu a také jejich vážené a micro průměry [39].

Výpis 3.5: Sekce nastavení zobrazování výsledků.

```
[visualize_results]
print_top = 10
generate_confusion_matrix = True
generate_classification_report = True
```

Volba metriky pro seřazení

Předposlední sekce konfiguračního souboru *sort_by* je vypsána ve výpise 3.6. V této sekci si uživatel vybírá metriku, podle které budou architektury po otestování seřazeny, vypočítány jsou vždy všechny čtyři. Také si zde vybírá jakým způsobem mají být vypočítány finální hodnoty metrik odvozených z matice záměn – na výběr jsou jejich vážené průměry (weighted), macro průměry a micro průměry.

Výpis 3.6: Sekce nastavení metriky pro seřazení.

```
[sort_by]
metrics_type = weighted
accuracy = True
precision = False
recall = False
f1 = False
```

Ukládaní modelů

Poslední sekcí je *save_models*, kde uživatel nastavuje ukládání natrénovaných modelů do složky. Tato sekce je vyobrazená ve výpise 3.7 a obsahuje celkem tři klíče:

- *save* – možnost vypnout a zapnout ukládání modelů,
- *path* – specifikace cesty ke složce, do které si uživatel přeje uložit natrénované modely,
- *save_top* – výběr kolik nejlepších natrénovaných modelů (podle zvolené metriky) bude do určené složky uloženo.

Výpis 3.7: Sekce nastavení ukládaní modelů.

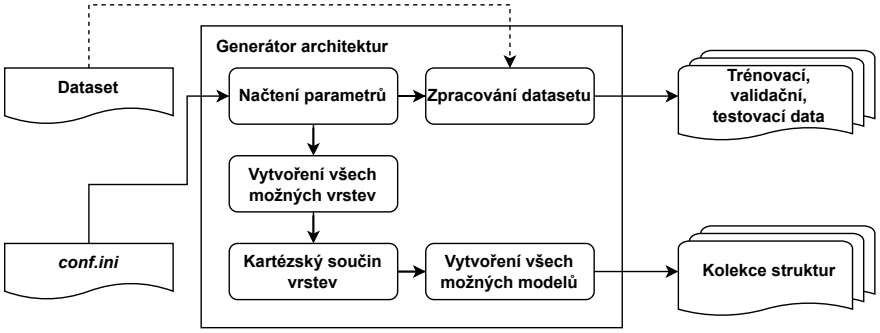
```
[save_models]
save = True
path = Y:\PythonProjekty\Modely
save_top = 5
```

3.4.2 Generátory architektur

Klíčovou součástí vytvořeného nástroje je samotné generování architektur na základě hodnot, které uživatel zadal do konfiguračních souborů. V této kapitole je nejprve představena logika, na základě které program převádí uživatelské nastavení na list možných architektur a poté samotný kód, který tuto logiku vykonává. Uvnitř generátorů architektur je rovněž implementována funkce určená ke zpracování datasetu. Výstupy generátoru architektur jsou tedy trojice trénovacích, validačních a testovacích dat, a samotný list vytvořených architektur.

Logika

Na obr. 3.2 je vyobrazen diagram generátoru architektur. Nejprve je z konfiguračního souboru načteno a zpracováno uživatelské nastavení. Zde je nutné brát ohled na to, že hodnoty různých klíčů můžou být rozdílných typů (číslo, více čísel, řetězec, ...). Tyto parametry jsou následně využity pro úpravu datasetu a vygenerování samotných architektur. Zpracováním datasetu se rozumí jeho rozdělení na trénovací, validační a testovací data v uživatelem stanoveném poměru, popřípadě provedení základního preprocessingu ve formě standardizace x-ových hodnot a převodu labelů y do one-hot vektorů.



Obr. 3.2: Diagram generátoru architektur.

Z načtených optimalizovávaných hyperparametrů potřebných pro tvorbu samotných architektur se nejprve vytvoří všechny možné (hustě propojené nebo konvoluční) vrstvy. Tyto vrstvy jsou následně uloženy v listu a slouží jako vzory pro vrstvy, ze kterých budou skládány finální architektury. Následně jsou z těchto vzorů k-násobnými kartézskými součiny vypočteny všechny možné uspořádání vrstev, kde k jsou uživatelem zvolené možné počty skrytých vrstev. Pro každý možný počet vrstev k jsou tedy vypočteny všechny variace s opakováním z množiny všech možných vrstev. Pro každou variaci se pak vytvoří sekvenční model, do kterého jsou přidány

naklonované vrstvy[3] z této variace. Ke každému modelu jsou také přidány vstupní a výstupní vrstvy vytvořené podle konfiguračního souboru.

Implementace

Jak již bylo zmíněno, generování architektur je vykonáváno objektem třídy *ArchGen* (*CArchGen* pro CNN), který je obsažen v modulu *arch_generator.py*. Výpis 3.8 představuje konstruktor této třídy. Jak je z výpisu patrné, konstruktor přijímá parametr *conf*, který představuje cestu ke konfiguračnímu souboru *conf.ini* (popřípadě *cnn_conf.ini*). Poté jsou v konstruktoru načteny některé hodnoty z konfiguračního souboru a uloženy do odpovídajících proměnných.

Výpis 3.8: Konstruktor generátoru architektur.

```
def __init__(self, conf):
    self.config = config.ConfigParser()
    self.config.read(conf)

    self.output_neurons = self.config.getint
        ('learning_settings', 'output_neurons')
    self.output_function = self.config
        ['learning_settings']['output_activation']
    self.selected_loss = self.config
        ['learning_settings']['loss_function']
    self.epochs = self.config.getint
        ('learning_settings', 'epochs')
    self.input_shape = (t.str_to_int_tuple(
        self.config['learning_settings']
        ['input_shape'])[0])
```

Další metodou objektu *ArchGen* je metoda *read_hyperpars*, která je vypsána ve výpise 3.9. Tato metoda slouží k přečtení a zpracování hyperparametrů, na základě kterých se později budou vytvářet architektury. Tato metoda využívá funkce implementované v modulu *tools.py*, který je naimportován pod označením *t*, například funkce *t.str_to_int_list* převádí řetězec přečtený z konfiguračního souboru na list celých čísel. Modulu *tools.py* se blíže věnuje sekce 3.4.3. Tato funkce se využívá například pro přečtení možných počtů vrstev nebo neuronů ve vrstvách. Výstupem metody *read_hyperpars* jsou v případě ANN zpracované listy možných počtů plně propojených vrstev, počtů neuronů a aktivačních funkcí. V případě CNN to jsou zpracované listy možných počtů konvolučních vrstev, počtů filtrů v konvolučních

[3]Variace obsahují pouze vzory, tyto vzory je potřeba naklonovat, aby modely nepracovaly se stejnými váhami a biasy

vrstvách, rozměrů kernelů filtrů konvolučních vrstev, rozměrů poolingu v pooling vrstvách a aktivačních funkcí.

Výpis 3.9: Metoda pro čtení hyperparametrů ANN.

```
def read_hyperpars(self):

    hyperpars = self.config['hyperpars']

    layer_range = eval(hyperpars['layer_range'])
    number_of_layers = t.str_to_int_list
        (hyperpars['number_of_hidden_layers'])

    if layer_range:
        number_of_layers = range
            (1, max(number_of_layers) + 1)

    number_of_neurons = t.str_to_int_list
        (hyperpars['number_of_neurons'])
    activation_funcs = t.str_to_str_list
        (hyperpars['activation_funcs'])

    return number_of_layers, number_of_neurons,
        activation_funcs
```

Výpis 3.10 představuje metodu *read_dataset*, která slouží k načtení datasetu, provedení základního předzpracování a rozdělení na trénovací, validační a testovací data dle výběru uživatele. Nejprve jsou x a y načteny do NumPy pole, poté je pomocí funkcí z *tools.py* proveden převod y do one-hot formátu a standardizace x, pokud si to uživatel zvolil[4]. Následně je x přetvarováno do uživatelem zamýšleného tvaru, protože NumPy čte csv (Comma-separated values) soubory jako jednorozměrné vektory. Nakonec je x a y pomocí funkce z *tools.py* rozděleno v určeném poměru na *x_train, y_train, x_val, y_val, x_test* a *y_test*, které jsou i výstupem této metody.

Výpis 3.10: Metoda pro čtení a zpracování datasetu.

```
def read_dataset(self):

    dataset = self.config['dataset']
    x = np.loadtxt(dataset['x'], delimiter=',')
    y = np.loadtxt(dataset['y'], delimiter=',')
```

[4]Program však očekává, že y bude ve formě one-hot vektorů.

```
    if eval(dataset['y_do_one_hot']):
        y = t.do_one_hot(y)

    if eval(dataset['x_do_scaling']):
        x = t.do_scaling(x)

    x = x.reshape(t.str_to_int_tuple
    (dataset['x_shape'])[0])

    x_train, x_val, x_test = t.split_nparray_to_3(x,
        r1=float(dataset['train_part']),
        r2=float(dataset['validation_part']),
        r3=float(dataset['test_part']))

    y_train, y_val, y_test = t.split_nparray_to_3(y,
        r1=float(dataset['train_part']),
        r2=float(dataset['validation_part']),
        r3=float(dataset['test_part']))

    return x_train, y_train, x_val, y_val, x_test, y_test
```

Hlavní funkcí generátoru architektur je samozřejmě generování architektur, to je realizováno metodou *generate_archs*, která je vypsána ve výpise 3.11. Tento výpis obsahuje verzi metody pro generování ANN. Metoda nejprve zavolá výše zmíněnou metodu pro načtení hyperparametrů. Následně jsou z uživatelem poskytnutých možných počtů neuronů ve vrstvách a možných aktivačních funkcí vytvořeny všechny možné hustě propojené vrstvy [40]. Tyto vrstvy jsou ukládány do listu *possible_layers*. Poté jsou pro každý možný počet skrytých vrstev (list *n_layers*) vypočteny kartézské součiny všech možných vrstev z *possible_layers*. Pro každý jeden kartézský součin je zároveň vytvořen sekvenční model Kerasu [41], do kterého se vloží:

- vstupní vrstva s patřičnými parametry,
- i skrytých vrstev (na základě kartézského součinu),
- výstupní vrstva s patřičnými parametry.

Pro šetření paměti kartézské součiny obsahují pouze reference na vrstvy v listu *possible_layers*, skryté vrstvy tedy musí být naklonovány před jejich přidáním do modelu. Bez tohoto naklonování by modely mezi sebou sdílely váhy a biasy. Klonování je prováděno exportováním nastavení vzorové vrstvy z *possible_layers* pomocí *get_config* a následným vytvořením nové vrstvy na základě tohoto nastavení. Nová

vrstva se liší od vzoru pouze jejím jménem, které má každá skrytá vrstva v programu unikátní. Takto vytvářené modely jsou ukládány do listu *model_collection*, který je i konečným výstupem této metody.

Výpis 3.11: Metoda pro generování architektur ANN.

```
def generate_archs(self):
    n_layers, n_neurons, activations_funcs =
        self.read_hyperpars()

    possible_layers = []

    for i in n_neurons:
        for j in activations_funcs:
            layer_blueprint =
                keras.layers.Dense(i, activation=j)
            possible_layers.append(layer_blueprint)

    model_collection = []
    layer_number = 0

    for i in n_layers:
        prod = itertools.product(
            possible_layers, repeat=i)

        for j in prod:
            model = keras.Sequential()
            model.add(keras.layers.Input(
                shape=self.input_shape))

            for layer in j:
                con = layer.get_config()
                cloned_layer = type(layer).from_config(con)
                cloned_layer._name =
                    cloned_layer.name + str(layer_number)
                layer_number = layer_number + 1
                model.add(cloned_layer)
            model.add(keras.layers.Dense(
                self.output_neurons,
                activation=self.output_function))
            model_collection.append(model)
    return model_collection
```

Generování CNN architektur

Architektury konvolučních neuronových sítí jsou generovány pomocí třídy *CArchGen*, která se nachází v modulu *cnn_arch_generator.py* a dědí ze třídy *ArchGen*, která byla právě popsána. V *CArchGen* jsou překryty metody *read_hyperpars* a *generate_arch*, aby odpovídaly vlastnostem CNN. Generování konvolučních sítí je mírně složitější kvůli většímu množství optimalizovávaných hyperparametrů a existenci pooling vrstev [42]. Z tohoto důvodu nemůžou být z možných vrstev generovány všechny možné variace s opakováním (kartézským součinem), protože by tím vznikaly i nesmyslné architektury, například složené ze 4 po sobě jdoucích pooling vrstev.

Platí tedy, že kartézské součiny jsou počítány pouze z možných konvolučních vrstev [43]. Za každou konvoluční vrstvu pak je vložena pooling vrstva. Pro účely limitování množství vzniklých architektur využívají všechny pooling vrstvy uvnitř jedné architektury stejné rozměry poolingu.

3.4.3 Vybrané funkce z *tools.py*

Jak již bylo zmíněno, v modulu *tools.py* se mimo jiné nacházejí funkce, které jsou využívány při generování architektur. Také se tam nachází i funkce, které dále pracují s výstupem třídy *ArchGen* – listem *model_collection*. Tyto funkce slouží pro natrénování a otestování každého modelu z této kolekce a konečné zpracování výsledků. Další funkce, které se zde nacházejí, slouží například k načítání nastavení z konfiguračních souborů. V následujícím textu jsou vypsány a vysvětleny ty nejdůležitější z funkcí naimplementovaných v tomto modulu.

Zpracovávání hodnot z *conf.ini*

Výpis 3.12 představuje funkci, pomocí které objekty třídy *ArchGen* zpracovávají možné hodnoty optimalizovávaných hyperparametrů. Vstupem funkce je řetězec *values*, který může například představovat hodnotu klíče *number_of_hidden_layers* ze sekce *hyperpars* konfiguračního souboru *conf.ini*. Z tohoto řetězce jsou nejprve odstraněny mezery. Poté je řetězec na základě čárek rozdělen do listu několika podřetězců. List podřetězců je pak převeden na list celých čísel zbavený duplikátů[5], což je i výstup této funkce. S tímto listem celých čísel již dokáže pracovat objekt třídy *ArchGen* a použít ho pro generování architektur.

Výpis 3.12: Funkce pro zpracování možných hodnot hyperparametrů z *conf.ini*.

```
def str_to_int_list(values):
    og_values = values
```

[5]Funkce *remove_duplicates* je další z funkcí naimplementovaných v *tools.py*.

```
    try:
        values = ''.join(values.split())
        splits = values.split(sep=",")
        return remove_duplicates(
            [int(s) for s in splits])
    except ValueError:
        print("Reading config unsuccessful,
            check the formatting of: " + og_values)
        exit(0)
```

V *tools.py* se také nacházejí obdobné funkce pro převod řetězců z konfiguračních souborů na jiné typy, než listy celých čísel. Například hodnota klíče *activation_funcs* musí být převedena na list řetězců, z nichž každý bude odpovídat určitému druhu aktivační funkce. Vypisování všech těchto funkcí je však nadbytečné, protože fungují skoro stejně.

Předzpracování datasetu

Jak již bylo zmíněno, nástroj umožňuje provést velmi základní předzpracování datasetu – naimplementováno je převedení y do formy one-hot vektorů[6] a standardizace numerických features z x. Předzpracování dat však není účelem tohoto nástroje, rozsáhlejší předzpracování tedy musí být provedeno uživatelem předem. Výpis 3.13 vyobrazuje další funkci z *tools.py*, která provádí standardizaci x pomocí objektu třídy *StandardScaler* z knihovny *scikit-learn* [44]. Pomocí objektu třídy *OneHotEncoder* ze té samé knihovny je pak stejným způsobem vyřešeno i one-hot kódování y [45].

Výpis 3.13: Funkce pro standardizaci x.

```
def do_scaling(x):
    scaler = StandardScaler()
    return scaler.fit_transform(x)
```

Učení a testování modelů

V *tools.py* je také implementována funkce *train_model*, která slouží ke kompilaci, natrénování a otestování jednoho sekvenčního Keras modelu. Tato funkce je volána dalšími funkcemi z *tools.py*, a to funkcí *train_models*, která ji spouští na listu všech architektur vygenerovaných pomocí *ArchGen*, a funkcí *train_models_on_thread*, která ji spouští na listu architektur přiřazených jednomu vláknu. Můžou tedy nastat dvě situace:

- multithreading není aktivován:

[6] Nástroj pro správné fungování vyhodnocení výsledků očekává tuto formu y.

  - funkce *train_models* zavolá funkci *train_model* pro každou architekturu z listu vytvořeným objektem třídy *ArchGen*.
- Multithreading je aktivován s n vlákny:
  - funkce *train_models* vytvoří n vláken,
  - každé vlákno si samo přiřadí část listu architektur,
  - vlákna na své části architektur zavolají funkci *train_models_on_thread*, která rovněž využívá funkci *train_model*.

Výpis 3.14 vyobrazuje zmíněnou funkci *train_model*. Tato funkce očekává celkem 10 vstupů:

- *config* – objekt třídy *ConfigParser* [35] s načteným konfiguračním souborem,
- *model* – sekvenční model Kerasu [41], který má být natrénován a otestován,
- *x_train, y_train, x_val, y_val, x_test, y_test* – připravený dataset,
- *hall_of_fame* – list, do kterého bude uložen výsledný natrénovaný model spolu s jeho dosaženými výsledky,
- *verb* – nastavení výpisů do konzole během trénování modelu[7].

Nejprve je model zkompilován metodou *compile*, která přijímá jako argumenty zvolený optimizer gradient descentu a zvolenou ztrátovou funkci. Tyto argumenty jsou přečteny z konfiguračního souboru. Argument *metrics* stanovuje jaké metriky budou vypisovány během učení, zde je konstantně ponechána celková správnost. Po zkompilování je model natrénován pomocí metody *fit*. Kromě zpracovaného datasetu tato metoda potřebuje i argumenty s počtem epoch a velikostí mini-batche. Ty jsou načteny z konfiguračního souboru. Po natrénování model vytvoří predikce pro testovací data pomocí další funkce z *tools.py* – funkce *get_predictions*. Na základě těchto predikcí jsou pak pro daný model vypočítány výsledky testování ve formě:

- celkové správnosti,
- přesnosti (vážený průměr/macro průměr/micro průměr)[8],
- úplnosti (vážený průměr/macro průměr/micro průměr),
- F1 skóre (vážený průměr/macro průměr/micro průměr),
- matice záměn (volitelné),
- klasifikačního reportu (volitelné).

Funkce, které výpočet těchto výsledků vykonávají, jsou opět všechny implementované v rámci *tools.py*. Všechny tyto funkce využívají pro své výpočty modul *sklearn.metrics* [39]. Klasifikační report mimo jiné obsahuje přesnost, úplnost a F1 skóre pro každou klasifikační třídu. Následně je z modelu a těchto výsledků vytvořen slovník *model_results*, který je vložen do listu *hall_of_fame*.

[7] Tyto výpisy musí být vypnuty pokud je použit multithreading, protože by se je všechna vlákna pokoušela vypsat současně.

[8] Možné zvolit v konfiguračním souboru.

Výpis 3.14: Funkce pro kompilaci, natrénování a otestováni modelu.

```
def train_model(config, model, x_train, y_train, x_val,
y_val, x_test, y_test, hall_of_fame, verb):
    model.compile(
        optimizer=config['learning_settings']
                        ['optimizer'],
        loss=config['learning_settings']
                    ['loss_function'],
        metrics='accuracy')

    model.fit(x=x_train, y=y_train,
        validation_data=(x_val, y_val),
        batch_size=config.getint(
            'learning_settings', 'batch_size'),
        epochs=config.getint(
            'learning_settings', 'epochs'),
        verbose=verb)

    y_pred_amax, y_test_amax =
        get_predictions(model, x_test, y_test)

    if eval(config['visualize_results']
    ['generate_confusion_matrix']):
        cm = print_confusion_matrix(
            y_test_amax, y_pred_amax)
    else:
        cm = 'Confusion matrix was not generated'

    if eval(config['visualize_results']
    ['generate_classification_report']):
        cr = print_classification_report(
            y_test_amax, y_pred_amax)
    else:
        cr = 'Classification report was not generated'

    acc, pre, rec, f1 =
        calculate_metrics(y_test_amax, y_pred_amax,
            config)

    model_results = {'model': model, 'accuracy': acc,
        'precision': pre, 'recall': rec, 'f1': f1,
```

```
        'classification_report': cr,
        'confusion_matrix': cm}

    hall_of_fame.append(model_results)
```

Výpis 3.15 zobrazuje funkci *train_models*, která zavolá funkci *train_model* pro každý model z *models*. Tento argument odpovídá výstupu funkce *generate_archs* z objektu třídy *ArchGen*. V rámci funkce *train_models* je také maimplementován multithreading, který uživatel může aktivovat v konfiguračním souboru. V takovém případě dojde k vytvoření specifikovaného množství vláken a list *models* je mezi ně rozdělen. Samotné rozdělení je naimplementováno ve funkci *train_models_on_thread* a vychází z *i*, tedy z čísla vlákna.

Výpis 3.15: Funkce pro natrénování kolekce modelů.

```
def train_models(models, config, x_train, y_train, x_val,
y_val, x_test, y_test, hall_of_fame):

    num_models = len(models)

    if config.getboolean('learning_settings',
    'multithreading'):
        num_threads = min(num_models, config.getint(
            'learning_settings', 'number_of_threads'))
        print(f'Training {len(num_models)} models
            using {num_threads} threads...')

        threads = []
        for i in range(num_threads):
            t = threading.Thread(
                target=train_models_on_thread,
                args=(config, model,
                x_train, y_train, x_val,
                y_val, x_test, y_test, hall_of_fame, i))
            threads.append(t)
            t.start()

        for t in threads:
            t.join()

    else:
        print(f'Training {num_models}
            models using single thread...')
```

```
        for model in models:
            train_model(config, model, x_train, y_train,
                x_val, y_val, x_test,
                y_test, hall_of_fame, 1)
```

Zpracování a zobrazení výsledků

Další funkce z *tools.py* slouží k vyhodnocení, zobrazení a uložení výsledků. Funkce *sort_hall* slouží k utřídění listu *hall_of_fame*, ve kterém se nachází slovník pro každý model s jeho výsledky. List je utříděn podle jedné uživatelem zvolené metriky. K zobrazování výsledků patří například funkce *get_model_layers_and_activations*, která slouží pro vygenerování popisu ANN architektury. Tato funkce je vyobrazena ve výpise 3.16.

Výpis 3.16: Funkce pro vytvoření popisu struktury modelu.

```
def get_model_layers_and_activations(model):
    desc = [f"Number of layers: {len(model.layers)}"]
    for i, layer in enumerate(model.layers):
        desc.append(f"Layer {i + 1}: {layer.name}
            ({layer.activation.__name__}),
            Number of neurons: {layer.units}")

    desc = "\n".join(desc)
    return desc
```

Tuto funkci využívá funkce *format_model_results*, která vytváří ze slovníků z listu *hall_of_fame*, reprezentující různé modely, finální report pro daný model. Ukázka reportu je vyobrazena ve výpise 3.17. Prvních pět řádků reportu představuje výstup funkce *get_model_layers_and_activations*. Tyto reporty jsou vypsány do konzole pro uživatelem stanovených n nejlepších architektur. Reporty jsou také uloženy společně s m nejlepšími natrénovanými modely.

Výpis 3.17: Finální report o architektuře natrénované na titanic datasetu.

```
Number of layers: 4
Layer 1: dense_2102 (relu), Number of neurons: 64
Layer 2: dense_103 (relu), Number of neurons: 32
Layer 3: dense_2104 (relu), Number of neurons: 64
Layer 4: dense_38 (softmax), Number of neurons: 2

accuracy:
```

```
0.82682
precision:
0.81063
recall:
0.81671
f1:
0.81340

classification_report:
              precision    recall  f1-score   support

           0    0.87500   0.85217   0.86344       115
           1    0.74627   0.78125   0.76336        64

    accuracy                        0.82682       179
   macro avg    0.81063   0.81671   0.81340       179
weighted avg    0.82897   0.82682   0.82765       179

confusion_matrix:
array([[98, 17],
       [14, 50]], dtype=int64)
```

V modulu *tools.py* se nachází další obdobné funkce, které provádí stejné operace s konvolučními neuronovými sítěmi. Také je nutné poznamenat, že některé funkce z *tools.py* nebyly v tomto textu vypsány, například funkce pro ukládání modelů spolu s jejich reportem do souborů.

3.4.4 Příprava testovacích datasetů

Modul *dataset_playground.py* obsahuje funkce, které uloží vybrané datasety zakomponované v Kerasu [46], TensorFlow [47] nebo scikit-learn [48] do csv souborů. Také je zde naimplementována funkce na preprocessing a uložení DNP3 datasetu [34]. Na těchto datasetech je pak možné nástroj otestovat. Výpis 3.18 vyobrazuje funkci, která uloží dobře známý *MNIST* dataset ručně psaných čísel [32] do souborů. Tento dataset pak lze načíst do nástroje specifikováním cesty k x a y v příslušných hodnotách konfiguračních souborů.

Výpis 3.18: Funkce pro uložení *MNIST* datasetu do csv souborů.

```
def mnist_digits():
    (X_train, Y_train), (X_test, Y_test) =
        keras.datasets.mnist.load_data()
```

```
    X = np.concatenate((X_train, X_test))
    X = tf.reshape(X, [70000, 28*28])

    Y = np.concatenate((Y_train, Y_test))

    np.savetxt('../MNIST_digits/x.csv', X, delimiter=',')
    np.savetxt('../MNIST_digits/y.csv', Y, delimiter=',')
```

3.4.5 Modul *main.py*

Soubor *main.py* slouží ke spouštění celého nástroje postupným voláním jeho všech do teď popsaných částí (kromě *datatest_playground.py*). Nejprve je vytvořen objekt třídy *ConfigParser*, následně je uživatel dotázán, jestli si přeje nástroj spustit v módu pro ANN nebo CNN. Zde nastávají celkem čtyři možnosti:

- uživatel zadá „A“ – objekt třídy *ConfigParser* načte soubor *conf.ini* a objekt třídy *ArchGen* je vytvořen,
- uživatel zadá „C“ – objekt třídy *ConfigParser* načte soubor *cnn_conf.ini* a objekt třídy *CArchGen* je vytvořen,
- uživatel zadá „E“ – program je ukončen,
- uživatel zadá jakékoliv jiné písmeno – program se znova zeptá na záměr uživatele.

Výše popsané je vyobrazeno ve výpise 3.19.

Výpis 3.19: Kód pro volbu módu nástroje.

```
config = config.ConfigParser()

while True:
    paradigm = input('Type A for ANN, C for CNN or
        E for exit: ')
    if paradigm == 'a' or paradigm == 'A':
        config.read('conf.ini')
        print('You have chosen ANN!')
        ag = ArchGen('conf.ini')
        break
    elif paradigm == 'c' or paradigm == 'C':
        config.read('cnn_conf.ini')
        print('You have chosen CNN!')
        ag = CArchGen('cnn_conf.ini')
        break
    elif paradigm == 'e' or paradigm == 'E':
        exit(0)
```

```
        break
    else:
        print('Try again')
```

Výpis 3.20 představuje zbytek kódu v modulu *main.py*. Postupně jsou ve správném pořadí volány všechny součásti nástroje – proběhne tedy:

1. načtení datasetu,
2. vygenerování architektur,
3. natrénování a otestování každé architektury,
4. utřídění listu výsledků od nejlepší architektury po nejhorší,
5. vypsání n nejlepších architektur a jejich výsledků do konzole,
6. uložení[9] m nejlepších architektur a jejich výsledků do zvolené složky,
7. vypsání doby běhu programu.

Výpis 3.20: Kód pro spuštění všech částí nástroje.

```
start_time = time.time()

hall_of_fame = []

print('Reading dataset...')
selected_loss, epochs, output_neurons, output_function
    = ag.get_learning_params()
x_train, y_train, x_val, y_val, x_test, y_test =
    ag.read_dataset()

print('Generating architectures...')
model_collection = ag.generate_archs()

t.train_models(model_collection, config, x_train,
    y_train, x_val, y_val, x_test, y_test, hall_of_fame)

t.sort_hall(hall_of_fame, config)
print("TRAINING COMPLETED")

t.print_top_struct(config, hall_of_fame)

if config.getboolean('save_models', 'save'):
    t.save_models(hall_of_fame,
        config.getint('save_models', 'save_top'),
        config['save_models']['path'])
```

[9]Pokud je tato možnost aktivována v konfiguračním souboru.

```
end_time = time.time()
total_time = end_time - start_time

print(f"Program completed in {total_time:.2f} seconds.")
```

3.5 Testování nástroje na ukázkových datasetech

V této sekci je popsáno testování nástroje na datasetech zmíněných v sekci 3.3. Vypsáno je nejprve nastavení, se kterým byl nástroj spuštěn, a následně je zmíněna nejlepší nástrojem nalezená architektura společně s jejím výsledným vygenerovaným reportem. Pokud více architektur dosahuje stejných výsledků, je vypsána ta s nejmenším počtem trénovatelných parametrů.

Testování probíhalo na počítači s čtyřjádrovým procesorem Intel Core i5-7400 a 16 GB operační paměti. Je nutno podotknout, že program běžel na pozadí během toho, co byl počítač využíván i k jiným úkonům.

3.5.1 Ukázkové datasety pro ANN

Nástroj byl nejprve otestován na ukázkových datasetech pro ANN vypsaných v sekci 3.3. Validační data byla ponechána prázdná, protože z důvodu šetření času byl aktivován multithreading, během kterého jsou vypnuty průběžné výpisy o učení modelu, jak již bylo vysvětleno.

Palmer penguins a Iris flowers

Pro dataset Palmer penguins [31] a Iris flowers [30] bylo zvoleno nastavení vypsané ve výpise 3.21. Nastavení je pro tyto dva datasety totožné[10], protože se jedná o velmi podobné datasety.

Výpis 3.21: Nastavení nástroje pro dataset penguings a Iris.

```
[learning_settings]
loss_function = categorical_crossentropy
epochs = 10
output_neurons = 3
batch_size = 1
output_activation = softmax
optimizer = adam
multithreading = True
number_of_threads = 4
```

[10]Kromě hodnoty klíče *x_shape*.

```
input_shape = (4)

[dataset]
train_part = 0.6
validation_part = 0.0
test_part = 0.4

[hyperpars]
number_of_hidden_layers = 2, 3
layer_range = False
number_of_neurons = 16, 32, 64
activation_funcs = relu, sigmoid
```

Na základě nastavení možných hyperparametrů bylo celkem vytvořeno a otestováno 252 různých ANN architektur, běh programu trval celkem 611 sekund pro dataset Palmer penguins, 428 sekund pro dataset Iris flowers.

Celkem 23 architektur dosáhlo na datasetu Palmer penguins celkové správnosti 100 %. Nejméně natrénovatelných parametrů z nich měla architektura vypsaná ve výpise 3.22. Celkové správnosti 96,67 % dosáhlo pro Iris flowers dataset 30 vygenerovaných architektur, z nichž má nejméně parametrů struktura vypsaná ve výpise 3.23. Při interpretaci těchto výsledků je však nutné si uvědomit, že tyto dva datasety obsahují relativně málo párů (x, y), a to 334 pro Palmer penguins a 150 pro Iris flowers.

Výpis 3.22: Nejlepší architektura pro dataset penguings.

```
Number of layers: 3
Layer 1: dense_4 (relu), Number of neurons: 16
Layer 2: dense_25 (relu), Number of neurons: 32
Layer 3: dense_8 (softmax), Number of neurons: 3
accuracy:
1.00000
precision:
1.00000
recall:
1.00000
f1:
1.00000
classification_report:
              precision    recall  f1-score   support

           0    1.00000   1.00000   1.00000        59
           1    1.00000   1.00000   1.00000        26
```

```
           2    1.00000   1.00000   1.00000        49

    accuracy                        1.00000       134
   macro avg    1.00000   1.00000   1.00000       134
weighted avg    1.00000   1.00000   1.00000       134

confusion_matrix:
array([[59,  0,  0],
       [ 0, 26,  0],
       [ 0,  0, 49]], dtype=int64)
```

Výpis 3.23: Nejlepší architektura pro dataset Iris flowers.

```
Number of layers: 3
Layer 1: dense_336 (sigmoid), Number of neurons: 32
Layer 2: dense_37 (relu), Number of neurons: 16
Layer 3: dense_24 (softmax), Number of neurons: 3
accuracy:
0.96667
precision:
0.96957
recall:
0.96667
f1:
0.96667
classification_report:
              precision    recall  f1-score   support

           0    1.00000   1.00000   1.00000        16
           1    1.00000   0.91304   0.95455        23
           2    0.91304   1.00000   0.95455        21

    accuracy                        0.96667        60
   macro avg    0.97101   0.97101   0.96970        60
weighted avg    0.96957   0.96667   0.96667        60

confusion_matrix:
array([[16,  0,  0],
       [ 0, 21,  2],
       [ 0,  0, 21]], dtype=int64)
```

Titanic dataset

Třetím a posledním zkušebním ukázkovým datasetem pro ANN je dataset přeživších havárie Titanicu. Nástroj byl spuštěn s nastavením vyobrazeným ve výpise 3.24. Na základě tohoto nastavení bylo vygenerováno a otestováno celkem 1548 architektur.

Výpis 3.24: Nastavení nástroje pro dataset Titanic.

```
[learning_settings]
loss_function = categorical_crossentropy
epochs = 10
output_neurons = 2
batch_size = 8
output_activation = softmax
optimizer = adam
multithreading = True
number_of_threads = 4
input_shape = (23)

[dataset]
train_part = 0.6
validation_part = 0.0
test_part = 0.4

[hyperpars]
number_of_hidden_layers = 2, 3, 4
layer_range = False
number_of_neurons = 16, 32, 64
activation_funcs = relu, sigmoid
```

Trénování a testování těchto 1548 architektur trvalo 3496 sekund, tedy bezmála hodinu. Nejúspěšnější z těchto architektur dosáhla celkové správnosti 85,15 %. Report pro tuto architekturu je vypsán ve výpise 3.25.

Výpis 3.25: Nejlepší architektura pro dataset Titanic.

```
Number of layers: 5
Layer 1: dense_44520 (relu), Number of neurons: 64
Layer 2: dense_24521 (relu), Number of neurons: 32
Layer 3: dense_24522 (relu), Number of neurons: 32
Layer 4: dense_24523 (relu), Number of neurons: 32
Layer 5: dense_1208 (softmax), Number of neurons: 2
accuracy:
0.85154
```

```
precision:
0.85714
recall:
0.85154
f1:
0.84566
classification_report:
              precision    recall  f1-score   support

           0    0.83525   0.95614   0.89162       228
           1    0.89583   0.66667   0.76444       129

    accuracy                        0.85154       357
   macro avg    0.86554   0.81140   0.82803       357
weighted avg    0.85714   0.85154   0.84566       357

confusion_matrix:
array([[218,  10],
       [ 43,  86]], dtype=int64)
```

3.5.2 Ukázkové datasety pro CNN

Nástroj byl také otestován na obrazových datech, která byla klasifikována pomocí konvolučních neuronových sítích. Pro tento druh sítí nebylo možné testovat hlubší architektury z důvodu omezeného výkonu počítače, na kterém testování probíhalo.

MNIST digits dataset

Pro dataset psaných cifer [32] bylo zvoleno nastavení vypsané ve výpise 3.26. Architektury musely být omezeny na dvě konvoluční vrstvy.

Výpis 3.26: Nastavení pro MNIST digits.

```
[learning_settings]
loss_function = categorical_crossentropy
epochs = 5
output_neurons = 10
batch_size = 128
output_activation = softmax
optimizer = adam
multithreading = True
number_of_threads = 4
```

```
input_shape = (28, 28, 1)

[dataset]
train_part = 0.6
validation_part = 0.0
test_part = 0.4

[hyperpars]
number_of_conv_layers = 2
layer_range = False
number_of_filters = 32, 64
kernel_size = (2, 2), (3, 3)
pooling_size = (2, 2)
activation_funcs = relu, sigmoid
```

Na základě tohoto nastavení bylo vytvořeno a otestováno celkem 64 CNN architektur. Běh programu trval 4113 sekund, tedy asi hodinu a devět minut. Nejlepší architektura dosáhla celkové správnosti 98,34 %, report pro tuto architekturu je vyobrazen ve výpise 3.27[11].

Výpis 3.27: Nejlepší struktura pro dataset MNIST digits.

```
Number of layers: 6
Layer 1: Conv2D, 64 filters, kernel size (2, 2),
    activation function relu
Layer 2: MaxPooling2D layer, pool size (2, 2)
Layer 3: Conv2D, 64 filters, kernel size (3, 3),
    activation function relu
Layer 4: MaxPooling2D layer, pool size (2, 2)
Layer 5: Flatten layer
Layer 6: Dense layer, 10 neurons,
    activation function softmax

accuracy:
0.98336
precision:
0.98343
recall:
0.98336
f1:
0.98335
classification_report:
```

[11]Matice záměn musela být přeformátována, aby nepřetékala z obrazce.

```
              precision    recall  f1-score   support

           0    0.99161   0.98514   0.98837      2760
           1    0.98743   0.99545   0.99143      3078
           2    0.98073   0.98452   0.98262      2843
           3    0.96872   0.99165   0.98005      2873
           4    0.98002   0.99009   0.98503      2725
           5    0.99311   0.96916   0.98099      2529
           6    0.98516   0.98516   0.98516      2696
           7    0.98120   0.98616   0.98367      2963
           8    0.98719   0.96876   0.97789      2785
           9    0.98023   0.97453   0.97737      2748

    accuracy                        0.98336     28000
   macro avg    0.98354   0.98306   0.98326     28000
weighted avg    0.98343   0.98336   0.98335     28000

confusion_matrix:
[2719, 0, 9, 3, 4, 1, 11, 4, 6, 3],
[ 0, 3064, 5, 2, 0, 1, 4, 2, 0, 0],
[ 1, 8, 2799, 10, 3, 0, 1, 16, 2, 3],
[ 0, 0, 11, 2849, 0, 1, 0, 4, 5, 3],
[ 0, 6, 0, 0, 2698, 0, 1, 2, 2, 16],
[ 3, 2, 3, 37, 3, 2451, 15, 3, 6, 6],
[ 9, 2, 2, 1, 10, 5, 2656, 0, 11, 0],
[ 0, 9, 16, 8, 5, 0, 0, 2922, 0, 3],
[ 8, 8, 9, 14, 9, 5, 8, 6, 2698, 20],
[ 2, 4, 0, 17, 21, 4, 0, 19, 3, 2678],
```

MNIST fashion dataset

Pro dataset obrázků oblečení [29] bylo zvoleno nastavení vypsané ve výpise 3.28. Oproti testování na MNIST cifrách bylo učiněno několik změn:

- byla použita jen polovina datasetu, tedy 35000 párů,
- možné aktivační funkce byly omezeny na ReL,
- počet konvolučních vrstev byl zvýšen ze 2 na 3,
- počet filtrů byl zvýšen z 32/64 na 64/128.

Díky snížení počtu trénovacích párů, testovacích párů a aktivačních funkcí byl nástroj schopný v rozumném čase otestovat i architektury se třemi konvolučními vrstvami až se 128 filtry.

Výpis 3.28: Nastavení pro MNIST fashion.

```
[learning_settings]
loss_function = categorical_crossentropy
epochs = 5
output_neurons = 10
batch_size = 128
output_activation = softmax
optimizer = adam
multithreading = True
number_of_threads = 4
input_shape = (28, 28, 1)

[dataset]
train_part = 0.6
validation_part = 0.0
test_part = 0.4

[hyperpars]
number_of_conv_layers = 3
layer_range = False
number_of_filters = 64, 128
kernel_size = (2, 2), (3, 3)
pooling_size = (2, 2)
activation_funcs = relu
```

Na základě tohoto nastavení bylo vytvořeno a otestováno opět celkem 64 CNN architektur. Běh programu trval 4620 sekund – hodinu a 17 minut. Nejlepší architektura dosáhla celkové správnosti 88,84 %, report pro tuto architekturu je vyobrazen ve výpise 3.29[12].

Výpis 3.29: Nejlepší struktura pro dataset MNIST fashion.

```
Number of layers: 8
Layer 1: Conv2D, 128 filters, kernel size (3, 3),
    activation function relu
Layer 2: MaxPooling2D layer, pool size (2, 2)
Layer 3: Conv2D, 128 filters, kernel size (2, 2),
    activation function relu
Layer 4: MaxPooling2D layer, pool size (2, 2)
Layer 5: Conv2D, 128 filters, kernel size (3, 3),
    activation function relu
```

[12]Matice záměn musela být přeformátována, aby se vlezla na stranu.

```
Layer 6: MaxPooling2D layer, pool size (2, 2)
Layer 7: Flatten layer
Layer 8: Dense layer, 10 neurons,
    activation function softmax

accuracy:
0.88836
precision:
0.88898
recall:
0.88836
f1:
0.88752
classification_report:
              precision    recall  f1-score   support

           0    0.82792   0.83835   0.83310      1429
           1    0.99332   0.97238   0.98274      1376
           2    0.84814   0.85735   0.85272      1381
           3    0.79717   0.94811   0.86611      1426
           4    0.83826   0.82246   0.83029      1380
           5    0.94865   0.96882   0.95863      1411
           6    0.73881   0.64147   0.68671      1389
           7    0.94270   0.95268   0.94766      1416
           8    0.99106   0.93077   0.95997      1430
           9    0.96490   0.94860   0.95668      1362

    accuracy                        0.88836     14000
   macro avg    0.88909   0.88810   0.88746     14000
weighted avg    0.88898   0.88836   0.88752     14000
confusion_matrix:
[1198, 0, 13, 92, 5, 2, 115, 0, 4, 0],
[ 2, 1338,  0, 34, 0, 0, 2, 0, 0, 0],
[ 12, 0, 1184, 21, 82, 1, 80, 0, 1, 0],
[ 17, 5, 7, 1352, 27, 1, 17, 0, 0, 0],
[ 3, 2, 76, 82, 1135, 0, 81, 0, 1, 0],
[ 0, 0, 0, 0, 0, 1367, 0, 26, 2, 16],
[ 202, 1, 107, 84, 101, 0, 891, 0, 3, 0],
[ 0, 0, 0, 0, 0, 36, 0, 1349, 0, 31],
[ 12, 1, 9, 30, 4, 17, 19, 7, 1331, 0],
[ 1, 0, 0, 1, 0, 17, 1, 49, 1, 1292]],
```

3.6 DNP3 Intrusion detection dataset

Jako dataset byly použity dva soubory nacházející se v adresáři *Custom DNP3 Parser* ve složce trénovacích a testovacích vyvážených csv souborů uvnitř publikovaného zkomprimovaného souboru [34]. Oba soubory dohromady obsahují 5994 párů (x, y), x je pak tvořeno vektory se 103 features. Labely y představují jeden z celkem 9 různých typů útoku.

Předzpracování datasetu

Výpis 3.30 vyobrazuje funkci implementovanou v modulu *dataset_playground.py*, která slouží k předzpracování tohoto datasetu. Nejprve jsou pomocí nástroje *pandas* [49] načteny oba soubory, obsahující trénovací a testovací data. Tyto soubory jsou následně sloučeny v jeden, aby bylo možné si zvolit vlastní poměr trénovacích, validačních a testovacích dat. Tento soubor je pak rozdělen na DataFrame[13] x, obsahující všechny features, a y, obsahující pouze sloupec *Labels*. Z x jsou následně odstraněny sloupce, které nejsou relevantní pro klasifikaci. Feature *firstPacketDIR* je pak převedena na one-hot vektory, protože tato feature představuje relevantní, ale kategorická data. Výsledné x tedy obsahuje celkem 97 sloupců. Také y je převedeno na formu one-hot vektorů, protože s tímto formátem nástroj umí pracovat. Nakonec je takto připravený dataset uložen pomocí *NumPy* [28] do csv souborů.

Výpis 3.30: Funkce pro přípravu DNP3 datasetu.

```
def dnp3():
    training = pd.read_csv(
        ".../Datasets/DNP3/training.csv")
    testing = pd.read_csv(
        ".../Datasets/DNP3/testing.csv")

    ds = pd.concat([training, testing], axis=0,
        ignore_index=True)

    y = ds.pop(' Label')
    x = ds

    to_be_removed = ["flow ID", " source IP",
                     " destination IP", " date",
                     " source port", " destination port"]

    x.drop(columns=to_be_removed, inplace=True)
```

[13] Typ, se kterým pracuje *pandas*.

```
    to_one_hot = [" firstPacketDIR"]

    x = pd.get_dummies(x, columns=to_one_hot)
    y = pd.get_dummies(y, columns=[" Label"])

    np_x = x.to_numpy()
    np_y = y.to_numpy()

    np.savetxt('Y:/PythonProjekty/Datasets/DNP3_p/x.csv',
        np_x, delimiter=',')
    np.savetxt('Y:/PythonProjekty/Datasets/DNP3_p/y.csv',
        np_y, delimiter=',')
```

Nastavení nástroje pro dataset

Část nastavení v souboru *conf.ini* je vypsána ve výpise 3.31. Možné počty skrytých vrstev jsou nastaveny na 2, 3 a 4. Možné počty neuronů jsou 16, 32 a 64. Aktivační funkce pak můžou být buďto ReL nebo sigmoida. Na základě tohoto nastavení bylo vygenerováno celkem 1548 ANN architektur.

Výpis 3.31: Nastavení pro DNP3 dataset.

```
[learning_settings]
loss_function = categorical_crossentropy
epochs = 10
output_neurons = 9
batch_size = 8
output_activation = softmax
optimizer = adam
multithreading = True
number_of_threads = 4
input_shape = (97)

[dataset]
x = Y:/PythonProjekty/Datasets/DNP3_p/x.csv
y = Y:/PythonProjekty/Datasets/DNP3_p/y.csv

x_shape = (5994, 97)

train_part = 0.6
validation_part = 0.0
```

```
test_part = 0.4

y_do_one_hot = False
x_do_scaling = True

[hyperpars]
number_of_hidden_layers = 2, 3, 4
layer_range = False
number_of_neurons = 16, 32, 64
activation_funcs = relu, sigmoid
```

Výsledky

Trénování a testování těchto 1548 architektur trvalo na čtyřech vláknech již popsaného počítače celkem 6102 sekund, tedy asi hodinu a 42 minut. Nejlepší architektura dosáhla celkové správnosti rovné 99,12 %. Výsledný report pro tuto architekturu je vypsán ve výpise 3.32.

Výpis 3.32: Nejlepší struktura pro DNP3 dataset.

```
Number of layers: 5
Layer 1: dense_44496 (relu), Number of neurons: 64
Layer 2: dense_24497 (relu), Number of neurons: 32
Layer 3: dense_14498 (sigmoid), Number of neurons: 16
Layer 4: dense_24499 (relu), Number of neurons: 32
Layer 5: dense_1202 (softmax), Number of neurons: 9
accuracy:
0.99124
precision:
0.99140
recall:
0.99124
f1:
0.99124
classification_report:
              precision    recall  f1-score   support

           0    0.99615   1.00000   0.99807       259
           1    1.00000   1.00000   1.00000       274
           2    0.99254   0.98519   0.98885       270
           3    0.98467   0.99228   0.98846       259
           4    1.00000   1.00000   1.00000       269
```

```
           5     0.99231   0.95556   0.97358       270
           6     0.95556   0.99231   0.97358       260
           7     1.00000   1.00000   1.00000       264
           8     1.00000   0.99634   0.99817       273

    accuracy                         0.99124      2398
   macro avg     0.99125   0.99130   0.99119      2398
weighted avg     0.99140   0.99124   0.99124      2398

confusion_matrix:
     ([[259,   0,   0,   0,   0,   0,   0,   0,   0],
       [  0, 274,   0,   0,   0,   0,   0,   0,   0],
       [  0,   0, 266,   4,   0,   0,   0,   0,   0],
       [  0,   0,   2, 257,   0,   0,   0,   0,   0],
       [  0,   0,   0,   0, 269,   0,   0,   0,   0],
       [  0,   0,   0,   0,   0, 258,  12,   0,   0],
       [  0,   0,   0,   0,   0,   2, 258,   0,   0],
       [  0,   0,   0,   0,   0,   0,   0, 264,   0],
       [  1,   0,   0,   0,   0,   0,   0,   0, 272]])
```

Závěr

V teoretické části diplomové práce bylo nejprve představeno strojové učení obecně, s důrazem na učení s učitelem. Následně byly teoreticky popsány samotné neuronové sítě. Pozornost byla věnována nejprve neuronům, jakožto základním stavebním jednotkám neuronových sítí. Dále byly představeny některé dobře známé aktivační funkce, které jsou důležitou součástí jednotlivých neuronů. Pak byl představen algoritmus gradient descent, pomocí kterého dochází k optimalizaci parametrů neuronové sítě, tedy k samotnému učení.

Praktickým výstupem práce je nástroj naprogramovaný v jazyce Python, který využívá zejména state-of-the-art knihovny strojového učení TensorFlow a Keras. Nástroj slouží ke hledání co nejlepší struktury umělé nebo konvoluční neuronové sítě pro daný dataset. Prostor hledání je vytvářen dynamicky na základě uživatelského nastavení. Vhodnost struktury je programem posouzena jejím natrénováním a otestováním na testovacích datech a následném výpočtu relevantních klasifikačních metrik. Nástroj také umí vypočítat a zobrazit klasifikační metriky pro jednotlivé třídy a matice záměn. Výsledky nástroj prezentuje pomocí přehledného reportu, který lze vygenerovat pro stanovený počet nejlepších architektur. Naimplementována byla také možnost výsledné natrénované modely uložit do složky spolu s jejich výsledným reportem. Princip fungování nástroje je podrobně popsán pomocí četných výpisů zdrojového kódu.

Program byl otestován na pěti dobře známých ukázkových datasetech, třech pro ANN a dvěma pro CNN. Struktury nalezené programem dosahují pro ukázkové datasety celkové správnosti v rozmezí 85–100 %. Nástroj byl také otestován na datasetu útoků na průmyslový protokol DNP3, kde dokázal najít architekturu dosahující celkové správnosti 99,13 %.

Z testování vyplývá, že výsledný nástroj je vhodnější pro hledání architektur umělých neuronových sítí, které pracují maximálně s nižšími stovkami features. V případě obrazových dat trvá hledání konvolučních architektur na osobním počítači podstatně déle, nemůže tedy být rychle vyzkoušeno větší množství různých kombinací hyperparametrů.

Literatura

[1] QIN, T. *Dual Learning* [online]. Singapore: Springer Nature Singapore, 2020. Dostupné z: https://doi.org/10.1007/978-981-15-8884-6.

[2] BHAVITHA, B. K., RODRIGUES, A. P., CHIPLUNKAR N. N. Comparative study of machine learning techniques in sentimental analysis [online]. In: *2017 International Conference on Inventive Communication and Computational Technologies*. IEEE, 2017. Dostupné z: https://doi.org/10.1109/icicct.2017.7975191.

[3] SUTTON, R. S., BARTO, A. G. *Reinforcement learning: An introduction* [online]. MIT press, 2018. ISBN: 9780262352703. Dostupné z: https://books.google.cz/books?id=uWV0DwAAQBAJ.

[4] ARULKAMARAN, K., DEISENROTH, M. P., BRUNDAGE, M., BHARATH, A. A. Deep reinforcement learning: A brief survey [online]. In: *IEEE Signal Processing Magazine*. IEEE, 2017. Dostupné z: https://doi.org/10.1109/MSP.2017.2743240.

[5] XIONG, Z., CUI, Y., LIU, Z., ZHAO, Y., HU, M., HU, J. Evaluating explorative prediction power of machine learning algorithms for materials discovery using k-fold forward cross-validation [online]. In: *Computational Materials Science*. Elsevier, 2020. Dostupné z: https://doi.org/10.1016/j.commatsci.2019.109203.

[6] BAJAJ, A. Performance Metrics in Machine Learning. In: *MLOps Blog* [online]. Poslední změna: 14.11.2022. [cit. 27.11.2022]. Dostupné z: https://neptune.ai/blog/performance-metrics-in-machine-learning-complete-guide.

[7] AGRAWAL, R. Know The Best Evaluation Metrics for Your Regression Model. In: *Analytics Vidhya* [online]. Poslední změna: 21.07.2022. [cit. 27.11.2022]. Dostupné z: https://www.analyticsvidhya.com/blog/2021/05/know-the-best-evaluation-metrics-for-your-regression-model/.

[8] Classification: Accuracy. In: *Machine Learning Foundational Courses* [online]. Poslední změna: 18.07.2022. [cit. 27.11.2022]. Dostupné z: https://developers.google.com/machine-learning/crash-course/classification/accuracy.

[9] Classification: Precision and Recall. In: *Machine Learning Foundational Courses* [online]. Poslední změna: 18.07.2022. [cit. 27.11.2022]. Dostupné

z: https://developers.google.com/machine-learning/crash-course/classification/precision-and-recall.

[10] ZAYEGH, A., BASSAM, N. A. Neural Network Principles and Applications [online]. In: ASADPOUR, V. *Digital Systems*. Londýn: IntechOpen, 2018. ISBN: 978-1-78984-541-9. Dostupné z https://www.intechopen.com/chapters/63906.

[11] HAYKIN, S. *Neural Networks and Learning Machines*. Upper Saddle River, New Jersey: Pearson Education, 2009. ISBN: 978-0-13-147139-9.

[12] SHARKAWY, A. Principle of Neural Network and Its Main Types: Review [online]. In: *Journal of Advances in Applied & Computational Mathematics*. Avanti Publishers, 2020. Dostupné z https://hal.univ-brest.fr/hal-03114305/document .

[13] BLAHA, M. Umělá inteligence [online]. In: HOLČÍK, J., KOMENDA, M. a kol. *Matematická biologie: e-learningová učebnice*. 1. vydání. Brno: Masarykova univerzita, 2015. ISBN 978-80-210-8095-9. Dostupné z https://portal.matematickabiologie.cz/.

[14] BROWNLEE, J. How to Choose an Activation Function for Deep Learning. In: *Machine Learning Mastery* [online]. Poslední změna 22.1.2021. [cit. 17.10.2022]. Dostupné z https://machinelearningmastery.com/choose-an-activation-function-for-deep-learning/.

[15] NIELSEN, M. A. *Neural Networks and Deep Learning* [online]. San Francisco, California: Determination Press, 2015 [cit. 23.10.2022]. Dostupné z http://neuralnetworksanddeeplearning.com.

[16] PAI, A. CNN vs. RNN vs. ANN – Analyzing 3 Types of Neural Networks in Deep Learning. In: *Analytics Vidhya* [online]. Poslední změna: 19.10.2020. [cit. 29.11.2022]. Dostupné z: https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/.

[17] LI, Z., LIU, F., YANG, W., PENG, S., ZHOU, J. A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects [online]. In: *IEEE Transactions on Neural Networks and Learning Systems*. IEEE, 2021. Dostupné z: https://doi.org/10.1109/TNNLS.2021.3084827.

[18] ALOYSIUS, N., GEETHA, M. A review on deep convolutional neural networks [online]. In: *International Conference on Communication and Signal*

Processing (ICCSP). IEEE, 2017. Dostupné z: https://doi.org/10.1109/ICCSP.2017.8286426.

[19] MCGONAGLE, J., WILLIAMS, C., Khim, J. Recurrent Neural Network. In: *Brilliant Wiki* [online]. [cit. 29.10.2022]. Dostupné z https://brilliant.org/wiki/recurrent-neural-network/.

[20] BAHETI, P. Activation Functions in Neural Networks [12 Types & Use Cases]. In: *V7 labs* [online]. Poslední změna 21.10.2022. [cit. 31.10.2022]. Dostupné z https://www.v7labs.com/blog/neural-networks-activation-functions.

[21] BROWNLEE, J. A Gentle Introduction to the Rectified Linear Unit (ReLU). In: *Machine Learning Mastery* [online]. Poslední změna 20.8.2020. [cit. 6.11.2022]. Dostupné z https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/.

[22] BROWNLEE, J. Softmax Activation Function with Python. In: *Machine Learning Mastery* [online]. Poslední změna 19.10.2020. [cit. 6.11.2022]. Dostupné z https://machinelearningmastery.com/softmax-activation-function-with-python/.

[23] RUDER, S. *An overview of gradient descent optimization algorithms* [online]. ArXiv, 2017. Dostupné z: https://doi.org/10.48550/arXiv.1609.04747.

[24] ELSKEN, T., METZEN, J. H., HUTTER, F. Neural Architecture Search: A Survey [online]. In: *Journal of Machine Learning Research 20*. ArXiv, 2019. Dostupné z: https://doi.org/10.48550/arXiv.1808.05377.

[25] *Stack Overflow Developer Survey 2022* [online]. Dostupné z: https://survey.stackoverflow.co/2022/.

[26] TensorFlow Basics [online]. In: *TensorFlow Guide*. 2022. Dostupné z: https://www.tensorflow.org/guide/basics.

[27] About Keras [online]. In: *Keras API docs*. 2022. Dostupné z: https://keras.io/about/.

[28] NumPy user guide [online]. In: *NumPy documentation*. 2023. Dostupné z: https://numpy.org/doc/stable/user/.

[29] XIAO, H., RASUL, K., VOLLGRAF, R. *Fashion MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms* [online]. ArXiv, 2017. Dostupné z: https://doi.org/10.48550/arXiv.1708.07747.

[30] DUA, D., GRAFF, C. Iris flower dataset [online]. In: *UCI Machine Learning Repository*. University of California, 2017. Dostupné z: https://archive.ics.uci.edu/ml/datasets/Iris.

[31] HORST, A. M., PRESMANES, A., GORMAN, K. B. *Palmer Archipelago (Antarctica) penguin data* [online]. 2020. DOI: 10.5281/zenodo.3960218. Dostupné z: https://doi.org/10.5281/zenodo.3960218.

[32] LECUN, Y., CORTES, C., BURGES, C. *MNIST handwritten digit database* [online]. 2010. Dostupné z: http://yann.lecun.com/exdb/mnist.

[33] HARRELL, F. E., CASON, T. *Titanic dataset* [online]. 2017. Dostupné z: https://www.openml.org/d/40945.

[34] RADOGLOU-GRAMMATIKIS, P., Kelli V., LAGKAS, T., ARGYRIOU, V., SARIGIANNIDIS, P. DNP3 Intrusion Detection Dataset [online]. In: *IEEE Dataport*. 2022. Dostupné z: https://dx.doi.org/10.21227/s7h0-b081.

[35] Configuration file parser documentation [online]. In: *Python documentation*. 2023. Dostupné z: https://docs.python.org/3/library/configparser.html.

[36] Losses [online]. In: *Keras API docs*. 2023. Dostupné z: https://keras.io/api/losses/.

[37] Layer activation functions [online]. In: *Keras API docs*. 2023. Dostupné z: https://keras.io/api/layers/activations/

[38] Optimizers [online]. In: *Keras API docs*. 2023. Dostupné z: https://keras.io/api/optimizers/

[39] Metrics and scoring: quantifying the quality of predictions [online]. In: *scikit-learn modules*. 2023. Dostupné z: https://scikit-learn.org/stable/modules/model_evaluation.html.

[40] Dense layer [online]. In: *Keras API docs*. 2023. Dostupné z: https://keras.io/api/layers/core_layers/dense/

[41] The Sequential model [online]. In: *Keras developer guides*. 2023. Dostupné z: https://keras.io/guides/sequential_model/.

[42] MaxPooling2D layer [online]. In: *Keras API docs*. 2023. Dostupné z: https://keras.io/api/layers/pooling_layers/max_pooling2d/.

[43] Conv2D layer [online]. In: *Keras API docs*. 2023. Dostupné z: https://keras.io/api/layers/convolution_layers/convolution2d/.

[44] Standard scaler documentation [online]. In: *scikit-learn modules*. 2023. Dostupné z: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html.

[45] One-hot encoder documentation [online]. In: *scikit-learn modules*. 2023. Dostupné z: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.OneHotEncoder.html.

[46] Built-in small datasets [online]. In: *Keras API docs*. 2023. Dostupné z: https://keras.io/api/datasets/.

[47] TensorFlow Datasets [online]. In: *TensorFlow Resources*. 2023. Dostupné z: https://www.tensorflow.org/datasets/catalog/overview.

[48] Datasets documentation [online]. In: *scikit-learn modules*. 2023. Dostupné z: https://scikit-learn.org/stable/modules/classes.html#module-sklearn.datasets.

[49] Pandas User Guide [online]. In: *pandas.pydata.org*. 2023. Dostupné z: https://pandas.pydata.org/docs/user_guide/index.html

Seznam symbolů a zkratek

MSE	Mean Squared Error
MAE	Mean Absolute Error
RMSE	Root Mean Squared Error
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
ReL	Rectified Linear Activation Function
ReLU	Rectified Linear Unit
API	Application Programming Interface
IDE	Integrated Development Environment
MNIST	Modified National Institute of Standards and Technology
DNP3	Distributed Network Protocol 3
csv	Comma-separated values

Odevzdané soubory

Veškeré soubory, zahrnující zdrojový kód vytvořeného nástroje a použité datasety, byly z důvodu jejich velikosti odevzdány přímo vedoucí práce na její pokyny. Soubory tak budou k dispozici po domluvě u vedoucí práce.