Středoškolská technika 2022

Setkání a prezentace prací středoškolských studentů na ČVUT

ALGORITMY HLUBOKÉHO UČENÍ A JEJICH VYUŽITÍ V JAZYCE PYTHON

Karel Bartůněk

Gymnázium, Milevsko, Masarykova 183

Gymnázium, Milevsko, Masarykova 183

ALGORITMY HLUBOKÉHO UČENÍ A JEJICH VYUŽITÍ V JAZYCE PYTHON

Maturitní práce z předmětu Informační a komunikační technologie

Karel Bartůněk

Informační a komunikační technologie

Prohlášení

Prohlašuji, že jsem tuto práci vypracoval samostatně a s použitím uvedené literatury a pramenů.

v Milevsku dne …….

Karel Bartůněk

Anotace

Práce se zabývá fenoménem hlubokého učení. Stručně jsem shrnul historii a vývoj různých metod strojového učení. Dále jsem podrobně popsal základní strukturu umělých neuronových sítí jako nejvýznamnější skupiny algoritmů hlubokého učení. Také jsem zmínil některé běžné nejlepší postupy při tvorbě takových algoritmů. Poznatky z této práce jsou následně předvedeny na ukázce ve frameworku Keras.

Abstract

This work looks into the phenomenon of deep learning. I have briefly summarized the history and development of various methods of machine learning. Then I have closely depicted the basic structure of artificial neural networks as the most significant group of deep learning algorithms. I have also mentioned some best practices for working with such algorithms. The contents of the work are presented on an example in the Keras framework.

Klíčová slova

hluboké učení, umělá inteligence, konvoluční neuronové sítě (CNN), shrnutí, Keras

Keywords

deep learning, artificial intelligence, convolutional neural network (CNN), summary, Keras

Poděkování

Děkuji vedoucímu práce Ing. Jiřímu Školníkovi, za podnětné rady, konstruktivní kritiku a osobní odborné konzultace. Dále děkuji mé rodině a přátelům za jejich ustavičnou podporu.

Obsah

1 Úvod

Algoritmy umělé inteligence (AI) mají za cíl řešit problémy, které počítač předem nezná, a to na základě sady obecných logických pravidel pro manipulaci s explicitními poznatky, kterými je algoritmus vybaven. Tento přístup umožňuje řešit mnohé úlohy, které můžeme jasně logicky definovat. Tyto algoritmy našly využití především v dynamickém řešení různých deskových her. Jednou z významných nevýhod těchto algoritmů je však fakt, že i tyto obecná pravidla umožňují řešit pouze úzkou skupinu problémů, kterým byla pravidla přispůsobena. Psaní takových pravidel je taktéž značně pracné, přičemž u složitých problémů by takový přístup vyžadoval vytvořit velké množství explicitních pravidel pro manipulaci s fakty; u některých problémů[1] se dokonce polemizuje, že přesná pravidla neexistují.[1]

Algoritmy strojového učení nám umožňují řešit problémy pouze z dostatečně velkého vzorku dat a očekávaných odpovědí k danému souboru. Tímto se vyhneme potřebě ručně programovat pravidla pro zpracování našich dat. Vlastnosti strojového učení mu umožňují zpracovávat rozsáhlé a složité datové soubory a extrahovat z nich dostatek přibližných pravidel, které následně můžeme uplatnit na jiný soubor dat. Místo standardního paradigmatu, kde programátor dodává pravidla, které jsou následně použita pro zpracování dat, jsou pravidla odvozována algoritmem strojového učení na základě dodaných dat.[1]

Algoritmy hlubokého učení, které jsou podmnožinou strojového učení, jsou založeny na přetvoření souboru dat vstupu na jejich stále obecnější reprezentace, které hledají stále obecnější vztahy mezi souborem dat a odpověďmi, případně mezi sebou. Tímto tyto algoritmy umožňují zpracování vysoce komplexních a velikých problémů, které bychom jinak řešili standardní statistickou analýzou velice složitě a nepřesně.[1]

1 Jako problémy neřešitelné klasickou umělou inteligencí se běžně uvádějí algoritmy počítačového vnímání, jako je například počítačové vidění.

Díky algoritmům hlubokého učení lze s dostatečnou přesností řešit složité problémy umělé inteligence způsoby, které nevyžadují přesné chápání a zapsání logického postupu, který je vyžadován ke vhodnému zpracování vstupu na výstup, který představuje naši odpověď. Nejen z těchto důvodů se tyto algoritmy v odvětví využívají pro kategorizaci umění, předvídání burzovních cen, nebo dokonce pro účely autonomních vozidel.[1]

Cílem této práce je čtenáře s některými těmito postupy seznámit, popsat jejich principy a zobrazit je na praktických příkladech za použití reálných dat, a to s ohledem na složitost a obsáhlost oboru. Není praktické se snažit popsat veškeré podrobnosti, které ohledně těchto algoritmů existují. Tato práce se proto zaměřuje na popis obecných znalostí, a to i za cenu ztrátu určité přesnosti.

2 Jazyk Python

V této práci se nachází ukázky kódu v programovacím jazyce Python, který je „dynamickým, objektově orientovaným programovacím jazykem, který se může využít v mnoha oblastech vývoje softwaru.“[2] Jako vysokoúrovňový interpretovaný jazyk nevyžaduje hluboké znalosti zařízení, na kterém bude kód spouštěn, naopak je kód v tomto jazyce lehce přenositelný a uplatnitelný napříč různými počítači bez velikých úprav. Díky svým vlastnostem a pravidlům nelimituje uživatele v použitých paradigmatech, umožňuje tedy například skloubit do jediného programu kód funkcionální i objektově orientovaný.

2.1 Framework Keras

Keras je modelová knihovna poskytující výkonné prvky pro algoritmy hlubokého učení. Tyto prvky umožňuje vkládat do modulárních uskupení, čímž je umožňuje libovolně kombinovat. Tato knihovna taktéž vyniká svým přívětivým uživatelským prostředím, které usnadňuje vývoj prototypů modelů, ale i produkčních algoritmů. Tato knihovna je vydána pod licencí MIT, která umožňuje i její komerční použití. [1]

2.2 Knihovna Tensorflow

Pro vykonání samotných tensorových operací potřebuje však knihovna Keras tzv. backend. Nejznámější z těchto knihoven je knihovna *Tensorflo*w vyvíjená společností Google. *Tensorflow* umožňuje trénování a práci na velkém množství počítacích zařízení, dokáže pracovat na jednotkách CPU i GPU s vysokou efektivitou. *Tensorflow* implementuje jak standardní konvoluční neuronové sítě, tak i různé druhy komplexních algoritmů, jako jsou rekurentní neuronové sítě[1], tak i jiné běžně užívané metody hlubokého učení jako například metodu podpůrných vektorů.[1]

3 Historie vývoje hlubokého učení

Historie hlubokého učení a metod neuronových sítí je silně spjata s vývojem algoritmů umělé inteligence a strojového učení. První myšlenky byly využity v různých výzkumech v teorii her již v 50. letech 20. století. Jako základní kámen moderních algoritmů učení se považuje objev algoritmu zpětného šíření několika nezávislými skupinami najednou.[2] Jako první úspěšné využití těchto sítí se počítá výzkum Yann LeCuna z Bellovy laboratoře, který využil ideje konvolučních neuronových sítí a algoritmů zpětného šíření, aby natrénoval síť umožňující automatizaci čtení amerických poštovních směrovacích čísel obálek. Nato síť, nazvaná LeNet, byla v 90. letech využita poštovní službou Spojených států amerických.[1]

1 Jedním z nejvýznamnějších příkladů rekurentních neuronových sítí je LSTM (Long Short Term Memory), jež je užívána při práci se sekvencemi.

2 Využití této metody pro trénování neuronových sítí bylo například představeno v článku David E. Rumelharta, Geoffrey E. Hinton a Ronald J. Williams *Learning representations by back-propagating errors* z 323. vydání prestižního časopisu *Nature*

V 90. letech byly standardní postupy nahrazeny postupy jádrových metod. Nejpopulárnější z těchto metod, metoda podpůrných vektorů (SVM), řeší klasifikační problémy nalezením rozhodovacích hranic v oboru dat. Těmito hranicemi lze dělit množinu dat do dvou různých oblastí odpovídajících dvěma kategoriím. Tento přístup umožňuje velice jednoduše klasifikovat nové body: stačí nám zjistil, na jaké straně rozhodovací hranice se naše nová data nacházejí. Tato metoda, zformulována Vladimírem Vapnikem a Corinnou Cortesovou v roce 1995, získala v oblasti jednoduchých klasifikačních úloh vysokou popularitu a díky své jednoduchosti a svému podložení matematickou teorií.[1]

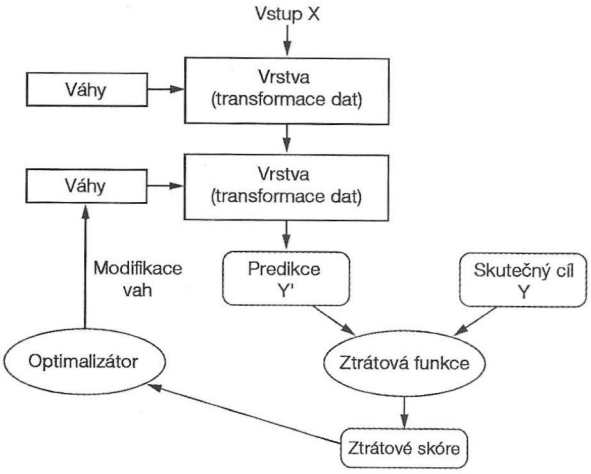
Tyto algoritmy, běžně označované jako algoritmy mělkého učení, jsou však nevhodné pro velké a komplexní datové množiny, které jsou potřebné pro problémy související s lidským vnímáním. Návrat hlubokých neuronových sítí zpět do popředí nastal až v roce 2012, kdy se tým Geoffreyho Hintona účastnil soutěže ImageNet[3] v klasifikaci barevných fotografií s úspěšností 74,3 %. Tímto způsobem se hluboké konvoluční neuronové sítě staly dominantní metodou využívanou v problémech klasifikace obrazů.[1]

3 Soutěž ImageNet je prestižní soutěž Stanfordské univerzity v klasifikaci barevných obrazů. v této soutěži je za cíl s co největší přesností automatizovat klasifikaci objektů na předložených fotografií.

4 Hluboké učení a neuronové sítě

Algoritmy hlubokého učení jsou obecně podmnožinou algoritmů strojového učení, které se snaží přiblížit fungování lidského mozku pomocí výpočetní techniky.[3] Z toho vyplývá, že takové algoritmy musí umět vyvozovat smysluplné závěry z neoznačených a neseřazených dat, přičemž tato vlastnost je žádoucí. Dle autora jsou nejvýznamnějšími algoritmy takové s umělými neuronovými sítěmi, tato práce se tudíž bude dále zabírat především těmito algoritmy.

Algoritmy neuronové sítě jsou tvořeny několika provázanými součástmi. Obrázek 1 naznačuje minimální strukturu pro umělou neuronovou síť:



Obrázek 1: Minimální struktura neuronové sítě [1]

Základem umělých neuronových sítí, tedy i algoritmů hlubokého učení, jsou různě zapojené *transformační vrstvy*, soustavy neuronových jednotek, které tvoří orientovaný graf, čímž vzniká obecnější reprezentace původních dat. Takovému grafu se běžněji říká *model neuronové sítě*. Ideálně natrénovaný model dokáže přetvořit libovolná vstupní data na reprezentaci stejnou očekávaným výstupním datům.[1]

Avšak takový model nedosahuje při svém vzniku požadovaných kvalit. Pro vytvoření modelu, který dokáže naše data správně zpracovat, se využívá procesu *trénování*. Tento proces je funkcí dvojic vstupních a výstupních dat, kdy modelu dodáme náhodný vzorek ze vstupních dat a pomocí něho přetvoříme tyto data na předpovědi výstupních dat.[1]

Dále tento předpoklad porovnáme s korespondujícími výstupními daty, a to tak, že vypočteme tzv. z*trátové skóre*, které představuje odchylku předpovědi od správného výstupu. k tomuto účelu je potřeba *ztrátové funkce*, která může být pouze malým doplněkem neuronové sítě, může být být taktéž vlastní neuronovou sítí, která je trénována pro rozpoznání rozdílů mezi výstupy a předpověďmi.[1]

Samotné „učení“ v algoritmu hlubokého učení je zastoupeno *optimalizátorem*, algoritmem, jehož účelem je úprava aktivity vrstev modelu, aby se model svými předpovědmi co nejvíce přibližoval skutečným výstupům. v sítí je tohoto docíleno přidáním posledního prvku, *vah vrstev*, které jsou přímými koeficienty jednotek vrstev. v takové konfiguraci optimalizátor přijímá ztrátové skóre a upravuje hodnoty vah, které tak přímo ovlivňují funkci vrstev.[1]

Úpravou vah je trénování dokončeno, v ideálním případě se při dalším průchodu učení předpoklad vrstev více přibližuje skutečnému výsledku než před učením. U skutečných aplikací hlubokého učení je však naším cílem získat model univerzální, tedy takový, který dokáže reagovat na různé data. V takovém případě se musí učení provádět mnohokrát na různorodých datech a musí být zvolit taková ztrátová funkce a takový optimalizátor, aby se model dostatečně přibližoval většímu množství výstupů.[1] Tehdy je dokonce nežádoucí, aby vrstvy byly přesně naučeny na určitý vstup, protože by tím model ztratil svou univerzálnost, správně by určil pouze výsledek pro ten vstup, na který byl trénován. Při takovém trénování je u reálných umělých sítí nutné omezit vliv jednoho trénovacího cyklu na váhy, jelikož by jinak každé další trénování přepisovalo původní hodnoty vah, což je pro trénování na více vzorcích nežádoucí.[1]

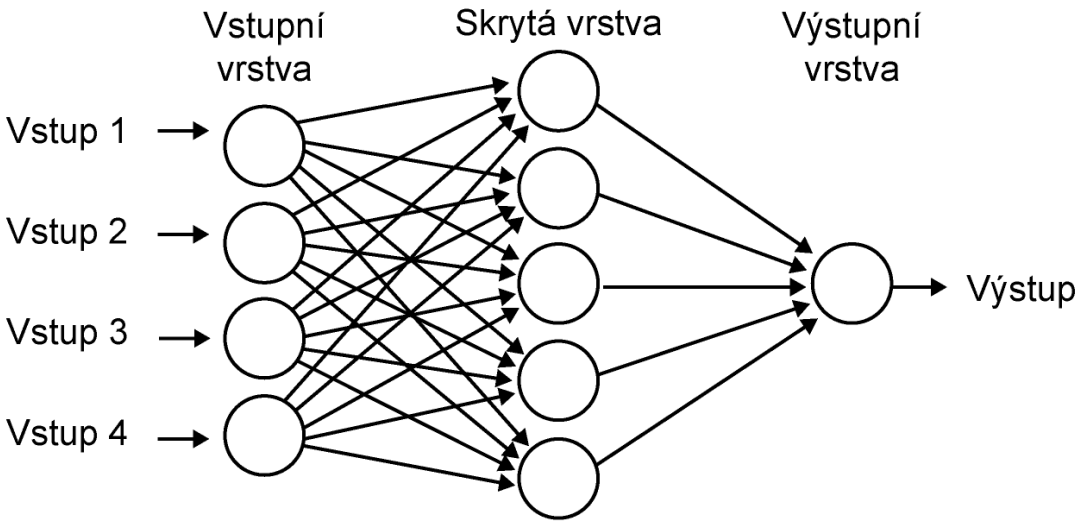
4.1 Dopředná umělá neuronová síť

Vrstvy algoritmů hlubokého učení fungují na základě principů umělých neuronových sítí. Pro účely našich algoritmů vytváříme matematické modely takových umělých neuronových sítí. Jako umělou neuronovou síť obvykle definujeme uspořádání jednotek

1 Při modelování umělých neuronových sítí se přizpůsobení sítě jedinému typu vstupu říká *přetrénování* a je pro tvorbu sítí silně nežádoucí. Většina sítí je tvořena s účelem užití na různorodých vstupech. Pro takové použití je potřeba z trénovacího vzorku přejmout pouze obrysy vzorku, a to i za cenu snížení přesnosti výstupu. Touto *generalizací* sítě předáme síti požadované vlastnosti, nikoliv pouze znalost vzorku, na kterém byla síť trénována.

(neuronů), které jsou tak navzájem spojeny, že mohou mezi sebou vhodně komunikovat. Pro sítě dopředné dále platí, že taková síť je acyklická a vzruch se vždy šíří směrem od vstupu k výstupu.[4]

Následující obrázek znázorňuje příklad takového uspořádání jednotek:



Obrázek 2: Uspořádání neuronů do vrstev v dopředné neuronové sítě. [4]

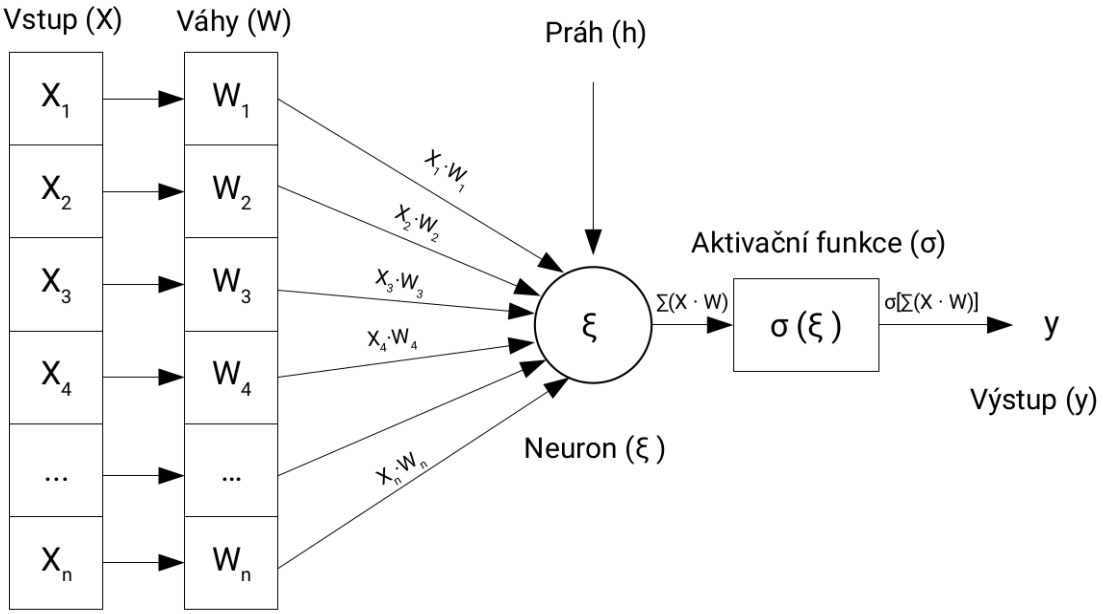
Jak z obrázku vidíme, vrstvy této sítě mohou být různě velké a mohou na sebe působit různými způsoby. Snad nejvýznamnějším poznatkem však je, že neurony v jedné vrstvě na sebe nijak nepůsobí, tudíž se nemohou nijak navzájem ovlivňovat.[1] Mohou vždy pouze aktivovat neurony další vrstvy.[4]

Výhody takové sítě oproti jiným přístupů, je především její masivní paralelnost, protože každá jednotka pracuje nezávisle na každé jiné ve stejné vrstvě. Této vlastnosti využijeme ve vlastních modelech tak, že budeme moci simulovat celou vrstvu naráz.[4]

4.1.1 Matematický model neuronové jednotky

Pro uplatnění modelu umělé neuronové sítě počítačem je nejdříve nutné vytvořit model, který bude představovat samotný neuron. Model jednoduchého neuronu si můžeme představit následujícím diagramem:

1 V praxi existují i modely, ve kterých je propojení jednotlivých neuronů omezeno, mezi takové propojení se například řadí propojení řídké, kde jsou spoje neuronů mezi vrstvami omezeny.



Obrázek 3: Model umělého neuronu [5]

Vstupem je vektor reálných čísel libovolné velikosti *X a* reálná prahová hodnota *h*. Váhy jsou vektorem reálných čísel *W*, ideálně o stejné velikosti jako je vektor *X*. Každý samostatný neuron má jeden jediný výstup, další reálné číslo *y*.[5]

Jak samotný obrázek napovídá, aktivaci jednotlivého neuronu lze získat pomocí následujícího výpočtu:

$$\xi = \sum_{i=1}^{n} w_i \cdot x_i - h$$

[5]

Kde w_i představuje prvky vektoru vah, x_i představuje prvky vektoru vstupu a *h* představuje prahovou hodnotu.

Po aplikaci aktivační funkce je výstup jednotky následující:

$$y = \sigma(\xi) = \sigma\left(\sum_{i=1}^{n} w_i \cdot x_i - h\right)$$

[5]

Kde σ představuje aktivační funkci, ξ aktivaci jednotky, w_i představuje prvky vektoru vah, x_i představuje prvky vektoru vstupu a h představuje prahovou hodnotu.

4.1.2 Aktivační funkce jednotek neuronových sítí

Důležitou součástí jednotky neuronové sítě je aktivační funkce, která přímo ovlivňuje chování samotné jednotky. Tato funkce přímo určuje vlastnosti jednotky. Tato funkce často slouží k velice specifickým účelům v celé neuronové síti. Dále následuje výčet běžně používaných aktivačních funkcí.[1]

4.1.2.1 Funkce sigmoid

Sigmoid je specifickou logistickou funkcí, která je svým tvarem zajímavá pro převod vstupu na hodnoty v intervalu mezi -1 a 1. Tato funkce se taktéž běžně používá ve statistice pro převod vstupních dat na hodnoty, které se dají představit procenty. Tímto je vhodná i pro modely neuronových sítí, kde je potřeba převést různorodá data do vhodného intervalu.[1]

4.1.2.2 Logaritmická funkce

V algoritmech hlubokého učení se běžně používají i funkce logaritmické. Tato funkce je logaritmem funkce sigmoidu. Tato funkce je často užívána místo samotného sigmoidu, přestože je její výstupem číslo v intervalu mezi -∞ a 0. Používána je především kvůli matematickým zjednodušením, které použití logaritmu umožňuje a kvůli své známé derivaci, která výrazně zlehčuje proces optimalizace.[6]

4.1.2.3 Funkce ReLu

Funkce ReLU se běžně používá pro konvoluční neuronové sítě, především pro takové, kde chceme získat kladné hodnoty aktivace sítě. Je definována následovně:

$$\sigma_{ReLU}(\xi) = max(0, \xi)$$

[1]

Z tohoto vzorce vyplývá, že tato funkce je nelineární a zalamuje jakékoliv negativní hodnoty na nulu. Proto se používá v případech, kdy chceme, aby aktivace dané vrstvy byly vždy kladné.[1]

4.2 Backpropagace neuronové sítě optimalizátorem

Optimalizátor je jednou s nejdůležitějších součástí umělé neuronové sítě. Vhodně upravuje váhy, čímž upravuje aktivaci jednotlivých neuronů, proto umožňuje přizpůsobit chování sítě trénovaným vstupům. Optimalizátor z těchto důvodů zajišťuje přenos znaků ze vstupních dat na neuronovou síť, čímž dodává sítí její vlastností.[1]

4.3 Stochastický gradientní sestup (SGD)

Tento nejzákladnější a nejdůležitější optimalizátor aplikuje matematické metody derivace pro přibližné zjištění minima ztrátové funkce. Pro úplnou optimalizaci, tedy nalezení hodnot vah, které korespondují k nulové hodnotě ztrátové funkce, je možné najít řešení vyřešením rovnice pro hodnotu vah, kde se gradient ztrátové funkce rovná nule. Tato funkce se však s rostoucí velikostí vah stává neřešitelnou. [1]

4.3.1 Minidávkový stochastický gradientní sestup

Pro využití SGD se využívá Newtonovy metody pro hodnoty vah v dávkách V praxi to znamená, že pro každou dávku dat spočítáme predikce naší neuronové sítě, pro které spolu s předpokládanými hodnotami přepočteme hodnotu *ztrátové funkce*. Dalším krokem je výpočet *gradientu* této ztráty s ohledem na váhy. Tomuto kroku se běžně říká *zpětný průchod*. Posledním krokem je úprava vah, kdy se od jednotlivých hodnot vah násobek tohoto gradientu s *konstantou sestupového kroku*. Tato konstanta, taktéž běžně nazývána *rychlost učení,* určuje rychlost, se kterou se bude predikce gradientu měnit. Při nevhodně nízkých hodnotách bude potřeba nadměrného množství odhadů, což výrazně zpomalí trénovací proces, při příliš vysokých hodnotách bude tento odhad náhodně kolísat, což může znemožnit úspěšnou optimalizaci.[1]

4.3.2 Adagrad

Adagrad je optimalizačním algoritmem vycházející z SGD, který umožňuje při procesu trénování dynamicky měnit konstantu sestupového kroku pro každý parametr vah. Tato konstanta se dynamicky mění v čase pro každý parametr vah, a to odečtem podílu obecné rychlosti učení a odmocniny součtu druhých mocnin předchozích gradientů a malého čísla, které zamezuje dělení nulou. [7]

Největším problémem algoritmu Adagrad je nastřádání těchto druhých mocnin gradientů, které po čase sníží rychlost učení na astronomicky malé číslo, čímž je učení uměle ukončeno.[7]

4.3.3 RMSprop a Adadelta

RMSprop a Adadelta jsou dva nezávisle na sobě objevené metody redukce agresivnosti Adagradu. Oba algoritmy spočívají v postupné redukci velikosti součtu druhých mocnin předchozích gradientů. RMSprop, jednodušší a běžněji používaný z těchto dvou, využívá konstanty γ, která pro své hodnoty v otevřeném intervalu mezi 0 a 1 snižuje součinem hodnotu součtu druhých mocnin předchozích gradientů.[7]

4.4 Ztrátové funkce neuronové sítě

Ztrátové funkce umožňují sledovat přesnost předpovědi výstupu oproti správnému výstupu. Tato funkce tudíž musí umět přesně a vhodně určovat chyby mezi 2 výstupy. Mezi nejdůležitější ztrátové funkce patří střední kvadratická chyba (MSE), která vrací druhou mocninu čistého rozdílu a křížová entropie, jež se používá při kategorizaci a klasifikaci. [1]

5 Algoritmy hlubokého učení v praxi

5.1 Konstrukce příznaků

V praxi je běžné, že budeme chtít použít umělou neuronovou síť k řešení problému, o kterém již něco víme. Vyšetřujeme-li například sled znaků (větu) jako vstup, můžeme tento problém zjednodušit úvahou a znalostí, že daný vstup bude pravděpodobně užívat běžných slov jazyka a útvaru, ze kterého vstup pochází. Tudíž místo například podávání jednotlivých písmen můžeme externím lingvistickým algoritmem, který danou větu rozdělí na lingvistické celky, jež se teprve stanou vstupem sítě.[1]

Takový postup umožňuje zjednodušit a zrychlit trénování námi požadované sítě odebráním komplexity vstupu. Očividnou nevýhodou tohoto postupu je vytváření fixních předpokladů lidskou rukou, které mohou být nepřesně, či chybné.[1] v našem příkladě se vystavujeme možnosti, že nějaký vstup bude zapsán chybným pravopisem nebo bude používat nestandardní slovní zásobu. Proto je v mnohých případech vhodnější použít automatické metody, které umožňují takové síti přejímat znalosti o vnitřních vzorech vstupu. Jednou z nejdůležitějších takových metod je metoda konvoluce.

5.2 Konvoluce

Konvoluce umožňuje modelu se naučit lokální vzory vstupu, které následně dokáže aplikovat na různé místa vstupu. Říká se, že takové vzory jsou *invariantní*, jelikož je takový model umí uplatnit nezávisle na pozici ve vstupu. Konvoluce typicky používá vícerozměrné *tensory*[1], nazývané *mapy příznaků*.[1]

1 Tensory jsou datové struktury s daným množství os a danými rozměry, přičemž do dosazení libovolného čísla mezi 0 a rozměrem dané osy pro každou osu tensoru existuje v takovém místě

Konvoluce hledá malé podobné oblasti (tzv. *záplat*) ve vstupu, aplikuje na ně stejnou transformaci a zapíše je do mapy výstupních příznaků. Takto vznikají filtry, které korespondují s určitými vlastnostmi vstupu. Takto vznikne pro každý filtr *mapa odpovědí*, která určuje odezvu filtru v závislosti na pozici jeho umístění ve vstupu.[1]

Samotný krok konvoluce prochází vstupem po skocích o velikosti *záplat* a vytváří mapu odpovědí okolních příznaků. Tato mapa je dále jádrovým trikem transformována na mapu odpovědí.[1]

Konvoluce se v praxi užívá především na takových datech, které jsou tvořeny spoustou malých vzorů, které se ve vstupu opakují. Díky těmto vlastnostem jsou sítě obsahující konvoluci vhodné pro zpracovávání obrazového materiálu, jelikož ten se zpravidla skládá z primitivních vzorů, které dokáže konvoluce zachytit a třídit.

5.3 Dávkování dat

Trénování mnohovrstvé neuronové sítě je výpočetně složitým procesem. Pro zamezení přeučení je kritickým požadavkem trénovat takovou síť rovnoměrně náhodně, a i při takovém trénování není jakýkoliv úspěch garantován. Proto se pro běžně používané algoritmy hlubokého učení používá metoda dávkování.[1]

Metoda dávkování dat spočívá v rozdělení trénovaných dat na dávky, dostatečně malé skupiny vstupů. Dále pro každou dávku probíhá trénování normálně, jenom s tím rozdílem, že optimalizátor nemění váhy vrstev přímo. Místo toho jsou změny vah pro všechny vstupy v dávce načítány do samostatného vektoru, který je až po dokončení dávky aplikován na samotné váhy. [1]

Výraznou výhodou tohoto postupu je, že nám umožní trénovat s méně širokou doménou dat, jelikož takto každý jednotlivý vstup ztrácí schopnost se výrazně otisknout na hodnotách vah. Tímto trénovaná síť získává na generalizovanosti, která je zde žádanou vlastností.[1]

datový bod. Nejznámějším příkladem tensoru je matice o dvou rozměrech.

5.4 Omezení podučení

Nejtěžším problémem trénování neuronové sítě je zamezení jejímu podučení a přeučení. Podučení vykazuje stav, kdy síť nemá dostatečné znalosti pro přesné přiblížení očekávanému výstupu. Podučení nastává především při nedostatečném množství vstupních dat, při nevhodném dávkování dat nebo při nedostatečném množství trénovacích cyklů. Podučení se typicky řeší trénováním sítě ve více epochách, tedy vícekrát na stejných datech. Tato metoda umožňuje silnější přiblížení vstupu, avšak zvyšuje riziko přeučení. Dále je možné zmenšit velikost dávek, což zamezí přílišné generalizaci trénování. Nevýhodou tohoto přístupu je silné zpomalení trénování. [1]

5.4.1 Rozšíření dat

Často je potřeba trénovat neuronovou síť na malém množství dat, které vždy nemusí být dostačující pro dostatečné natrénování sítě, protože dojde k jejímu podučení. Z tohoto důvodu se stala technika rozšíření dat vysoce populární. Vychází podobně jako konstrukce příznaků z předchozích znalostí vstupu. Při tomto postupu hledáme znaky, které lze na vstupu pozměnit, aniž bychom změnili očekávaný výstup. Tímto vytváříme nové iterace vstupních dat a zvyšujeme přesnost trénování. Tato technika se uplatňuje především při klasifikaci obrazů, kde lze tento postup uplatňovat grafickými manipulacemi s obrazem. Mezi nejběžnější takové úpravy patří rotace obrazu, jeho přiblížení, posunutí a mírné změny barev.[1]

5.5 Zamezení přeučení

Ne-li silnějším problémem než podučení neuronové sítě je její přeučení. Přeučení je stav, kdy je síť silně přizpůsobena trénovaným datům, avšak úplně ztratila schopnost generalizace, tedy taková síť není schopna předpovědět správné výstupy od jiných než trénovaných vstupů. Existuje několik postupů omezení přeučení, při jejich aplikaci je však potřeba dbát, abychom přílišně neomezili schopnost sítě se učit.[1]

5.5.1 Regulace vah

Regulace vah ve vrstvách sítě umožňuje umírnění trénování a zamezuje přeučení nevhodnými nesprávnými vstupy. Regulace zvyšuje náklady na změnu vah a tím omezuje složitost transformací. Běžně se používá L1 regulace, přidávající náklady úměrné absolutní hodnotě stávajících hodnot vah, a L2 regulace, kde jsou tyto náklady přidávány o druhou mocninu stávajících hodnot.[1]

5.5.2 Redukce velikosti sítě

Při použití velkých neuronových sítí se může stát, že se doména vstupu může stát menší než nejširší vrstva naší sítě. V tomto případě se vstup přímo přepisuje do vrstev, přičemž nedochází k jakékoliv generalizaci sítě. Toto chování je ve většině případů nežádoucí, jelikož takové trénování neučí jakékoliv obecné znaky vstupu. Pro opětovné nabytí vhodných vlastností je proto nutné snížit velikost vrstev, abychom způsobili vhodnou transformaci vstupu.[1]

5.5.3 Výpadek

Výpadek umožňuje náhodné vynechání skupiny příznaků vrstev při trénování. Toto náhodné vynechávání umožňuje odstraňovat malé nevýznamné vzory a zvyšuje váhu významných primárních znaků, které v síti zůstanou i po zavedení tohoto náhodného šumu.[1]

6 Ukázka programu hlubokého učení

Tato část práce si klade za cíl představit práci s frameworkem Keras a knihovnou Tensorflow na ukázkovém problému řešeného algoritmem hlubokého učení. Zvolený program využívá veřejně dostupných dat k výrobě algoritmu, který dokáže s vysokou přesností rozeznat typické tiskací písmo z černobílého obrazu. Problém rozeznání znaků psaného písma je typickým klasifikačním problémem, kde konvoluční neuronové sítě vynikají, a proto byl právě tento problém pro účely této práce zvolen.

6.1 Získání vhodných dat

Získání vhodných dat a jejich příprava je mnohdy jednou z nejsložitějších součástí procesu tvorby algoritmu hlubokého učení. Je nutné získat data v jednotné podobě, aby je bylo možné statisticky porovnávat. Data musí být podobné velikosti, podobného stylu i podobného formátu, přičemž musí být jednotně ohodnoceny štítky, tedy očekávanými výstupy. Dále je nutné omezit množství špatně namapovaných vstupů, tedy vstupu, kterým je chybně přiřazen nesprávný validační výstup. Pro účely této práce použijeme předem připravenou datovou sadu Znaků tiskacího písma.[1] Tato předem připravená sada obsahuje přes tři tisíce vzorků písmen a číslic, které jsou již pro nás roztříděny do vhodných kategorií.

Konkrétně se tato datová sada skládá ze složky obsahující samotné vstupní data ve formátu PNG a ze souboru CSV, který ke každému obrázku přiřazuje správný znak. Ke zpracování souboru se znaky jsem použil knihovnu Pandas, která umožňuje rychlé zpracování těchto souborů do tabulkové formy.

6.2 Poskytnutí dat počítači

V přiloženém kódu jsme přípravu dat do sítě vyřešili vytvořením vlastní třídy `HandwritingDataGenerator`, která zajišťuje celkové načtení dat a jejich přípravu do datového souboru Tensorflow, který můžeme přímo použít pro trénování.

Náš datový generátor přijímá jako vstup pouze souborové lokace souboru štítků a složky s obrázky, přičemž sám řeší pomocí funkcí knihovny Tensorflow konverzi obrázků na Tensorflow Tensory, jejich normalizaci na desetinná čísla v intervalu 0 až 1 a grafickou úpravu vstupu, v našem případě dostatečné snížení rozlišení vstupu.

1 Tato datová sada je svobodně dostupná pod licencí OdbL, například na webových stránkách https://www.kaggle.com/datasets/dhruvildave/english-handwritten-characters-dataset

Dále formuje manuálně vytvořené štítky na výstupní tensory o velikosti počtu znaků. Výstup je kódován metodou *one-hot*, která vytváří prázdné (nulové) tensory o velikosti množství znaků, které však na pozici očekávaného vstupu mají maximální hodnotu, v našem případě číslo 1. Tento postup je pro tento problém vhodný, jelikož klasifikujeme malé množství štítků (v datové sadě je celkem 62 znaků) a očekáváme, že v jednom vstupu bude jen jediný znak písma. Každé pozici ve výsledném tensoru v takové sestavě přidělujeme jeden znak.

Konečně tato třída vytváří třídy `tensorflow.Dataset`, které dynamicky generují dvojice vstupu a výstupu tak, abychom minimalizovali množství použité systémové paměti. Pro trénování neuronových sítí totiž standardně knihovna Tensorflow vyžaduje poskytnutí tensorů celkové datové sady, což by prudce zvýšilo paměťovou složitost učení a zvýšilo pořizovací náklady algoritmu.

6.3 Stavba modelu neuronové sítě

Díky frameworku Keras můžeme zjednodušit výstavbu vrstev umělé neuronové sítě na poskládání jednotlivých dílů do modelové třídy. Náš model umělé neuronové sítě bude sekvenční sítí skládanou z dvou hlavních částí: skupiny konvolučních 2D transformací a finální klasifikační třídy.

První část je tvořena třemi skupinami, kde je vždy operace konvoluce následována operací sdružování dle maxima. Sdružování dle maxima je operace podobná konvoluci, kde avšak není použita trénovaná lineární transformace, ale zabudovaná operace tensorového maxima. Výhodou tohoto postupu je mnohonásobně silnější extrakce prostorových znaků obrazu a zmenšení množství výstupních neuronů.[1]

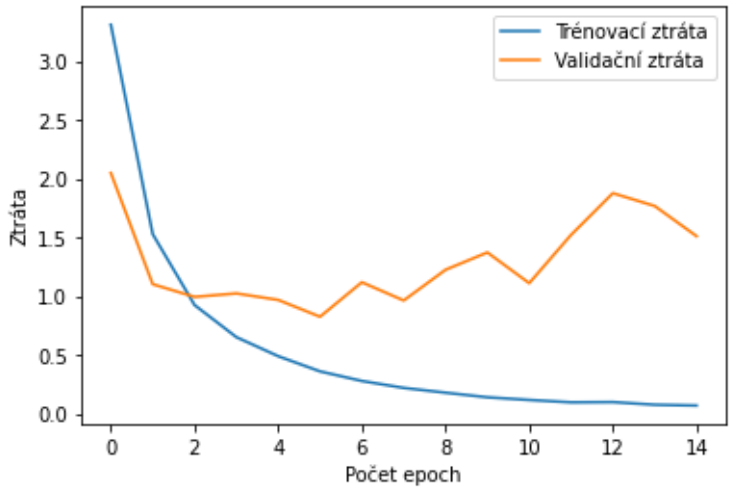
Druhá část je tvořena dvěma hustě propojenými sítěmi, jednou o velikosti čtvrtiny výstupu první skupiny a poslední vrstvou o přesné velikosti výstupního tensoru, které mají za cíl přesně napodobovat mapu pravděpodobností výskytu jednotlivých psaných znaků.

Jako aktivační funkce sítě byla zvolena ve všech vrstvách sítě vyjma poslední klasifikační vrstvy funkce *ReLu*, a to především vzhledem k formátu vstupu a výstupu, protože obě množiny jsou tvořeny tensory kladných čísel. Pro poslední klasifikační vrstvu byla použita aktivační funkce *softmax*, která je uzpůsobena pro výpočet pravděpodobností znaků v poli.[1]

Jako ztrátová funkce byla použita křížová entropie, jež je vhodná pro kategorizaci vstupu. Jako optimalizátor byla pro svou výkonnost, vysokou účinnost v problémech počítačového vidění a zvolena funkce *RMSprop*.[1] Následně je tento model s těmito parametry zkompilován.

V závěru samotného kódu vytvoříme instanci našeho generátoru dat a sdělíme knihovně Tensorflow, aby započala trénovaní sítě s trénovacími daty a síť pravidelně validoval validačním vzorkem. Tato operace je opakována tolikrát, kolik jsme nastavili trénovacímu algoritmu epoch. Nyní je již síť natrénována a do proměnné se pro každou epochu uložila validační ztráta a přesnost.

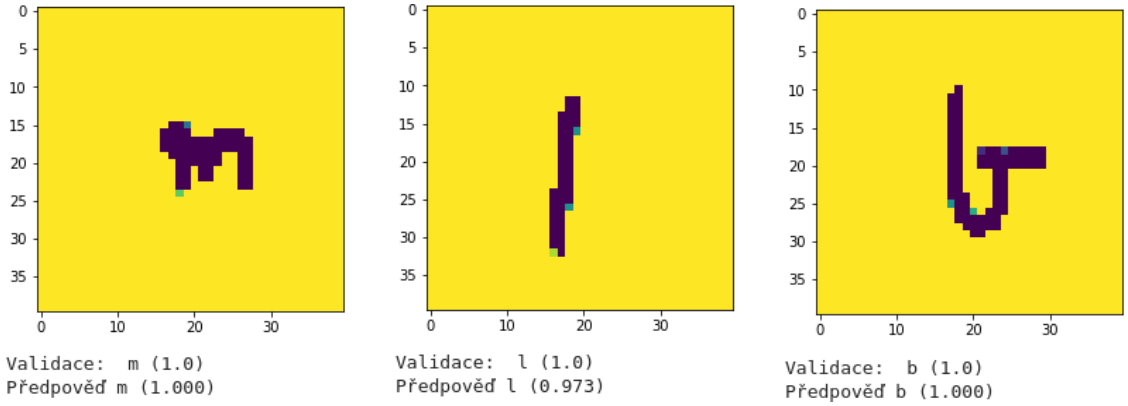
Graf validační ztráty této sítě naznačuje, že již po třetí etapě trénování dochází k přeučení a síť již nezískává na přesnosti. Můžeme tedy omezit trénovací cyklus na 3 epochy bez ztráty funkčnosti. Po přibližně páté epoše dokonce dochází i ke zhoršení výkonu sítě na validačních datech. Tento úkaz je jedním z důvodů uplatnění různých metod zabránění přeučení jako je například výpadek. Ten zde již nebyl použit, jelikož i bez něho dosahuje síť požadovaných výsledků a přidání výpadku by zásadně zpomalilo proces učení.



Obrázek 4: Graf trénovací a validační ztráty sítě po několika epochách

6.4 Shrnutí ukázky

Po natrénování sítě můžeme dokázat schopnost sítě aplikací na data, na kterých nebyla síť trénována. Síť nyní dokáže od sebe rozeznat s přesností až 77 % různé číslice a písmena, dokonce i v případech, které by běžnému čtenáři způsobily problémy. Tímto krátkým a jednoduchým kódem jsme vytvořili model konkurující i některým standardním. Podobné algoritmy jsou proto hojně používány jako algoritmy optického čtení znaků neboli OCR. Algoritmy s konvolučními sítěmi se běžně používají i v profesionálních řešeních, a to pro především pro svou vysokou efektivitu a adaptabilitu novému textu, protože je vždy možné takovou síť jemně přetrénovat pro libovolný podobný typ písma a znaků.[8]



Obrázek 5: Ukázky rozeznání těžce člověkem rozeznatelných znaků

7 Závěr

Seznámil jsem čtenáře s algoritmy hlubokého učení, jejich historií, jejich součástmi a jejich některými druhy. Dále jsem předvedl nabyté znalosti na ukázce takového algoritmu ve frameworku Keras v programovacím jazyce Python. Popsal jsem také matematický model funkce umělého neuronu a jeho zapojení to vrstev umělé neuronové sítě. Popsal jsem princip Stochastického gradientního sestupu a jeho význam ve světě strojového učení. Dále jsem zmínil některé z nejvýraznějších součástí algoritmů hlubokého učení a nastínil jejich užití. Na závěr jsem zmínil některé běžně užívané postupy při tvorbě a správě takových algoritmů.

Cílem této práce nebylo vyčerpávajícím způsobem popsat veškeré zákoutí této vědy, ale pouze seznámit potenciální zájemce se základy tohoto oboru takovým způsobem, aby mohl své nabyté znalosti použit ke vlastnímu výzkumu a vlastní práci. Existuje však mnoho dalších zajímavých skupin algoritmů, které zde nebyly vůbec zmíněny, především rekurentní neuronové sítě (RNN) užívané pro práci se sekvencemi dat a generativní soupeřivé sítě (GAN) používané pro umělecké účely a pro detekci falešných předmětů.

Případný návazný výzkum by se měl soustředit především na rozšíření znalostí právě o tyto a další typy sítí, které dle mého názoru získaly v poslední době silného ohlasu, doplnění o další algoritmy strojového učení, které by přinesly širší rozhled a dodaly širší kontext některým pasážím této práce. Další výzkum by se měl také zaměřit na vlastní tvorbu algoritmů a řešení komplexnějších a složitějších problémů na větších datových sadách.

Tabulka obrázků

Seznam použité literatury

[1]: CHOLLET, François. Deep learning v jazyku Python: knihovny Keras, Tensorflow Přeložil Rudolf PECINOVSKÝ. PrahaGrada Publishing, 2019. ISBN 9788024731001.

[2]: vinil, PyCZ: Programovací jazyk Python, 2019, pořízeno dne 13. 1. 2022 (dostupné online: https://www.py.cz/FrontPage)

[3]: Petr Kopic, Hluboké genetické programování, 2020, Univerzita Pardubice, diplomová práce

[4]: Ing. Milan Blaha, Ph.D., Koncept umělé neuronové sítě, 2016, pořízeno dne 20. 3. 2022 (dostupné online: https://portal.matematickabiologie.cz/index.php?pg=analyza-a-hodnoceni-biologickych-dat--umela-inteligence--neuronove-site-jednotlivy-neuron--uvod-do-neuronovych-siti--koncept-umele-neuronove-site)

[5]: Ing. Milan Blaha, Ph.D., Matematický model a aktivní dynamika neuronu, 2016, pořízeno dne 20. 3. 2022 (dostupné online: https://portal.matematickabiologie.cz/index.php?pg=analyza-a-hodnoceni-biologickych-dat--umela-inteligence--neuronove-site-jednotlivy-neuron--jednotlivy-neuron--matematicky-model-a-aktivni-dynamika-neuronu)

[6]: Serengil, Sefik Ilkin, Logarithm of Sigmoid As a Neural Networks Activation Function, 2017, pořízeno dne 7. 4. 2022 (dostupné online: https://sefiks.com/2017/12/09/logarithm-of-sigmoid-as-a-neural-networks-activation-function/)

[7]: Sebastian Ruder, An overview of gradient descent optimization algorithms, 2017, pořízeno dne 20.3.2022 (dostupné online: https://ruder.io/optimizing-gradient-descent/)

[8]: Tim Keary, AI and OCR: How optical character recognition is being revitalised , 2019, pořízeno dne 7. 4. 2022 (dostupné online: https://www.information-age.com/optical-character-recognition-tools-ocr-ai-123479324/)

[9]: Goodfellow, Ian; Pouget-Abadie, Jean; Mirza, Mehdi; Xu, Bing; Warde-Farley, David; Ozair, Sherjil; Courville, Aaron; Bengio, Yoshua, Generative Adversarial Nets, 2014

Příloha č. 1: Zdrojový kód ukázky z 6. kapitoly

```
import numpy as np # Matematické funkce a množiny
import os # Knihovna systému pro práci se složkami
import pandas as pd # Zpracování dat
import tensorflow as tf
import tensorflow.keras as Keras
import keras.layers as layers # Vrstvy modelu
import matplotlib.pyplot as plt # Zobrazení dat na grafech

EPOCHS = 15 # Počet epoch trénování
BATCH_SIZE = 4 # Velikost jedné dávky SDG
IMAGESIZE = [40,40] # Velikost obrazu pro síť

class HandwritingDataGenerator:
  def __init__(self,csv_path, directory_path, labels_table,
validation_split=0.1,batch_size=8,epochs=5):
    self.csv_path = csv_path
    self.directory = directory_path
    self.labels_table = labels_table
    self.validation_split = validation_split
    self.labels = pd.read_csv(csv_path)
    self.labels = self.labels.set_index("image",drop=True)
    # Indexujeme podle jména souboru
    self.file_list = os.listdir(self.directory)
    np.random.shuffle(self.file_list)
    # Prvotní promíchání seřazeného souborového listu

  def _one_hot(self,ind):
    """Vytvoří one-hot tensor"""
    arr = np.zeros(shape=(len(self.labelslist)),
                   dtype=np.float32)
                   #vzniká [0,0,0,...,počet_znaků]
    arr[ind] = 1.0
    return arr
  def _get_one_sample(self,file_name,rotate=0):
    """Vytvoří jednu trénovací množinu (x,y)"""
```

```
    img = tf.io.read_file(self.directory+file_name)
    # Načteme soubor obrázku
    img = tf.io.decode_image(img,
                channels=1,
                dtype=tf.dtypes.float32)
    # Necháme TF převést obrázek na tensor
    img = tf.image.resize(img, IMAGESIZE)
    label_str = self.labels.at["Img/"+file_name,"label"]
     # Vydá jeden one-hot štítek, např. E
    label = self._one_hot(self.labels_table.index(
                              str(label_str)))
    yield (img,label)

  def generate_train_data(self):
    """Generátor trénovacích dat"""
    training_files =
self.file_list[int(self.validation_split*len(self.file_list)):]
    np.random.shuffle(training_files)
    for file_name in training_files:
      yield self._get_one_sample(file_name)

  def generate_validation_data():
    """Generátor validačních dat"""
    validation_files =
self.file_list[:int(self.validation_split*len(self.file_list))-1]
    np.random.shuffle(validation_files)
    for file_name in validation_files:
      yield self._get_one_sample(file_name)

  def get_dataset(self,generator):
    """Vytvoří dataset Tensorflow s globálním dávkováním"""
    return tf.data.Dataset.from_generator(generator,
output_signature=(tf.TensorSpec([*IMAGESIZE,1]),
tf.TensorSpec((int(self.labels.nunique())))))).batch(BATCH_SIZE)

model = Keras.Sequential() #Sekvenční model
# 1. Konvoluce
model.add(layers.Conv2D(32,(3,3),activation="relu",
```

```
input_shape = (*IMAGESIZE,1)))          # Vstup: (50,50,1)
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(64,(3,3),activation="relu"))
model.add(layers.MaxPooling2D((2,2)))
model.add(layers.Conv2D(128,(3,3),activation="relu"))
model.add(layers.MaxPooling2D((2,2)))
# 2. Zploštění
model.add(layers.Flatten())
model.add(layers.Dense(300,activation="relu"))
model.add(layers.Dense(62,activation="softmax"))
                                    # Poslední vrstva: (64,)

# Kompilace modelu s optimalizátorem a ztrátovou funkcí
model.compile(optimizer="rmsprop",
              loss="categorical_crossentropy",
              metrics=["accuracy"])

labels =
"0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ"
# Všechny naše znaky
datagen =
HandwritingDataGenerator("english.csv","./Img/",labels,validation_spli
t=0.1,batch_size=8,epochs=5)


# Trénování sítě
history = model.fit(datagen.get_dataset(
                                    datagen.generate_train_data),
                    epochs=EPOCHS,
                    validation_data = datagen.get_dataset(
                              datagen.generate_validation_data))
```