Vyšší odborná škola a Střední průmyslová
škola elektrotechnická Olomouc,
Božetěchova 3

PRAKTICKÁ ZKOUŠKA Z ODBORNÝCH PŘEDMĚTŮ

Neuronové sítě

Autor **Jindřich Machka**
Obor **Elektrotechnika**
Vedoucí práce **Mgr. Jaroslav Krbec**
Školní rok **2022/2023**

Vyšší odborná škola a
Střední průmyslová škola elektrotechnická
Božetěchova 3, 772 00 Olomouc

ZADÁNÍ PRAKTICKÉ ZKOUŠKY Z ODBORNÝCH PŘEDMĚTŮ

Jméno žáka: **Jindřich Machka**

Obor vzdělání: **26-41-M/01 Elektrotechnika**

Třída: **4A**

Ředitelství Vyšší odborné školy a Střední průmyslové školy elektrotechnické Olomouc Vám podle vyhlášky Ministerstva školství, mládeže a tělovýchovy č. 177/2009 Sb., o bližších podmínkách ukončování vzdělávání ve středních školách maturitní zkouškou, ve znění vyhlášky č. 90/2010 Sb., vyhlášky č. 274/2010 Sb., vyhlášky č. 54/2011 Sb. a vyhlášky č. 273/2011 Sb., určuje tuto praktickou zkoušku z odborných předmětů.

Téma: **Neuronové sítě**

Způsob zpracování a pokyny k obsahu:

- Popište technologii neuronových sítí.
- Zpracujte charakteristiku nástroje Tensorflow.
- Porovnejte tvorbu v Tensorflow s implementací vlastní neuronové sítě.
- Zdokumentujte vlastní aplikaci vybrané technologie.
- Vytvořte poster (formát PDF, velikost A1) prezentující maturitní práci.
- Připravte podklady a aktivně se zúčastněte jednotlivých kol SOČ.
- Všechny body zadání jsou závazné, při jejich neplnění může škola změnit formu praktické zkoušky z dlouhodobé na jednodenní.

Rozsah: 25 až 35 stran

Kritéria hodnocení: Hodnocení práce probíhá ve třech fázích.

Průběžné hodnocení zohledňuje postupné plnění zadaných úkolů, dodržování termínů, míru samostatnosti žáka. Hodnocení závěrečné posuzuje míru splnění všech požadavků vyplývajících ze zadání práce a funkčnost produktů. Hodnocena je přehlednost, úplnost, srozumitelnost a formální stránka textové části práce. Hodnocení obhajoby práce zahrnuje způsob a srozumitelnost projevu, vzhled prezentace, odpovědi na dotazy.

Délka obhajoby: 15 minut

Odevzdání: 1 x PDF verze (Thesaurus) + 1x poster (Thesaurus)

Vedoucí práce: Mgr. Jaroslav Krbec **Oponent práce:** Ing. Václav Křižan

Datum zadání: 10. října 2022 Datum odevzdání: 20. března 2023

V Olomouci dne 10. října 2022

VYŠŠÍ ODBORNÁ ŠKOLA A STŘEDNÍ PRŮMYSLOVÁ ŠKOLA ELEKTROTECHNICKÁ
Božetěchova 755/3, 772 00 Olomouc
tel. 585 208 121 IČ: 00844012

.....
ředitel školy

Zadání převzal dne 19.10.2022

.....
podpis žáka

Prohlašuji, že jsem praktickou zkoušku z odborných předmětů vypracoval samostatně a všechny prameny jsem uvedl v seznamu použité literatury.

……

Jindřich Machka

Chtěl bych vyslovit poděkování panu Mgr. Jaroslav Krbec za odborné konzultace a poskytnuté informace, panu Mgr. Jíří Hátle Ph. D. za odbornou konzultaci ohledně matematické části mé práce a vedení školy za umožnění úpravy jednoho bodu zadání.

……

Jindřich Machka

Prohlašuji, že nemám námitek proti půjčování nebo zveřejňování mé práce nebo její části se souhlasem školy.

……

Jindřich Machka

ABSTRAKT

Ať už jde o vyvozování užitečných informací z vizuálních či zvukových vstupů, autonomní chování dosáhnuté zpětnovazebním učením, generování vlastních snímků z textového zadání nebo čím dál všestrannější vlastnosti velkých modelů pro zpracování jazyka, díky schopnosti neuronových sítí výpočetně reprezentovat vzory v datech s ohledem na určitý cíl, a tak aproximovat jakoukoliv funkci s libovolnou přesností, se umělá inteligence stala středobodem technologického vývoje 21. století.

Cílem mé práce bylo v programovacím jazyce Python s pomocí knihovny NumPy pro zprostředkování maticových operací vytvořit vlastní neuronovou síť a na problému klasifikace snímků ji porovnat s neuronovou sítí vytvořenou pomocí nejpoužívanějšího nástroje pro tvorbu neuronových sítí v průmyslové praxi - knihovny TensorFlow a API Keras.

Při tvorbě vlastní neuronové sítě jsem vycházel z matematické formalizace její nejzákladnější formy (dopředná propagace, logistická aktivační funkce, chybová funkce: střední kvadratická chyba, optimalizační algoritmus: stochastický gradientní sestup, algoritmus zpětné propagace) a při tvorbě neuronových sítí v TensorFlow Keras jsem vycházel z dnešních standardních metod pro klasifikaci obrazu (konvoluční neuronové sítě, regularizace metodou dropout, aktivační funkce ReLU a Softmax, chybová funkce: diskrétní křížová entropie, optimalizační algoritmus: Nadam).

Má vlastní neuronová síť při klasifikaci malých snímků oblečení z datového souboru Fashion MNIST dosahuje na testovací množině dat přesnosti predikcí 84 %, ale kvůli chybějící konvoluci, neoptimálně zvoleným aktivačním funkcím a optimalizačnímu algoritmu výsledný model nestačí na klasifikaci dopravních značek z datového souboru GTSRB. Na tento úkol jsem tedy využil konvoluční neuronovou síť vytvořenou pomocí TensorFlow Keras, která při klasifikaci dopravních značek dosahuje přesnosti 98 %.

OBSAH

Úvod

Obor umělé inteligence tkví ve snaze pochopit a v podobě inteligentních agentů výpočetně implementovat mechanismy, které dávají vzniknout inteligenci. V kontextu mého projektu lze uspokojivě inteligenci definovat jako schopnost agenta modelovat dostupné informace (získané prostřednictvím snímačů), a tak v závislosti na rozpoznaných vzorech v těchto informacích vytvářet správné predikce, díky kterým dále může agent (s užitím akčních členů) úspěšně dosáhnout stanovených cílů (například dosažení žádané hodnoty určité regulované veličiny) v široké škále různých prostředí.[1][2][3] Z tohoto ohledu je nejinteligentnější tvor, kterého známe, člověk. V průběhu 20. století začalo být s vývojem neurovědy čím dál více zřejmé, že veškeré kognitivní funkce, pod které spadá i inteligence, jsou výsledkem elektrochemických operací struktur tvořených komplikovanými sítěmi mozkových buněk – biologických neuronů.[3] V souladu s tímto poznatkem byl vynalezen model biologického neuronu – umělý neuron, který měl výpočetně implementovat funkci biologického neuronu.

Později se ukázalo, že skrze proces učení jsou sítě těchto umělých neuronů, stejně jako sítě biologických neuronů, schopny rozpoznávat vzory v určitém souboru informací, vytvořit tak jejich výpočetní reprezentaci a pomocí ní produkovat žádané predikce v souladu s určitým cílem (např. rozpoznání objektu na snímku) – jsou tedy schopny modelovat vstupní informace s ohledem na tento cíl. Nyní víme, že neuronové sítě jsou schopny modelovat vstupní informace tak, že dokážou na svém výstupu aproximovat jakoukoliv funkci s libovolnou přesností.[3]

Pro neuronové sítě se tak otevřely dveře do obrovského množství průmyslových (ale i vědeckých) aplikací, protože schopnost strojového učení – schopnost zlepšovat svůj výkon na základě nových informací s ohledem na dosažení určitého cíle, aniž bychom explicitně zadali instrukce, jak tohoto cíle dosáhnout[4] – umožňuje vytvářet autonomnější systémy, tedy systémy, které jsou schopny adaptovat se na variace ve vstupních informacích, a tak učením kompenzovat nedostatečné apriorní znalosti poskytnuté programátorem.[3] Tyto vlastnosti dávají vzniknout velkému průmyslovému potenciálu, zejména v automatizování různých procesů.

V mém projektu se zabývám výhradně problémy *klasifikace*. Jedná se o problém, jehož řešení spadá pod způsob strojového učení zvaný *učení s učitelem*. Při učení s učitelem se agent na základě známých, dříve pozorovaných a zpracovaných párů vstupů a výstupů (x, y) z tzv. trénovací množiny dat naučí hypotetickou funkci $h(x)$, která by měla co nejlépe aproximovat cílovou funkci $f(x)$, přičemž cílová funkce $f(x)$ pro vstup x produkuje výstup y s největší přesností, jaké můžeme při řešení daného problému dosáhnout. [3][5]

Při klasifikaci jsou výstupy y nespojité – říká se jim *označení* (angl. labels), protože se jedná o jména kategorií, které náleží vstupům x. Klasifikační hypotetická funkce tak na základě určitého vstupu x s určitou přesností produkuje náležité označení y.[3] Pokud je tato funkce úspěšná, říkáme, že vytváří správné predikce. Při matematickém popisu naučené neuronové sítě budu tedy její výstup označovat $h(x)$ v souladu s pojmem hypotetické funkce. V mém projektu na vstup neuronová síť přijímá vektor číselných hodnot pixelů, který byl vytvořen z původního snímku, a na výstupu je označení objektu, který se na snímku nachází.

Výsledná naučená hypotetická funkce je *model strojového učení*. Model strojového učení lze volně definovat jako výpočetní reprezentace vzorů v určitém souboru informací, na jejímž základě se dají vytvářet správné predikce ve stejném prostředí, ze kterého pochází daný soubor informací, s ohledem na stanovený cíl. Model strojového učení dostaneme uplatněním *algoritmu strojového učení* na daný soubor informací. Neuronové sítě jsou tedy druhem algoritmu strojového učení*[6], ale v průběhu písemné práce budu používat tuto terminologii volněji. Pokud neuronovou síť uplatníme na určitý správně zpracovaný trénovací datový soubor snímků s náležitými označeními, dostaneme model tohoto datového souboru, který je schopný predikcí správných označení podobných snímků jako těch, které se nacházejí v daném trénovacím datovém souboru.

Jelikož hypotetická funkce je součástí tzv. prostoru hypotéz, existuje více možných modelů určitého datového souboru, z nichž některé jsou více a některé méně

* Pojmem neuronové sítě dále budu označovat jen umělé neuronové sítě a pojmem neurony jen umělé neurony, pokud kontext nebude vyžadovat jinak.

úspěšné s ohledem na dosažení předepsaného cíle. Cílem při tvorbě výstupu mého projektu bylo:

1. Pomocí mé vlastní neuronové sítě vytvořené v Pythonu vytvořit takový model, který dosahuje alespoň 80% přesnosti klasifikace snímků oblečení z datového souboru Fashion MNIST.
2. Pomocí knihovny TensorFlow a API Keras pro Python vytvořit takový model, který dosahuje alespoň 90% přesnosti klasifikace snímků dopravních značek z datového souboru GTSRB.

1. SEZNÁMENÍ S VÝSTUPEM PROJEKTU

1.1 DATOVÉ SOUBORY PRO KLASIFIKACI OBRAZU POUŽITÉ V PROJEKTU

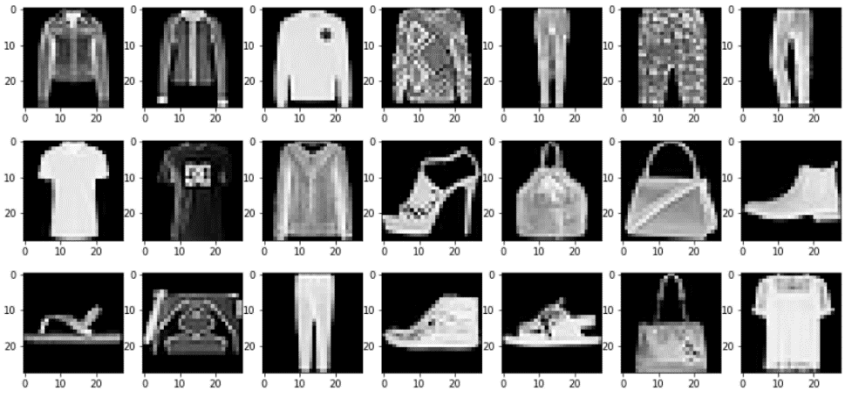
1.1.1 FASHION MNIST

Jedná se o volně dostupný datový soubor obsahující 70000 černobílých snímků o rozměrech 28x28 px, na kterých jsou vyobrazeny různé druhy lidského oblečení. Rozdělení na trénovací a testovací množinu je v poměru 6:1.* Dohromady tento datový soubor obsahuje 10 kategorií oblečení, přičemž pro každou kategorii je zde 7000 snímků.

Tento datový soubor byl vytvořen za účelem nahradit datový soubor MNIST, který obsahuje snímky s ručně napsanými číslicemi. MNIST byl několik let standardním datovým souborem pro výkonové testy algoritmů strojového učení, Fashion MNIST však nabízí větší rozmanitost a jeho klasifikace je složitější.[7]

Tento datový soubor v mém projektu slouží především k otestování správné funkce mé vlastní neuronové sítě.

Obrázek 1: Ukázka datového souboru Fashion MNIST



Zdroj: https://www.tensorflow.org/api_docs/python/tf/keras/datasets/fashion_mnist/load_data

* Trénovací a testovací fáze je vysvětlena v kapitole 5.2

1.1.2 German Traffic Recognition Benchmark

GTSRB (German Traffic Recognition Benchmark) obsahuje přes 50000 barevných snímků o různých rozměrech (nejmenší 25x26 px, největší 211x236 px), na kterých je vyobrazeno celkem 43 kategorií dopravních značek. Některé dopravní značky spadají pod stejnou obecnější kategorií, a tak si jsou z hlediska barev či symbolů podobné. Dále snímky v tomto datovém souboru byly pořízeny za různých světelných podmínek, v odlišném počasí a s variací v naklonění a úhlu dopravních značek.[8]

Na tomto datovém souboru chci ukázat nedostatky mé vlastní neuronové sítě a funkci novějších metod, které jsou dnes používané v praxi.

Obrázek 2: Ukázka datového souboru GTSRB

Zdroj: https://app.activeloop.ai/activeloop/gtsrb-test

1.2 Struktura výstupu

Výstupem tohoto projektu je:

1. má vlastní základní umělá neuronová síť vytvořená v jazyce Python pouze s pomocí knihovny NumPy, která zprostředkovává nezbytné maticové operace. Tuto neuronovou síť jsem nezdokumentoval podrobným popisem programu, ale matematickými vztahy, podle kterých jsem program napsal – tyto vztahy jsou také formou komentářů vyznačeny v konečném programu. Principy této neuronové sítě tak vysvětluji v kapitole č. 2 a 3. příloze je má vlastní neuronová síť pod jménem „custom_ann.py“ uložena jako soubor programovacího jazyka Python.

2. porovnání mé vlastní neuronové sítě s konvoluční neuronovou sítí vytvořenou pomocí knihovny TensorFlow a API Keras na problému klasifikace snímků z datových souborů Fashion MNIST a GTSRB. Konvoluční neuronové sítě jsou vysvětleny v kapitole č. 4, tvorba v TensorFlow Keras je vysvětlena v kapitole č. 5. Porovnání výstupních neuronových sítí je zdokumentováno v kapitole č. 6. Programování neuronové sítě v TensorFlow Keras, učení obou neuronových sítí a jejich zhodnocení na obou uvedených datových souborech je odevzdáno ve formě Jupyter notebooků s názvy „fashion_mnist_classification.ipynb“ a „gtsrb_classification.ipynb“. Notebooky lze otevřít a spustit v jakémkoliv rozhraní, které podporuje formát .ipynb, tedy například Google Colab, rozhraní Jupyter nebo VSCode, ale ve výchozím stavu jsou kódové buňky pro nahrání dat přizpůsobeny rozhraní Google Colab za předpokladu, že uživatel importoval složku s přílohou na svůj Google Disk do výchozí složky „Můj disk“. K snadnému prohlédnutí výsledných notebooků bez možnosti spouštění jednotlivých kódových buněk odkazuji na složku „notebooks_pdf“ v příloze.
3. jednoduchá aplikace demonstrující funkční model klasifikace snímků dopravních značek vytvořený pomocí TensorFlow Keras. Aplikace je implementována opět formou Jupyter notebooku s názvem „gtsrb_model_demonstration.ipynb“ a je zdokumentována v kapitole č. 7. Funkční model, který aplikace využívá, je v příloze uložený pod názvem „keras_gtsrb_model.h5“.

2. PRINCIP NEURONOVÝCH SÍTÍ

Znalosti o principu neuronových sítí shrnuté v této kapitole jsem získal především ze zdrojů [9], [10] a [11].

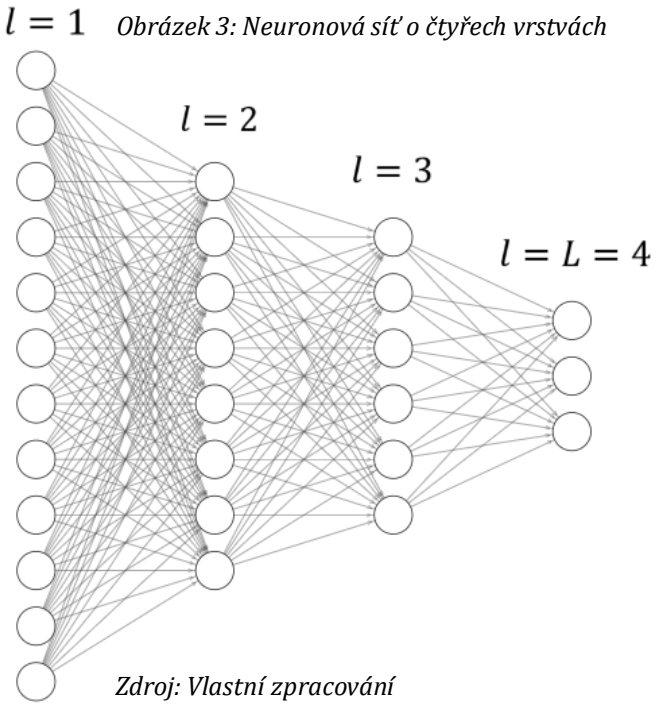
2.1 CO JE TO NEURONOVÁ SÍŤ?

Neuronová síť je druh algoritmu strojového učení, který je dán:

1. *Dopředným zpracováním* vstupních informací skrze určitý počet po sobě následujících vrstev obsahujících určitý počet paralelních výpočetních jednotek zvaných umělé neurony.
2. *Optimalizačním algoritmem*, který konfiguruje parametry neuronové sítě za účelem minimalizování její chybovosti vzhledem k dosažení určitého cíle. Právě touto optimalizací parametrů neuronová síť vytváří výpočetní model vstupních informací vzhledem k dosažení určitého cíle.
3. *Algoritmem zpětné propagace*, který pro každý parametr neuronové sítě spočítá, jak bychom ho měli změnit, abychom zmenšili chybovost neuronové sítě vzhledem k dosažení určitého cíle.

Každý neuron vrstvy $l \in \{1; 2; \dots; L\}$, pokud se nejedná o neuron ve vstupní vrstvě $l = 1$ či výstupní vrstvě $l = L$, na svůj vstup přijímá výstup (neboli *aktivaci*) každého neuronu $a_k^{(l-1)} \in \mathbb{R}$ z předešlé vrstvy $l - 1$, tyto vstupy výpočetně zpracovává a produkuje aktivaci $a_j^{(l)} \in \mathbb{R}$, kterou následně posílá na vstup všech neuronů v další vrstvě $l + 1$.

Obrázek 3: Neuronová síť o čtyřech vrstvách



Zdroj: Vlastní zpracování

Neuronová síť se tedy skládá:

1. Ze vstupní vrstvy neuronů $l = 1$, které nevykonávají žádné výpočetní operace a určují tak vektor vstupních dat $\vec{x}$ neuronové sítě, kde každý neuron ve vstupní vrstvě reprezentuje jednu vstupní veličinu. V mém projektu každý neuron ve vstupní vrstvě určuje číselnou hodnotu určitého pixelu vstupního snímku.
2. Z určitého počtu skrytých vrstev $l \in \{2; 3; \dots; L\}$ o určitém počtu umělých neuronů, kde každý neuron určité skryté vrstvy l přijímá na vstup aktivace všech neuronů z předešlé vrstvy $l - 1$ a svou aktivaci posílá na vstup každého neuronu následující vrstvy $l + 1$.
3. Z výstupní vrstvy neuronů L, kde každý neuron na vstup přijímá aktivaci každého neuronu z poslední skryté vrstvy $L - 1$ a svou aktivací určuje jednu hodnotu uvnitř výstupního vektoru neuronové sítě $h(\vec{x}) = \vec{a}^{(L)}$. V mém projektu každý výstupní neuron odpovídá jedné výstupní klasifikační kategorii, kde aktivace určitého výstupního neuronu udává pravděpodobnost, že objekt na snímku spadá pod onu kategorii, která je danému výstupnímu neuronu přiřazena. Výstupem mé neuronové sítě je tedy vektor pravděpodobností a konečná predikce je určena pozicí největší číselné hodnoty uvnitř výstupního vektoru.

V mém programu je neuronová síť realizována třídou `Artificial_Neural_Network` a všechny ostatní vysvětlené mechanismy neuronových sítí jsou implementovány jako metody této třídy.

2.2 Dopředná propagace napříč neurony

Dopředná propagace neuronové sítě napříč neurony (tedy určení aktivace jednoho neuronu pro každý neuron v síti) je dána takto[10]:

$$a_j^{(l)} = \sigma\left(\vec{w}_j^{(l)} \cdot \vec{a}^{(l-1)} + b_j^{(l)}\right) = \sigma(z_j^{(l)}) \tag{1}$$

kde:

$l \in \{2; 3; \dots; L\}$, protože aktivace neuronů vstupní vrstvy $l = 1$ jsou již dané vektorem vstupních informací $\vec{x}$.

$a_j^{(l)}$ je aktivace j-tého neuronu l-té vrstvy. Dále takto budu označovat i samotný neuron.

$\vec{a}^{(l-1)}$ je vektor aktivací neuronů $(l-1)$-té vrstvy, tedy vektor vstupů j-tého neuronu l-té vrstvy $a_j^{(l)}$.

$\vec{w}_j^{(l)} = (w_{j,1}^{(l)}; w_{j,2}^{(l)}; \dots; w_{j,n}^{(l)}) \in \mathbb{R}$ je vektor *vah* pro spojení mezi j-tým neuronem l-té vrstvy a jeho vstupy, tedy neurony $(l-1)$-té vrstvy, o počtu n, kde k-tá hodnota určuje váhu $w_{j,k}^{(l)}$ pro spojení j-tého neuronu l-té vrstvy $a_j^{(l)}$ a k-tého neuronu $(l-1)$-té vrstvy $a_k^{(l-1)}$. Váha $w_{j,k}^{(l)}$ pro spojení neuronu $a_j^{(l)}$ a neuronu $a_k^{(l-1)}$ udává sílu spojení mezi těmito dvěma neurony a tím zároveň udává, jak velký vliv má výstup neuronu $a_k^{(l-1)}$ na výstup neuronu $a_j^{(l)}$.

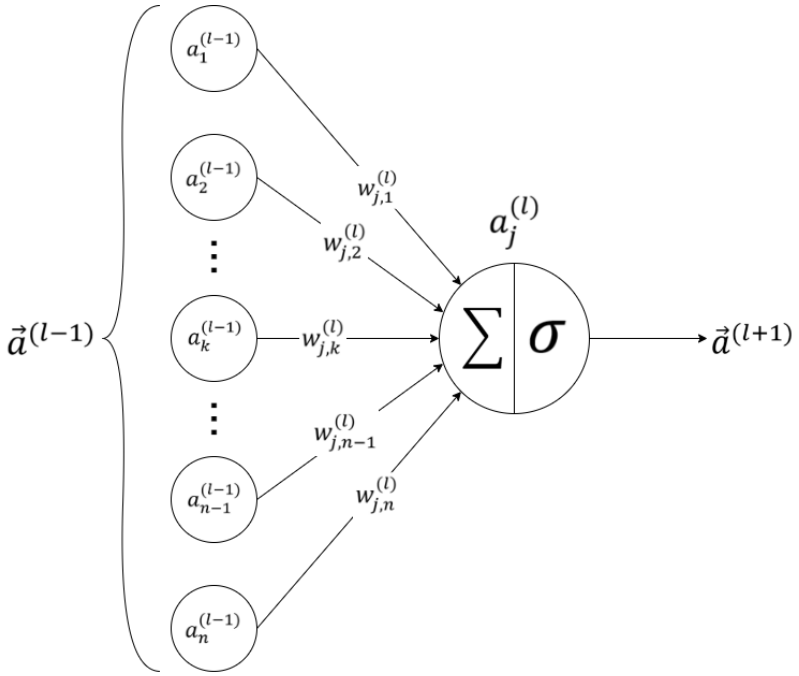
$b_j^{(l)} \in \mathbb{R}$ je *práh* j-tého neuronu l-té vrstvy, který způsobuje posun aktivační funkce $\sigma(z_j^{(l)})$. Právě váhy a prahy jsou *parametry neuronové sítě*.

$z_j^{(l)} \in \mathbb{R}$ je *vážený součet* vstupů j-tého neuronu l-té vrstvy $a_j^{(l)}$.*

$\sigma(z_j^{(l)})$ je *aktivační funkce* j-tého neuronu l-té vrstvy $a_j^{(l)}$, která na základě váženého součtu $z_j^{(l)}$ produkuje finální aktivaci neuronu.

* Váženým součtem vstupů neuronu dále neoznačuji pouze skalární součin vektoru vstupů neuronu a vektoru vah těchto vstupů, ale i přičtený práh, tedy celou operaci $\vec{w}_j^{(l)} \cdot \vec{a}^{(l-1)} + b_j^{(l)}$.

Obrázek 4: Dopředná propagace jednoho neuronu



Zdroj: Vlastní zpracování

2.3 AKTIVAČNÍ FUNKCE

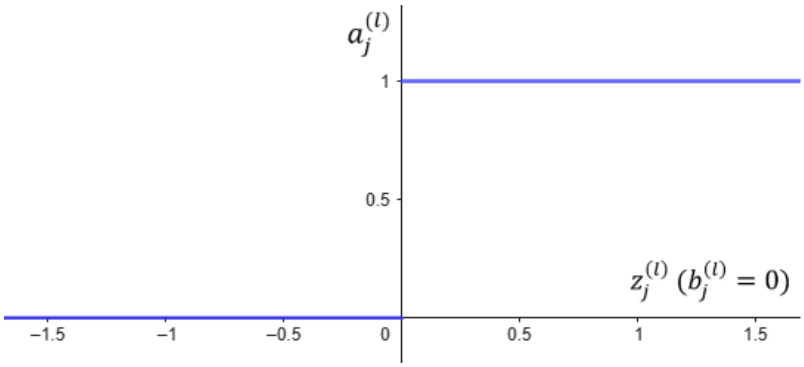
Různé aktivační funkce a jejich praktické vlastnosti objasňují zdroje [3] a [4].

Aktivační funkce neuronů udávají, jak se vstupní informace šíří a transformují skrze neuronovou síť. Existuje mnoho variací aktivačních funkcí. V souladu s definicí vztahu (1) pro dopřednou propagaci napříč neurony následuje popis aktivačních funkcí, které jsem v projektu použil, a popis původní skokové aktivační funkce:

2.3.1 SKOKOVÁ AKTIVAČNÍ FUNKCE

$$a_j^{(l)} = \sigma\left(z_j^{(l)}\right) = \begin{cases} 1\ pro\ z_j^{(l)} > 0 \\ 0\ pro\ z_j^{(l)} \leq 0 \end{cases} \tag{2}$$

Obrázek 5: Skoková aktivační funkce



Zdroj: Vlastní zpracování pomocí https://www.geogebra.org/

Umělý neuron se skokovou aktivační funkcí je původním výpočetním modelem biologického neuronu. Umělý neuron se skokovou aktivační funkcí, který na své vstupy přijímá reálná čísla, se nazývá perceptron.

V naší nervové soustavě se informace šíří pomocí tzv. *akčního potenciálu*, což je elektricky kladně nabitý stav neuronu. Pokud síla elektrických impulsů přijímaných vstupními výběžky neuronu (*dendrity*) přesáhne určitý práh napětí, pak na výstupu tohoto neuronu vzniká elektrický impuls, který putuje výstupními výběžky (*axony*) ke spojením s dendrity dalších neuronů (*synapsím*). Když dosadíme do nerovnic ve vztahu (2), přesuneme práh z výsledných nerovnic na druhou stranu nerovnic a uvědomíme si, že hodnota prahu, tedy včetně znaménka, je vždy při učení nastavena tak, jak učení vyžaduje, pak můžeme znaménko zanedbat a dostaneme vztah, který na první pohled správně, ač ve skutečnosti příliš jednoduše*, modeluje popsaný biologický jev:

$$a_j^{(l)} = \sigma\left(z_j^{(l)}\right) = \begin{cases} 1 \; pro \; \vec{w}_j^{(l)} \cdot \vec{a}^{(l-1)} > b_j^{(l)} \\ 0 \; pro \; \vec{w}_j^{(l)} \cdot \vec{a}^{(l-1)} \leq b_j^{(l)} \end{cases} \tag{3}$$

Při tvorbě neuronové sítě chceme, aby při malé změně některé váhy nebo prahu uvnitř neuronové sítě tato změna vyprodukovala malou změnu ve výstupu neuronové sítě, protože pak bychom mohli tohoto vztahu mezi parametry a

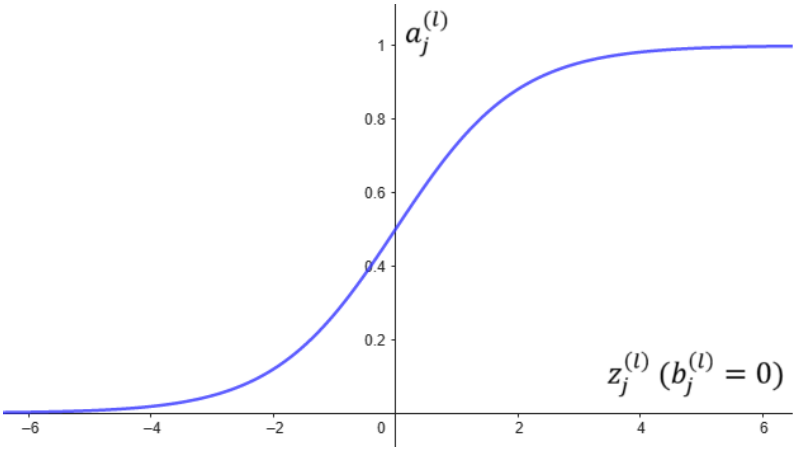
* Podle posledních výzkumů se totiž zdá, že samotné dendrity biologických neuronů také vykazují výpočetní vlastnosti, a tak jeden biologický neuron je schopný výpočtů, kterých jeden perceptron schopen není (například provedení logické operace XOR). Podle těchto výzkumů pak jednotlivé biologické neurony fungují více jako celé umělé neuronové sítě.[12]

výstupem neuronové sítě využit k jejímu učení. Při užití perceptronu ale dochází k tomu, že malá změna váhy může způsobit, že výstup perceptronu se kompletně změní z 0 na 1. Pokud je celá neuronová síť tvořena perceptrony, pak tato malá změna váhy může způsobit velkou změnu napříč celou neuronovou sítí. Učení perceptronových sítí je tak těžké udržet pod kontrolou.[9]

2.3.2 Logistická aktivační funkce

$$a_j^{(l)} = \sigma\left(z_j^{(l)}\right) = \frac{1}{1 + e^{-z_j^{(l)}}} \in (0; 1) \tag{4}$$

Obrázek 6: Logistická aktivační funkce



Zdroj: Vlastní zpracování pomocí https://www.geogebra.org/

Neurony s touto aktivační funkcí jsou v podstatě perceptrony s takovou modifikací, aby malé změny parametrů neuronové sítě vyústily v malou změnu na výstupu neuronové sítě. Výstup této aktivační funkcí se tak pohybuje v rozmezí od 0 do 1.[9]

Má vlastní neuronová síť je tvořena pouze neurony s touto aktivační funkcí. Při klasifikaci snímků oblečení z datového souboru Fashion MNIST, který obsahuje 10 kategorií, obsahuje výstupní vrstva mé neuronové sítě 10 výstupních neuronů. Na výstupu každého z nich je pravděpodobnost, že snímek spadá pod kategorii přiřazenou danému neuronu. Konečné rozhodnutí je dáno neuronem s výstupem nejbližším hodnotě 1. Součet výstupů všech 10 neuronů však při použití logistické aktivační funkce nemusí být roven 1, což znamená, že rozhodnutí neuronové sítě ohledně jedné kategorie neovlivňuje rozhodnutí týkající se jiné kategorie. Logistická aktivační funkce je tak ve skutečnosti vhodnější pro případy, kdy každý vstupní

vektor spadá pod více kategorií, a tak chceme také generovat více výstupních označení.[4]

Tato funkce je v mém programu implementována maticově jako metoda `Artificial_Neural_Network.logistic_activation(self, Z)`. kde `Z` je sloupcový vektor vážených součtů neuronů určité vrstvy. Pro správnou funkci učení neuronové sítě, kterým se zabývám níže, jsem musel implementovat i derivaci logistické aktivační funkce:

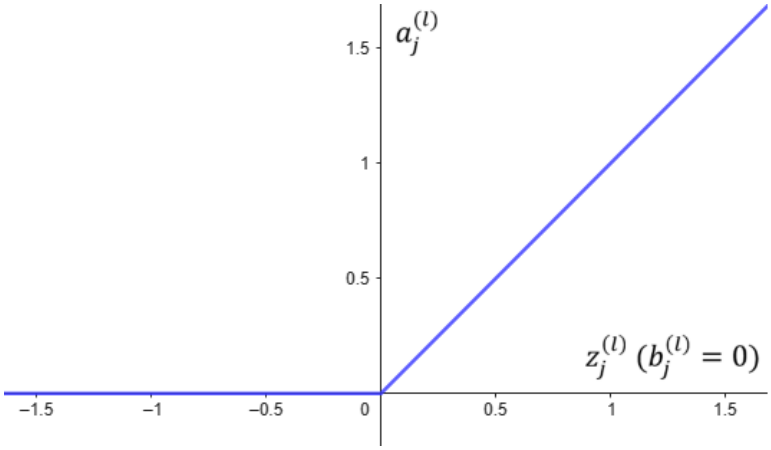
$$\sigma'\left(z_j^{(l)}\right) = \frac{1}{1+e^{-z_j^{(l)}}}\left(1 - \frac{1}{1+e^{-z_j^{(l)}}}\right) \tag{5}$$

V programu je derivace logistické aktivační funkce realizována metodou `Artificial_Neural_Network.logistic_derivative(self, Z)`, kde `Z` je opět sloupcový vektor vážených součtů neuronů určité vrstvy.

2.3.3 Aktivační funkce ReLU (Rectified Linear Unit)

$$a_j^{(l)} = \sigma\left(z_j^{(l)}\right) = \max\left(0,\, z_j^{(l)}\right) = \begin{cases} z_j^{(l)}\ pro\ z_j^{(l)} > 0 \\ 0\ pro\ z_j^{(l)} \leq 0 \end{cases} \in \langle 0; +\infty) \tag{6}$$

Obrázek 7: Aktivační funkce ReLU



Zdroj: Vlastní zpracování pomocí https://www.geogebra.org/

Obecně dnes platí, že tato aktivační funkce je vhodnou výchozí volbou pro všechny skryté vrstvy neuronové sítě. Ukázalo se totiž, že neuronové sítě, které ve skrytých vrstvách používají aktivační funkci ReLU, se snáze učí a obecně fungují optimálněji než neuronové sítě, které ve skrytých vrstvách používají logistickou aktivační funkci.[4][10]

2.3.4 Aktivační funkce Softmax

$$a_j^{(l)} = \sigma\left(\vec{z}^{(l)}\right)_j = \frac{e^{z_j^{(l)}}}{\sum_{i=1}^{n} e^{z_i^{(l)}}} \tag{7}$$

kde $\vec{z}^{(l)}$ je vektor vážených součtů neuronů vrstvy l a n je počet všech neuronů ve vrstvě l.

Pokud použijeme n výstupních neuronů pro n možných výstupních kategorií, pak každý z těchto výstupních neuronů vyprodukuje určité kladné číslo, které ve vztahu k výstupům ostatních neuronů udává, jak moc si je neuronová síť jistá, že vstupní informace odpovídají dané kategorii. Pokud určitý vstupní vektor může odpovídat jen jedné výstupní kategorii, pak by bylo z pravděpodobnostního hlediska vhodnější, kdyby součet výstupů všech výstupních neuronů byl roven 1. Ve výstupní vrstvě se tak pro klasifikaci používá aktivační funkce Softmax, která vektor vážených součtů všech výstupních neuronů transformuje v pravděpodobnostní distribuci.[4]

2.4 DOPŘEDNÁ PROPAGACE NAPŘÍČ VRSTVAMI

Dopředná propagace neuronové sítě napříč vrstvami (tedy určení aktivace všech neuronů v jedné vrstvě pro každou vrstvu v síti) je dáno takto[10]:

$$A^{(l)} = \sigma\left(W^{(l)} \cdot A^{(l-1)} + B^{(l)}\right) \tag{8}$$

$$= \sigma\left(\begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m,1}^{(l)} & w_{m,2}^{(l)} & \cdots & w_{m,n}^{(l)} \end{bmatrix} \cdot \begin{bmatrix} a_1^{(l-1)} \\ a_2^{(l-1)} \\ \vdots \\ a_n^{(l-1)} \end{bmatrix} + \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_m^{(l)} \end{bmatrix}\right)$$

$$= \sigma\left(\begin{bmatrix} w_{1,1}^{(l)} a_0^{(l-1)} + w_{1,2}^{(l)} a_1^{(l-1)} + & \cdots & + w_{1,n}^{(l)} a_n^{(l-1)} \\ w_{2,1}^{(l)} a_0^{(l-1)} + w_{2,2}^{(l)} a_1^{(l-1)} + & \cdots & + w_{2,n}^{(l)} a_n^{(l-1)} \\ \vdots & \ddots & \vdots \\ w_{m,1}^{(l)} a_0^{(l-1)} + w_{m,2}^{(l)} a_1^{(l-1)} + & \cdots & + w_{m,n}^{(l)} a_n^{(l-1)} \end{bmatrix} + \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_m^{(l)} \end{bmatrix}\right)$$

$$= \sigma\left(Z^{(l)}\right) = \sigma\left(\begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_m^{(l)} \end{bmatrix}\right) = \begin{bmatrix} \sigma(z_1^{(l)}) \\ \sigma(z_2^{(l)}) \\ \vdots \\ \sigma(z_m^{(l)}) \end{bmatrix} = \begin{bmatrix} a_0^{(l)} \\ a_1^{(l)} \\ \vdots \\ a_m^{(l)} \end{bmatrix}$$

kde:

$l \in \{2; 3; \ldots; L\}$

$A^{(l)} = \vec{a}^{(l)T}$ je sloupcový vektor aktivací neuronů l-té vrstvy o m řádcích, kde j-tý člen značí aktivaci j-tého neuronu l-té vrstvy neuronové sítě $a_j^{(l)}$.

$A^{(l-1)} = \vec{a}^{(l-1)T}$ je sloupcový vektor aktivací neuronů l-té vrstvy o n řádcích, kde k-tý člen značí aktivaci k-tého neuronu $(l-1)$-té vrstvy neuronové sítě $a_k^{(l-1)}$. Pak sloupcový vektor aktivací $(l-1)$-té vrstvy $A^{(l-1)}$ funguje jako vstupní vektor pro každý neuron l-té vrstvy neuronové sítě.

$W^{(l)}$ je matice vah o m řádcích a n sloupcích, kde j-tý řádek obsahuje vektor vah $\vec{w}_j^{(l)}$ pro spojení mezi $(l-1)$-tou vrstvou neuronů a j-tým neuronem l-té vrstvy neuronové sítě $a_j^{(l)}$. Pak k-tý člen j-tého řádku značí váhu $w_{j,k}^{(l)}$ pro

spojení mezi k-tým neuronem $(l-1)$-té vrstvy neuronové sítě $a_k^{(l-1)}$ a j-tým neuronem l-té vrstvy neuronové sítě $a_j^{(l)}$.

$B^{(l)} = \vec{b}^{(l)T}$ je sloupcový vektor prahů neuronů l-té vrstvy o m řádcích, kde j-tý člen značí práh $b_j^{(l)}$ j-tého neuronu l-té vrstvy neuronové sítě.

$Z^{(l)} = \vec{z}^{(l)T}$ je sloupcový vektor vážených součtů neuronů l-té vrstvy o m řádcích, kde j-tý člen značí vážený součet vstupů $z_j^{(l)}$ pro j-tý neuron l-té vrstvy neuronové sítě.

Dopřednou propagaci napříč vrstvami jsem implementoval jako metodu `Artificial_Neural_Network.forward_propagtaion(self, x)`, kde `x` je určitý vstupní vektor, na jehož základě tato metoda spočítá aktivace všech neuronů uvnitř neuronové sítě. Aktivace a parametry neuronové sítě jsou realizovány proměnnými:

`Artificial_Neural_Network.neuron_activations`, která formou seznamu ukládá všechny sloupcové vektory aktivací neuronů $A^{(l)}$.

`Artificial_Neural_Network.weights`, která formou seznamu ukládá všechny matice vah $W^{(l)}$.

`Artificial_Neural_Network.biases`, která formou seznamu ukládá všechny sloupcové vektory prahů $B^{(l)}$.

3. UČENÍ NEURONOVÝCH SÍTÍ

Můj popis toho, jak se neuronové sítě učí, je syntézou znalostí ze zdrojů [9] a [10].

3.1 CHYBOVÁ FUNKCE

Po zvolení vhodné aktivační funkce můžeme přistoupit k učení. Aby se neuronová síť mohla učit, tedy aby mohla upravovat své parametry na optimální hodnoty, musíme kvantifikovat chybovost neuronové sítě. K tomu slouží *chybová funkce* (angl. *loss function*), jako například *střední kvadratická chyba* (*MSE* z angl. *Mean Squared Error*)[9]:

$$C = \frac{1}{n}\sum_{i=1}^{n}(\vec{y}_i - h(\vec{x}_i))^2 \tag{9}$$

kde:

$X = \{\vec{x}_1, \vec{x}_2, \dots, \vec{x}_n\}$ je trénovací množina vstupních vektorů.

$Y = \{\vec{y}_1, \vec{y}_2, \dots, \vec{y}_n\}$ je trénovací množina výstupních označení ve formě cílových výstupních vektorů, kde na každé pozici jednoho cílového výstupního vektoru $\vec{y}_i$ je správná hodnota pro určitou kategorii, stejně jako na každé pozici ve výstupním vektoru neuronové sítě $h(\vec{x}_i)$ je predikovaná hodnota pro určitou kategorii. Pokud vstupní vektor náleží kategorii reprezentované danou pozicí v cílovém výstupním vektoru, pak hodnota na této pozici je 1 a všechny ostatní hodnoty cílového výstupního vektoru jsou 0.

n je počet všech trénovacích vstupních vektorů, a tak i počet všech klasifikací učiněných při učení.

$h(\vec{x}_i)$ je výstupní vektor hypotetické funkce neuronové sítě pro vstupní vektor $\vec{x}_i$.

$h(\vec{x}_i) = \vec{a}^L$, kde $\vec{a}^L$ je vektor aktivací poslední (L-té) vrstvy neuronové sítě.

Chybovou funkci C je třeba rozlišovat od *chyby* C_x pro jednu učiněnou klasifikaci, pro kterou platí:

$$C_x = (\vec{y}_i - h(\vec{x}_i))^2 \tag{10}$$

Cílem učení neuronové sítě je minimalizovat chybovou funkci prostřednictvím vhodné konfigurace vah a prahů neuronové sítě. S klesající hodnotou chybové funkce by měla růst přesnost predikcí neuronové sítě.* Problém učení neuronových sítí je tedy optimalizační problém. Klasickým řešením takového problému pomocí matematické analýzy by bylo vyřešit rovnici, ve které se derivace naší chybové funkce rovná nule, a následně najít váhy a prahy, které vytváří minima chybové funkce. Derivace chybové funkce při řešení neuronových sítí je ale vzhledem k počtu parametrů z výpočetního hlediska tak složitá, že klasický přístup není prakticky možný. Proto je pro nalezení optimálních vah a prahů užíván tzv. *gradientní sestup*.[13]

Chybovou funkci MSE jsem implementoval jako metodu `Artificial_Neural_Network.loss_function(self, H, Y)`, kde `H` je seznam všech klasifikací učiněných při učení ve formě výstupních vektorů a `Y` je seznam všech trénovacích označení ve formě cílových vektorů.

3.2 GRADIENTNÍ SESTUP

Gradientní sestup je iterativní optimalizační algoritmus, který je po prvotní inicializaci vah a prahů schopen s každou iterací, tzv. *epochou učení*, upravit hodnotu vah a prahů neuronové sítě určitým směrem tak, aby konvergoval k lokálnímu minimu chybové funkce neuronové sítě.

* Pro chybovou funkci a přesnost predikcí neuronové sítě ale neplatí žádný matematický vztah.

Nalezení pouze lokálního minima zde není problém, protože minima chybové funkce neuronové sítě nabývají dostatečně podobných hodnot, takže nalezení lokálního minima se moc neliší od nalezení globálního minima.[10]

Začneme tedy s náhodnou inicializací vah a prahů. Při každé epoše učení je potřeba nejdříve spočítat gradient chybové funkce ∇C_{wb} vzhledem k parametrům neuronové sítě, který je dán vektorem parciálních derivací chybové funkce vzhledem ke každé váze a každému prahu:

$$\nabla C_{wb} = \left(\frac{\partial C}{\partial p_1}, \frac{\partial C}{\partial\, p_2}, \ldots, \frac{\partial C}{\partial p_n}\right) \tag{11}$$

kde p_i je určitá váha nebo práh a n je počet všech parametrů neuronové sítě.

Parciální derivace chybové funkce k určitému parametru p_i v podstatě udává citlivost změny chybové funkce na změnu parametru p_i. Platí, že čím větší je absolutní hodnota parciální derivace chybové funkce vzhledem k parametru p_i, tím větší vliv má změna parametru p_i na změnu chybové funkce. Jelikož gradient určité funkce udává směr, ve kterém tato funkce nejvíce roste, pak jeho záporná hodnota udává směr největšího poklesu funkce. Negativní ∇C_{wb} tak udává, jakou změnu bychom měli aplikovat na každou váhu a práh, abychom minimalizovali chybovou funkci.[10] Pak tedy aktualizujeme každou váhu a práh neuronové sítě podle:

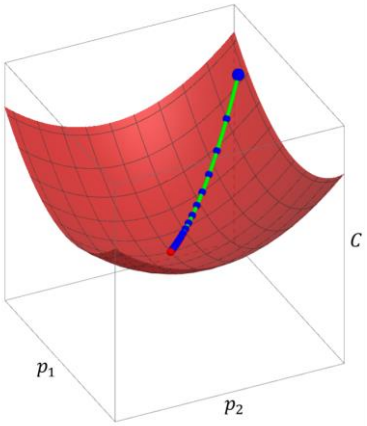
$$w_{jk}^{(l)\prime} = w_{jk}^{(l)} - \eta \frac{\partial C}{\partial w_{jk}^{(l)}} \tag{12}$$

$$b_j^{(l)\prime} = b_j^{(l)} - \eta \frac{\partial C}{\partial b_j} \tag{13}$$

kde $w_{jk}^{(l)}$ je určitá váha, $w_{jk}^{(l)\prime}$ je aktualizovaná váha $w_{jk}^{(l)}$, $b_j^{(l)}$ je určitý práh, $b_j^{(l)\prime}$ je aktualizovaný práh $b_j^{(l)}$ a η je *parametr učení* (angl. *learning rate)*, který udává, po jak velkých krocích při gradientním sestupu postupujeme, tedy po jak velkých krocích postupujeme při učení. Hodnotu parametru učení podle potřeb volí programátor.

Pokud parametr učení zvolíme moc velký, gradientní sestup může lokální minima často přeskočit, a tak k nějakému stěží konverguje. Pokud zvolíme parametr učení moc malý, pak gradientní sestup může být pro praktické účely moc pomalý. [4]

Obrázek 8: Gradientní sestup se dvěma parametry



Zdroj: Gradient Descent – GeoGebra (https://www.geogebra.org/m/ubz5aw8e)

Pro vypočítání ∇C_{wb} se používá algoritmus zpětné propagace. Pomocí algoritmu zpětné propagace pro každou klasifikaci učiněnou při učení spočítáme gradient chyby vzhledem k parametrům neuronové sítě ∇C_{wbx}. Gradient chybové funkce vzhledem k parametrům neuronové sítě pak získáme průměrem všech ∇C_{wbx}[9]:

$$\nabla C_{wb} = \frac{\sum_x^n \nabla C_{wbx}}{n} \tag{14}$$

kde n je počet všech trénovacích vstupních vektorů, a tak i počet všech klasifikací učiněných při učení.

3.2.1 Stochastický gradientní sestup

Učení pomocí gradientního sestupu jsem při implementaci mé vlastní neuronové sítě shledal moc pomalé, takže jsem implementoval tzv. *stochastický gradientní* sestup (*SGD*). SGD umožňuje zrychlit proces učení rozdělením datového souboru na malé vzorky vstupních vektorů, zvané *mini-batch*, a odhadnout ∇C_{wb} výpočtem jednotlivých ∇C_{wbx} pro zvolenou množinu mini-batch, místo pro celý soubor vstupních vektorů:

$$\nabla C_{wb} \approx \frac{\sum_x^m \nabla C_{wbx}}{m} \tag{15}$$

kde m je velikost zvolené mini-batch množiny.

Před každou novou epochou učení je vhodné trénovací datový soubor, ze kterého vybíráme množiny mini-batch, náhodně promíchat.[4]

Při zvolení dostatečné velké mini-batch tímto pro praktické účely aproximujeme ∇C_{wb} dostatečně přesně, abychom v průměru postupovali při gradientním sestupu správným směrem. Ukázalo se však, že k tomu stačí i docela malé velikosti mini-batch. Například v API Keras knihovny TensorFlow je výchozí hodnota velikosti mini-batch pouze 32 vstupních vektorů a má vlastní neuronová síť pro klasifikaci datového souboru Fashion MNIST dosahuje předepsaného cíle při mini-batch o velikosti 10, přestože velikost celé výchozí trénovací množiny vstupních vektorů je 60000 snímků.

V praxi dnešní neuronové sítě jako optimalizační algoritmus skoro vždy používají určitou variaci SGD.[3]

3.2.2 Přizpůsobení programové implementaci

Při implementaci SGD jsem však z důvodu zjednodušení kódu zvolil takový postup, že pro každý vstupní vektor $\vec{x}$ spočítám matici parciálních derivací chyby vzhledem k váhám každé vrstvy $\frac{\partial C_x}{\partial W^{(l)}}$ a sloupcový vektor parciálních derivací chyby vzhledem k prahům každé vrstvy $\frac{\partial C_x}{\partial B^{(l)}}$. Pak rovnou aktualizuji parametry neuronové sítě maticově a bez počítání ∇C_{wb}:

$$W^{(l)\prime} = W^{(l)} - \frac{\eta}{m}\frac{\partial C_x}{\partial W^{(l)}} \tag{16}$$

$$W^{(l)\prime} = W^{(l)} - \frac{\eta}{m}\frac{\partial C_x}{\partial W^{(l)}} \tag{17}$$

V mém programu inicializace vah a prahů probíhá v inicializační metodě `Artificial_Neural_Network.__init__(self, layers)`, kde `layers` je seznam počtů neuronů v jednotlivých po sobě následujících vrstvách, který třída

`Artificial_Neural_Network` přijímá na vstup při vytvoření její instance. SGD jsem implementoval jako metodu `Artificial_Neural_Network.stochastic_gradient_descent(`

```
self, X_train, y_train, epochs, mini_batch_size,
learning_rate, validation_data=None),
```

kde `X_train` je seznam všech trénovacích vstupních vektorů, `y_train` je seznam všech trénovacích označení, `epochs` je počet epoch učení, `mini_batch_size` je velikost mini-batch množin, `learning_rate` je parametr učení a `validation_data` je množina validačních dat, který vysvětluji v kapitole 6.1.

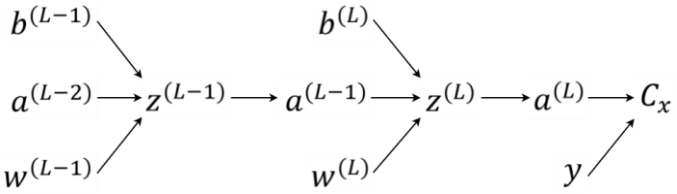
3.3 ALGORITMUS ZPĚTNÉ PROPAGACE

Postup vysvětlení algoritmu zpětné propagace jsem převzal ze zdroje [10].

Prostřednictvím algoritmu zpětné propagace jsme schopni spočítat parciální derivaci chyby vzhledem ke každému parametru neuronové sítě.

Nejdříve se podíváme, jak funguje zpětná propagace pro jednoduchou neuronovou síť o jedné skryté vrstvě:

Obrázek 9: Neuronová síť o třech vrstvách obsahujících vždy jeden neuron



Zdroj: Vlastní zpracování, koncepce ze zdroje [10]

Označme si poslední vrstvu jako L a vstupní vrstvu jako $L-2$. Každá vrstva má jen jeden neuron. Pak můžeme vidět, že pro vypočítání chyby jedné klasifikace potřebujeme aktivaci posledního neuronu $a^{(L)}$ a skutečnou výstupní hodnotu neuronové sítě y. K vypočítání aktivace posledního neuronu potřebujeme vážený součet tohoto neuronu $z^{(L-1)}$. K vypočítání váženého součtu tohoto neuronu potřebujeme váhu $w^{(L)}$ pro spojení tohoto neuronu a neuronu předešlé vrstvy $L-1$, aktivaci neuronu předešlé vrstvy $a^{(L-1)}$ a práh tohoto neuronu $b^{(L)}$.

Chybu tak můžeme popsat jako složenou funkci a podle řetězového pravidla platí, že:

$$\frac{\partial C_x}{\partial w^{(L)}} = \frac{\partial z^{(L)}}{\partial w^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_x}{\partial a^{(L)}} = a^{(L-1)} \sigma'\left(z^{(L)}\right) 2\left(a^{(L)} - y\right) \tag{18}$$

$$\frac{\partial C_x}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_x}{\partial a^{(L)}} = \sigma'\left(z^{(L)}\right) 2\left(a^{(L)} - y\right) \tag{19}$$

$$\frac{\partial C_x}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_x}{\partial a^{(L)}} = w^{(L)} \sigma'\left(z^{(L)}\right) 2\left(a^{(L)} - y\right) \tag{20}$$

Takto tedy získáme parciální derivace $\frac{\partial C_x}{\partial w^{(L)}}$ a $\frac{\partial C_x}{\partial b^{(L)}}$. Ke spočítání ∇C_{wb} uvedené neuronové sítě ale potřebujeme i parciální derivace $\frac{\partial C_x}{\partial w^{(L-1)}}$ a $\frac{\partial C_x}{\partial b^{(L-1)}}$. Ty získáme pomocí spočítané parciální derivace $\frac{\partial C_x}{\partial a^{(L-1)}}$, kterou použijeme k tzv. zpětné propagaci chyby skrze neuronovou síť. Pokud totiž zjistíme, jak citlivá je změna chyby neuronové sítě na změnu aktivace jakéhokoliv neuronu, pak díky výše uvedeným rovnicím můžeme zjistit citlivost změny chyby na změnu vah, prahů a aktivací neuronů z předešlé vrstvy. A spočítané parciální derivace chyby neuronové sítě vzhledem k aktivacím neuronů z předešlé vrstvy můžeme opět použít ke zpětné propagaci chyby o další vrstvu, až nakonec zjistíme parciální derivace chyby vzhledem ke každé váze a prahu uvnitř neuronové sítě. U jednoduché neuronové sítě popsané výše tedy kompletně vykonáme zpětnou propagaci takto:

$$\frac{\partial C_x}{\partial w^{(L-1)}} = \frac{\partial z^{(L-1)}}{\partial w^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial C_x}{\partial a^{(L-1)}} = a^{(L-2)} \sigma'\left(z^{(L-1)}\right) \frac{\partial C_x}{\partial a^{(L-1)}} \tag{21}$$

$$\frac{\partial C_x}{\partial b^{(L-1)}} = \frac{\partial z^{(L-1)}}{\partial b^{(L-1)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial C_x}{\partial a^{(L-1)}} = \sigma'\left(z^{(L-1)}\right) \frac{\partial C_x}{\partial a^{(L-1)}} \tag{22}$$

Parciální derivaci $\frac{\partial C_x}{\partial a^{(L-2)}}$ už nepočítáme, protože $a^{(L-2)}$ je aktivace vstupního neuronu, takže už jsme zjistili parciální derivace chyby vzhledem ke všem parametrům neuronové sítě a nemusíme provádět zpětnou propagaci o další vrstvu. ∇C_{wbx} této neuronové sítě nám tedy vyjde:

$$\nabla C_{wbx} = \left(\frac{\partial C_x}{\partial w^{(L-1)}}, \frac{\partial C_x}{\partial b^{(L-1)}}, \frac{\partial C_x}{\partial w^{(L)}}, \frac{\partial C_x}{\partial b^{(L)}}\right) \tag{23}$$

Takto jsme postupovali u neuronové sítě, která má pro zpětnou propagaci jen jednu možnou cestu. Při počítání ∇C_{wbx} libovolné neuronové sítě postupujeme stejně, ale pro každou možnou cestu uvnitř dané neuronové sítě. Obecně tedy platí, že:

$$\frac{\partial C_x}{\partial w_{j,k}^{(l)}} = \frac{\partial z_j^{(l)}}{\partial w_{j,k}^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial C_x}{\partial a_j^{(l)}} = a_k^{(l-1)} \sigma'\left(z_j^{(l)}\right) \frac{\partial C_x}{\partial a_j^{(l)}} \tag{24}$$

$$\frac{\partial C_x}{\partial b_j^{(l)}} = \frac{\partial z_j^{(l)}}{\partial b_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial C_x}{\partial a_j^{(l)}} = \sigma'\left(z_j^{(l)}\right) \frac{\partial C_x}{\partial a_j^{(l)}} \tag{25}$$

$$\frac{\partial C_x}{\partial a_k^{(l-1)}} = \sum_{\mathrm{j}=1}^{\mathrm{m}} \frac{\partial z_j^{(l)}}{\partial a_k^{(l-1)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial C_x}{\partial a_j^{(l)}} = \sum_{\mathrm{j}=1}^{\mathrm{m}} w_{j,k}^{(l)} \sigma'\left(z_j^{(l)}\right) \frac{\partial C_x}{\partial a_j^{(l)}} \tag{26}$$

kde:

$l \in \{2; 3; \ldots; L\}$

C_x je chyba jedné klasifikace

$w_{j,k}^{(l)}$ je váha spojení k-tého neuronu $(l-1)$-té vrstvy $a_k^{(l-1)}$ a j-tého neuronu l-té vrstvy $a_j^{(l)}$

$b_j^{(l)}$ je práh j-tého neuronu l-té vrstvy

$z_j^{(l)}$ je vážený součet j-tého neuronu l-té vrstvy

σ' je derivace aktivační funkce

$a_j^{(l)}$ je aktivace j-tého neuronu l-té vrstvy

$a_k^{(l-1)}$ je aktivace k-tého neuronu $(l-1)$-té vrstvy

m je počet neuronů v l-té vrstvě neuronové sítě

Neuron a_k^{l-1} ovlivňuje chybu přes všechny cesty, které následují po spojeních tohoto neuronu se všemi neurony následující vrstvy l. Neuron a_k^{l-1} tedy má vliv na aktivace všech neuronů vrstvy l, které opět ovlivňují chybu skrze spojení se všemi neurony vrstvy $l+1$, atd. Při počítání $\frac{\partial C_x}{\partial a_k^{(l-1)}}$ tedy musíme brát v potaz vliv neuronu $a_k^{(l-1)}$ na každý neuron vrstvy l, a proto je ve vztahu (26) použita suma, která sčítá výraz $\frac{\partial z_j^{(l)}}{\partial a_k^{(l-1)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial C_x}{\partial a_j^{(l)}}$ přes všechny neurony $a_j^{(l)}$ ve vrstvě l.

Z důvodu rychlosti algoritmu jsem musel uvedený proces implementovat maticově:

$$\frac{\partial C_x}{\partial W^{(l)}} = \sigma'\left(Z^{(l)}\right) \odot \frac{\partial C_x}{\partial A^{(l)}} \cdot A^{(l)T} \tag{27}$$

$$\frac{\partial C_x}{\partial B^{(l)}} = \sigma'\left(Z^{(l)}\right) \odot \frac{\partial C_x}{\partial A^{(l)}} \tag{28}$$

$$\frac{\partial C_x}{\partial A^{(l-1)}} = W^{(l)T} \cdot \left(\sigma'\left(Z^{(l)}\right) \odot \frac{\partial C_x}{\partial A^{(l)}}\right) \tag{29}$$

kde $\odot$ je Hadamardův součin a $^{\mathrm{T}}$ značí transponovanou matici.

Tyto spočítané hodnoty pak můžeme předat algoritmu SGD, a tak implementujeme učení neuronové sítě.

Algoritmus zpětné propagace je v mém programu realizován metodou `Artificial_Neural_Network.backward_propagation(self, h, y)`, kde `h` je učiněná klasifikace a `y` je správné označení. Parciální derivaci chyby vzhledem k aktivaci jednoho neuronu výstupní vrstvy $a_i^{(L)} = h(\vec{x})_i$, která je součástí vztahů (18), (19) a (20), jsem naprogramoval jako metodu `Artificial_Neural_Network.loss_derivative(self, hi, yi)`, kde `hi` je aktivace jednoho neuronu výstupní vrstvy a `yi` je jeho cílová hodnota.

Z uvedeného popisu je zřejmé, že chybová funkce a použité aktivační funkce musí být pro funkci algoritmu zpětné propagace diferencovatelné. Není to pouze o tom,

zda daná funkce je nebo není diferencovatelná v celém definičním oboru, ale i o dalších vlastnostech derivace jednotlivých aktivačních funkcí. Pro praktické důsledky derivací uvedených aktivačních funkcí na učení odkazuji na zdroj [4].

Přestože s rostoucím výpočetním výkonem nových hardwarových technologií se algoritmus zpětné propagace stal velice efektivním řešením pro učení neuronových sítí, synaptická spojení biologických neuronů jsou pouze dopředného charakteru.[14] Více se porovnáním umělých a biologických neuronových sítí zabývat nebudu, ale odkazuji na zdroj [14].

4. KONVOLUČNÍ NEURONOVÉ SÍTĚ

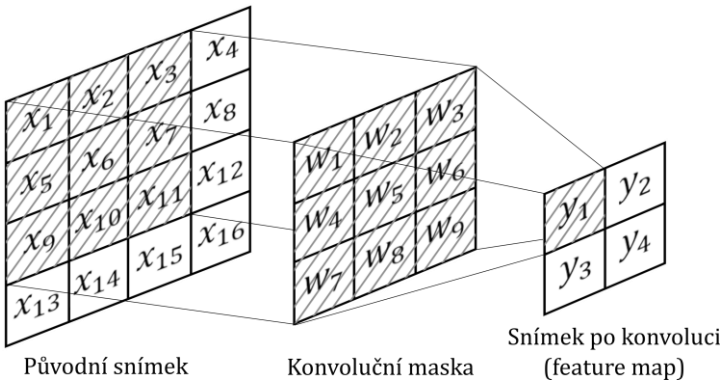
Znalosti o konvoluci shrnuté v této kapitole jsem získal ze zdrojů [11] a [4].

Neuronová síť při klasifikaci obrazu přijímá na vstup vektor číselných hodnot pixelů daného snímku a výstupem je označení toho, co se na snímku nachází za objekt. Pro úkol rozpoznání obrazu se používají tzv. *konvoluční neuronové sítě* (*CNN*).

Konvoluce je v kontextu zpracování obrazu proces, při kterém:

1. Každý pixel v určité části snímku vynásobíme jeho váhou z tzv. *konvoluční masky*, která je v této konvoluční masce na stejné pozici, jako je náležitý pixel v části snímku, na kterou aplikujeme konvoluční masku.
2. Výsledné vážené hodnoty pixelů sečteme a dostaneme jeden pixel nového snímku po konvoluci, tzv. *feature map*.

Obrázek 10: Konvoluce



Zdroj: Vlastní zpracování

Konvoluční maska je tedy určitý filtr, který uplatníme na takový soubor různých částí snímku, abychom při celého snímku konvoluci nevynechali žádný pixel. Můžeme například s konvolucí začít v levém horním okraji snímku a po určitých krocích posunovat konvoluční maskou po šířce a výšce snímku.

To, po jak velkých krocích uplatňujeme konvoluční masku po šířce a výšce snímku (tedy také to, jak moc se části snímku, na které uplatňujeme konvoluční masku, překrývají), je na volbě programátora. Tento parametr se nazývá *stride*. Pokud například použijeme stride 1, pak konvoluční masku posunujeme vždy o jeden pixel.

Abychom při konvoluci nezanedbali informace na okrajích snímku nebo pokud chceme, aby snímek po konvoluci (výsledná feature map) měl stejné rozměry, jako snímek před konvolucí, můžeme za okraje snímku přidat nulové pixely. Uplatníme tak tzv. *zero padding*.

Konvoluci můžeme do neuronových sítí zakomponovat pomocí konvolučních vrstev. Váhy konvoluční masky pak fungují jako všechny ostatní váhy neuronové sítě, takže to, jaká konvoluční maska je optimální pro rozpoznání obrazu, se neuronová síť sama naučí.

Konvoluce umožňuje:

1. aby neuronová síť nemusela hledat strukturu v dlouhém vektoru reálných čísel, ale aby mohla pracovat na základě vlastností určitých částí snímku a vzorů v daných částích snímku, které jsme pomocí konvoluce extrahovali (např. zda se v určité části nachází hrana nějakého objektu nebo zda se v této části nachází určitá křivka, …).
2. snížit počet vah neuronové sítě, který může pro velké snímky představovat problém, tak, že každý neuron v konvoluční vrstvě není propojený s každým neuronem předešlé vrstvy, ale pouze s neurony, které reprezentují pixely použité v dané konvoluční masce.

Neuronová síť může pracovat s více než jednou konvoluční maskou, což umožňuje extrahování různých vzorů v dané části snímku a vlastností této části.

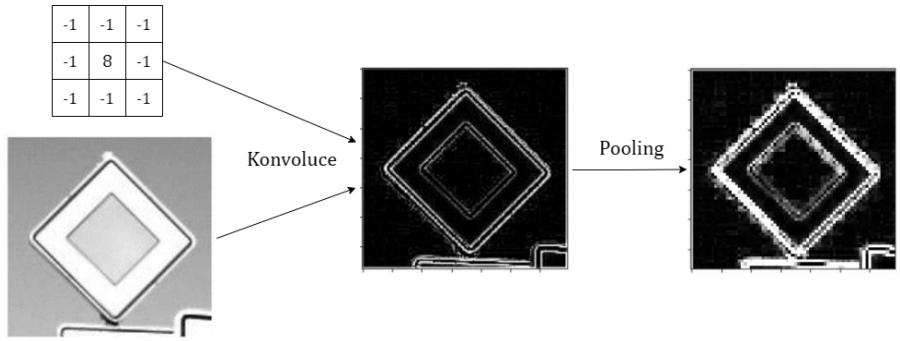
Pro shrnutí toho, jak jsou konvoluční neuronové sítě inspirovány vizuální kůrou člověka, odkazuji na zdroj [4].

V praxi po konvoluční vrstvě často následuje vrstva, která v neuronové síti implementuje tzv. *pooling*. Pooling k dalšímu snížení počtu parametrů, a tak i snížení počtu potřebných výpočtů, tím, že navzorkuje určité části snímku po konvoluci, a tak zmenší jeho rozměry. Nejčastěji se pro toto navzorkování používá buď průměr hodnot všech pixelů v dané části snímku (*average pooling*), nebo zkrátka výběr největší hodnoty ze všech pixelů v dané části snímku (*max pooling*). Je důležité, že

pomocí metody pooling zároveň dostatečně zachováme relevantní vzory v původním snímku a jeho vlastnosti extrahované konvolucí.

Například pokud použijeme konvoluční masku na obrázku 11, pak výsledná feature map bude zobrazovat pouze okraje všech objektů na snímku.

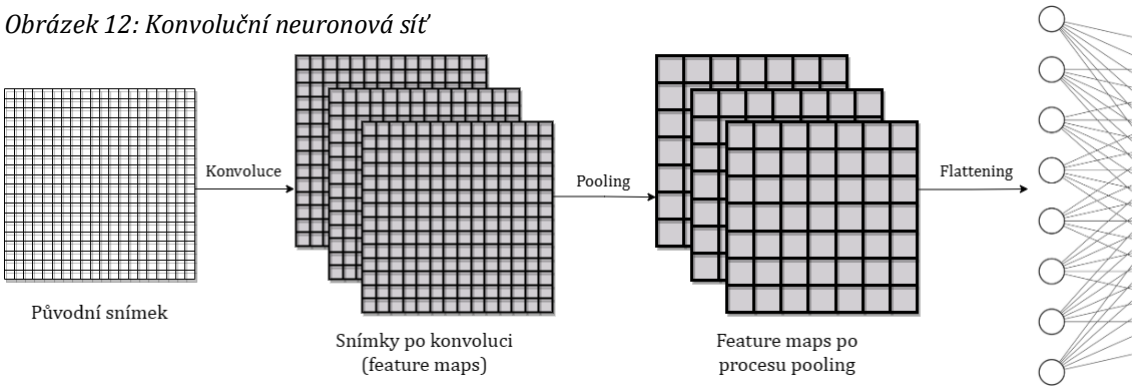
Obrázek 11: Příklad konvoluce a procesu pooling



Zdroj: Vlastní zpracování

Pokud výsledky vrstev pro konvoluci a pooling nejsou ve formě vektoru reálných čísel, tak je upravíme pomocí procesu *flattening*, a pak výsledný vektor přivedeme na vstup následující vrstvy neuronové sítě. Konvoluční vrstvy a vrstvy pro pooling můžeme v neuronové síti použít několikrát po sobě, čímž postupně extrahujeme vzory ve snímku a jeho vlastnosti na vyšší a vyšší úrovni, což neuronové síti následně pomůže určit, co se na daném snímku nachází. Dále je pomocí procesů konvoluce a pooling neuronová síť odolnější vůči variacím, například když dva snímky zobrazují stejný objekt, ale na jednom snímku je daný objekt trochu natočený.

Obrázek 12: Konvoluční neuronová síť



Zdroj: Vlastní zpracování, koncepce ze zdroje [11]

5. KNIHOVNA TENSORFLOW PRO IMPLEMENTACI NEURONOVÝCH SÍTÍ

5.1 ÚVOD DO TENSORFLOW

Znalosti o knihovně TensorFlow sumarizované v této kapitole jsem získal především ze zdrojů [4] a [15].

TensorFlow je open-source knihovna a komplexní systém nástrojů pro hluboké učení (učení neuronových sítí se skrytými vrstvami) vytvořený společností Google. Především obsahuje víceúrovňové API pro programování neuronových sítí. TensorFlow poskytuje API pro více programovacích jazyků, ale nejpoužívanější je verze pro jazyk Python, který se v posledních několika letech stal předním programovacím jazykem pro statistiku, datovou vědu, strojové učení a především hluboké učení zahrnující například i strojové vidění nebo zpracování jazyka. Toho Python v kombinaci s lehce čitelným kódem dosáhl právě díky širokému spektru knihoven, nástrojů a služeb, jako například NumPy, Pandas, Matplotlib, OpenCV, Jupyter, Scikit-Learn, PyTorch a TensorFlow, které usnadňují práci ve zmíněných doménách informatiky.

Obrázek 13: Logo TensorFlow

Zdroj: https://www.tensorflow.org/

TensorFlow podporuje Windows, Linux a MacOS, ale díky TensorFlow Lite lze používat TensorFlow modely také na mobilních zařízeních, které běží na operačních systémech iOS a Android. TensorFlow.js dále umožňuje tvorbu TensorFlow modelů v jazyce JavaScript a používání TensorFlow modelů ve webovém rozhraní.

Díky TensorFlow lze modely neuronových sítí spouštět pomocí GPU, takže jednotlivé neurony uvnitř neuronové sítě mohou své výpočty provádět paralelně, což je užitečné především, pokud náš problém vyžaduje implementaci hodně hluboké neuronové sítě nebo složitější neuronové architektury. Skrze cloudovou službu Google Colab,, která je založená na programovacím rozhraní Jupyter

notebook, a která umožňuje psát a spouštět Python skripty v prohlížeči, se lze bez poplatku připojit na vzdálenou GPU nebo dokonce na TPU (*Tensor Processing Unit*), což je ASIC hardware vytvořený specificky pro optimalizaci vykonávání operací, které hrají roli ve hlubokém učení.*

Další nástroje TensorFlow jsou například TensorBoard (pro vizualizaci statistik modelů a průběhu jejich učení), TensorFlow Hub (repozitář již naučených modelů) a TensorFlow Playground (simulátor neuronové sítě, kde lze volně konfigurovat hyperparametry a vizualizovat průběh učení na problému regrese nebo binární klasifikace).

Nejpoužívanější alternativou TensorFlow je knihovna PyTorch od společnosti Facebook. Zatímco TensorFlow je knihovna zaměřená spíše na průmyslové aplikace a technologie v produkci, PyTorch je více používána ve výzkumných projektech.[19]

Pro větší flexibilitu TensorFlow (dále také `tf`) zprostředkovává nízkoúrovňové API (`tf.nn`, `tf.losses`, `tf.metrics`, `tf.optimizers`, `tf.train`, `tf.initializers`), které slouží pro tvorbu a úpravu vlastních, komplexnějších neuronových sítí a přímou manipulaci tensorů.†

Pro účely mého projektu však využívám vysokoúrovňovou API TensorFlow Keras, která pouze na jednotkách až desítkách řádcích umožňuje snadnou tvorbu, kompilaci, učení, testování a použití modelu na nových datech. Jednoduchost Keras API v kombinaci s širokou online vzdělávací komunitou, která vyrostla okolo strojového učení, umožňuje, aby pochopení základních nástrojů a mechanismů, které jsou napříč průmysly a vědeckými obory používány pro řešení problémů hlubokého učení, bylo přístupné komukoli s přístupem k internetu.

Obrázek 14: Logo Keras

Zdroj: https://keras.io/

* Bezplatné využívání vzdálených GPU a TPU však podléhá dynamickému omezení, které není přesně stanovené, a které po určité době nepoužívání zase vyprší.
† Tensor je zjednodušeně n-dimenzionální pole (*array*), tedy také vektor a matice. Matematická definice tensoru je složitější a určitými vlastnostmi se tensor o dvou dimenzích liší od klasické matice.

5.2 Základní postup implementace neuronových sítí pomocí TensorFlow Keras

V této kapitole jsem čerpal z dokumentací [16] a [17].

Základní obecný postup při implementaci neuronové sítě pro klasifikaci vztažený k API Keras je následující:

1. Importujeme knihovnu TensorFlow jako `tf`
2. Nahrajeme a zpracujeme data tak, aby se na nich mohla učit neuronová síť. Tento krok může být více nebo méně komplikovaný v závislosti na tom, v jaké formě a v jakém stavu data importujeme. Nahraný a zpracovaný datový soubor rozdělíme na *trénovací množinu* (trénovací vstupy, trénovací označení), kterou použijeme pro učení neuronové sítě, a *testovací množinu* (testovací vstupy, testovací označení), kterou použijeme pro testování neuronové sítě. Standardní poměr tohoto dělení je 8:2, ale v praxi záleží na velikosti souboru. Může se stát, že naše neuronová síť se nadmíru přizpůsobí trénovacím datům a výsledný model nebude dost obecný na to, aby si dokázal poradit i se vstupními daty, které při učení nezaznamenal, tedy s testovacími daty. Tento jev se nazývá *přeučení* (angl.. *overfitting*) a dá se mu předejít například přidáním vrstvy `tf.keras.layers.Dropout()`, která při učení náhodně vynechává určité procento neuronů, čímž se stává robustnější. *Dropout* je jen jedna z tzv. *regularizačních metod*, které předcházejí přeučení. Některé často používané datové soubory lze naimportovat přímo z TensorFlow pomocí modulu `tf.keras.datasets`.
3. Vytvoříme neuronovou síť pomocí třídy `tf.keras.Sequential(layers=None)` a uložíme ji do proměnné `model`. Na vstup této metody ve formě seznamu vložíme posloupnost vrstev, které definujeme pomocí modulu `tf.keras.layers`. Tento modul obsahuje velké množství druhů neuronových vrstev, které můžeme využít. V mém projektu jsem využil:

- `tf.keras.layers.Dense(units, activation=None)`
  Tato vrstva provádí klasickou dopřednou propagaci a na vstup přivádí počet neuronů (`units`) a aktivační funkci (`activation`). „`Dense`“ proto, že každý neuron této vrstvy je propojen s každým neuronem předcházející vrstvy.
- `tf.keras.layers.Conv2D(filters,`
  `kernel_size,`
  `strides=(1, 1),`
  `padding=“valid“)`

  Tato vrstva provádí konvoluci a na svůj vstup přivádí počet filtrů (`filters`), které se v této vrstvě neuronová síť naučí, dále rozměry konvoluční masky (`kernel_size`), velikost kroků, po kterých je konvoluční maska uplatňována po výšce a šířce snímku (`strides`) a parametr `padding`, který může nabývat buď hodnoty „`valid`“, kdy nebude uplatněn žádný padding, nebo „`same`“, kdy bude uplatněn zero padding. Pokud jsou parametry `strides` a `padding` nastaveny na hodnoty `strides=(1, 1)` a `padding=“same“`, pak výsledný snímek po konvoluci má stejné rozměry jako snímek před konvolucí.
- `tf.keras.layers.MaxPooling2D(pool_size=(2, 2),`
  `strides=None,`
  `padding=“valid“)`

  Tato vrstva provádí max pooling a na vstup přijímá parametr `pool_size`, což je v tomto ohledu ekvivalentní parametru `kernel_size` pro konvoluční vrstvu, a poté opět parametry `strides` a `padding`.
- `tf.keras.layers.Flatten()`
  Tato vrstva převede n-dimenzionální matici na vstupu vrstvy na vektor, se kterým může dále neuronová síť pracovat. Tímto způsobem lze například „zploštit“ vstupní snímek na vstupní vektor.

- `tf.keras.layers.Dropout(rate)`

  Tato vrstva pomáhá předcházet jevu přeučení tak, že při každé epoše učení se vždy vynechá určitý náhodně vybraný soubor neuronů uvnitř skrytých vrstev a vstupní vrstvy neuronové sítě. Na vstup přijímá parametr `rate` o hodnotě v rozmezí od 0 do 1 určující zlomek počtu neuronů, který se bude při učení vynechávat.

4. Poté pomocí metody `model.compile(optimizer="rmsprop", loss=None, metrics=None)` konfigurujeme optimalizační algoritmus (`optimizer`, například SGD – `"SGD"`), chybovou funkce (`loss`, například MSE – `"mean_squared_error"`) a ukazatele úspěšnosti neuronové sítě (`metrics`, například procentuální přesnost – `"accuracy"`). Zatímco chybová funkce se používá pro učení neuronové sítě, a tak nemusí být snadno interpretovatelná člověkem a její derivace musí mít opět vhodné vlastnosti, ukazatele úspěšnosti jako přesnost se používají pouze ke zhodnocení neuronové sítě, takže musí být snadno interpretovatelné a nezáleží na jejich derivaci.[4] Existují samozřejmě i jiné a lepší druhy a modifikace základních možností optimalizačních algoritmů, chybových funkcí a metrik, než které jsem uvedl. Některé z nich zmiňuji v kapitole. 6.2.
5. Na vstup metody `model.fit(x=None, y=None, batch_size=None, epochs=1)` vložíme trénovací vstupy (`x`), trénovací označení (`y`) a počet trénovačích epoch (`epochs`) a spustíme tak učení neuronové sítě. Parametr `batch_size` určuje velikost množin mini-batch pro učení (pokud není nastaven ručně, tak se nastaví na výchozí hodnotu 32).
6. Metoda `model.evaluate(x=None, y=None)` zprostředkovává testovací fázi. Na její vstup vložíme testovací vstupy (`x`) a testovací označení (`y`) a metoda nám vrátí výsledek chybové funkce a spočítanou procentuální přesnost (pokud jsme si ji vybrali jako metriku) neuronové sítě na testovacích datech. Nyní můžeme naučený model neuronové sítě používat.

7. Pokud na vstup metody `model.predict(x)` vložíme nová vstupní data (x), dostaneme nové predikce, které by měly být stejně přesné, jako byly predikce v testovací fázi.

Obrázek 15: Postup implementace neuronových sítí pomocí v Keras

```
1 import tensorflow as tf

2 (X_train, Y_train), (X_test, Y_test) = \
  tf.keras.datasets.fashion_mnist.load_data()

3 model = tf.keras.models.Sequential([
      tf.keras.layers.Flatten(input_shape=(28, 28, 1)),
      tf.keras.layers.Dense(300, activation="relu"),
      tf.keras.layers.Dense(100, activation="relu"),
      tf.keras.layers.Dense(10, activation="softmax")
  ])

4 model.compile(optimizer="adam",
                loss="sparse_categorical_crossentropy",
                metrics=["accuracy"])

5 model.fit(X_train, Y_train, epochs=5)

6 test_loss, test_accuracy = model.evaluate(X_test, Y_test)
  print(f"Chybová funkce na testovacích datech: {test_loss}\n\
  Přesnost na testovacích datech: {test_accuracy}")

7 predictions = model.predict(X_test)
```

Zdroj: Vlastní zpracování pomocí , koncepce ze zdroje [15]

Obrázek 16: Průběh učení a zhodnocení modelu z programu na obrázku č. 15

```
Epoch 1/5
1875/1875 [==============================] - 63s 7ms/step - loss: 1.9815 - accuracy: 0.7448
Epoch 2/5
1875/1875 [==============================] - 14s 7ms/step - loss: 0.5684 - accuracy: 0.8062
Epoch 3/5
1875/1875 [==============================] - 14s 7ms/step - loss: 0.5006 - accuracy: 0.8264
Epoch 4/5
1875/1875 [==============================] - 14s 7ms/step - loss: 0.4482 - accuracy: 0.8437
Epoch 5/5
1875/1875 [==============================] - 14s 7ms/step - loss: 0.4226 - accuracy: 0.8495
313/313 [==============================] - 1s 3ms/step - loss: 0.4759 - accuracy: 0.8359
Chybová funkce na testovacích datech: 0.4759264588356018
Přesnost na testovacích datech: 0.8359000086784363
```

Zdroj: Vlastní zpracování pomocí https://snappify.com/

6. Porovnání výstupních neuronových sítí

6.1 Hyperparametry a validační data

Parametry neuronové sítě jsou nastaveny učením. Proces učení však může do jisté míry ovládat programátor, a to nastavením tzv. *hyperparametrů*. Mezi hyperparametry patří především: počet vrstev neuronové sítě, druhy jednotlivých vrstev (a tak i druhy aktivačních funkcí použitých v jednotlivých vrstvách), počet neuronů v jednotlivých vrstvách, parametr učení, velikost množin mini-batch při učení, počet epoch učení a druh samotného optimalizačního algoritmu. Jednotlivé hyperparametry zčásti ozřejmím porovnáním mých výstupních neuronových sítí. Pro stanovení hyperparametrů neexistuje formalizovaná metoda, a tak závisí na experimentaci (po letech výzkumu samozřejmě existují určité obecné směrnice nastavení hyperparametrů pro určité druhy problémů).[4]

Pro učení a testování se používají oddělené množiny dat, abychom poznali, zda je neuronová síť přeučená. Pokud ale volíme hyperparametry na základě toho, jak je daný model úspěšný na testovacích datech, pak zase můžeme způsobovat přeučení na testovacích datech, protože hyperparametry přizpůsobujeme právě testovací množině dat. Proto se datové soubory dále dělí na trénovací, testovací a *validační množinu*. Validační množina dat pak slouží ke zhodnocení modelů s různě nastavenými hyperparametry, z nichž vybereme ten optimální. Stejně jako testovací množina by ta validační měla být reprezentativním vzorkem dat, se kterými se bude výsledný model potýkat v praxi.[4]

6.2 Aktivační funkce, chybová funkce, optimalizační algoritmus a počet epoch učení

Pro oba modely vytvořené pomocí mé vlastní neuronové sítě platí, že všechny skryté vrstvy společně s výstupní vrstvou používají logistickou aktivační funkci. Podle dříve popsaných matematických vztahů jsem pro chybovou funkci implementoval střední kvadratickou chybu a pro optimalizační algoritmus stochastický gradientní

sestup. Implementoval jsem tedy opravdu základní typ neuronové sítě, který se dnes v praxi už nepoužívá.

Pro oba modely vytvořené pomocí TensorFlow Keras platí, že kromě vrstev pro pooling, flattening a dropout jsem pro všechny skryté vrstvy použil aktivační funkci ReLU a pro výstupní vrstvu jsem použil aktivační funkci Softmax, a to z důvodů zmíněných v kapitole 2.3. Jako chybovou funkci jsem zvolil tzv. *diskrétní křížovou entropii* (angl. *categorical crossentropy*) a jako optimalizační algoritmus jsem zvolil tzv. *Nadam* (angl.. *Nesterov-accelerated Adaptive Moment Estimation*). Tuto chybovou funkci a optimalizační algoritmus jsem zvolil na doporučení ze zdroje [4], proto na něj taky odkazuji k bližšímu popisu těchto metod a jejich praktickým vlastnostem. Tato konfigurace hyperparametrů je dnes běžně používaná pro řešení problému klasifikace.

Moc velký počet epoch učení také může vést k přeučení, Pro oba datové soubory jsem tak vždy zvolil 30 epoch učení, což se ukázalo jako dost, aby modely dosáhly požadovaných přesností, a dost málo na to, aby k přeučení nedošlo. Učení neuronových sítí také chvíli trvá, takže jsem zvolil 30 epoch i z časového hlediska.

Zatímco pro stochastický gradientní sestup je v TensorFlow Keras výchozí hodnota 0,01, má vlastní neuronová síť funguje uspokojivě při hodnotě 0,1 pro datový soubor Fashion MNIST a při hodnotě 0,5 pro datový soubor GTSRB, protože při menších hodnotách byl SGD příliš pomalý. Výchozí hodnota parametru učení pro optimalizační algoritmus Nadam, kterou jsem pro můj model nezměnil, je 0,001.

6.3 KLASIFIKACE FASHION MNIST

Tento datový soubor jsem importoval pomocí modulu `tf.keras.datasets`.

Má vlastní neuronová síť pro klasifikaci snímků oblečení (viz obr. 17) má 2 skryté vrstvy o počtu neuronů 32 a 16, velikost mini-batch množin je 10 vstupních vektorů a parametr učení je 0,1. Výsledný model má 84,38% přesnost klasifikace na

testovacích datech. Vstupní vrstva obsahuje 728 neuronů, protože černobílé snímky datového souboru Fashion MNIST mají rozměry 28x28 px.

Obrázek 17: Má vlastní neuronová síť pro klasifikaci Fashion MNIST

```
custom_model = Artificial_Neural_Network((784, 32, 16, 10))
custom_model.stochastic_gradient_descent(X_train_flattened, y_train_encoded,
                                         epochs=30, mini_batch_size=10, learning_rate=0.1,
                                         validation_data=(X_valid_flattened, y_valid_encoded))
```

Zdroj: Vlastní zpracování pomocí https://snappify.com/

Vrstvy a jejich vlastnosti pro konvoluční neuronovou síť vytvořenou pomocí TensorFlow Keras lze vyčíst z obrázku č. 18 pomocí popisů použitých metod v kapitole 5.2. Výsledný model dosahuje 91,75% přesnosti na testovacích datech.

Grafy na obrázku č. 19 a 20 zobrazují, jak se s každou epochou vyvíjí přesnost a chybová funkce odvozená z trénovacích a validačních dat. Lze vidět, že ani jeden model nepodléhá přeučení, protože nevzniká velká mezera mezi trénovacími a

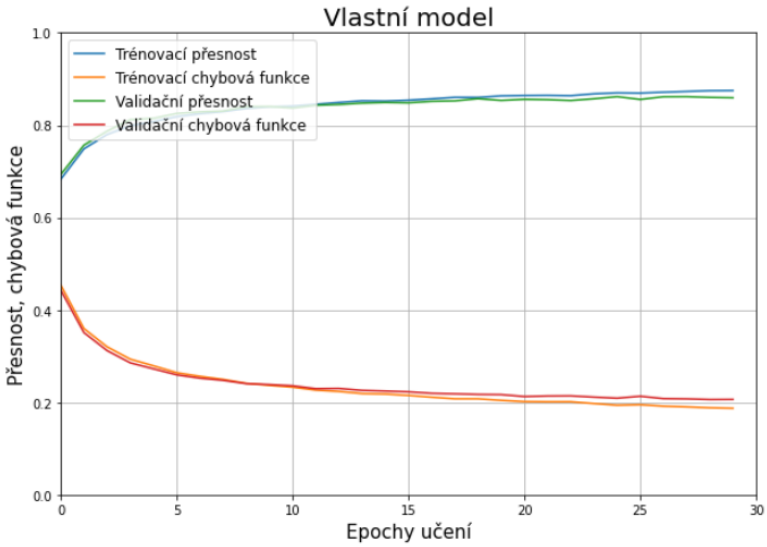
Obrázek 18: Keras neuronová síť pro klasifikaci Fashion MNIST

```
keras_model = tf.keras.models.Sequential([
      tf.keras.layers.Conv2D(32, (3, 3), activation="relu",
                             input_shape=(28, 28, 1)),
      tf.keras.layers.Conv2D(32, (3, 3), activation="relu"),
      tf.keras.layers.MaxPooling2D((2, 2)),
      tf.keras.layers.Dropout(0.25),
      tf.keras.layers.Conv2D(64, (3, 3), activation="relu"),
      tf.keras.layers.MaxPooling2D((2, 2)),
      tf.keras.layers.Dropout(0.25),
      tf.keras.layers.Flatten(),
      tf.keras.layers.Dense(128, activation="relu"),
      tf.keras.layers.Dropout(0.5),
      tf.keras.layers.Dense(64, activation="relu"),
      tf.keras.layers.Dropout(0.5),
      tf.keras.layers.Dense(10, activation="softmax")
])
```

Zdroj: Vlastní zpracování pomocí https://snappify.com/

validačními křivkami. U mého modelu vytvořeného pomocí mé vlastní neuronové sítě je to zapříčiněno velmi malou velikostí množin mini-batch pro SGD. Při takto malých velikostech neuronová síť odhaduje skutečný gradient vzhledem k parametrům neuronové sítě s menší přesností, což má podobný efekt, jako když přidáme vrstvu pro dropout, který do učení v podstatě přidává šum. Díky tomu je výsledný model obecnější a nedochází k přeučení.[18]

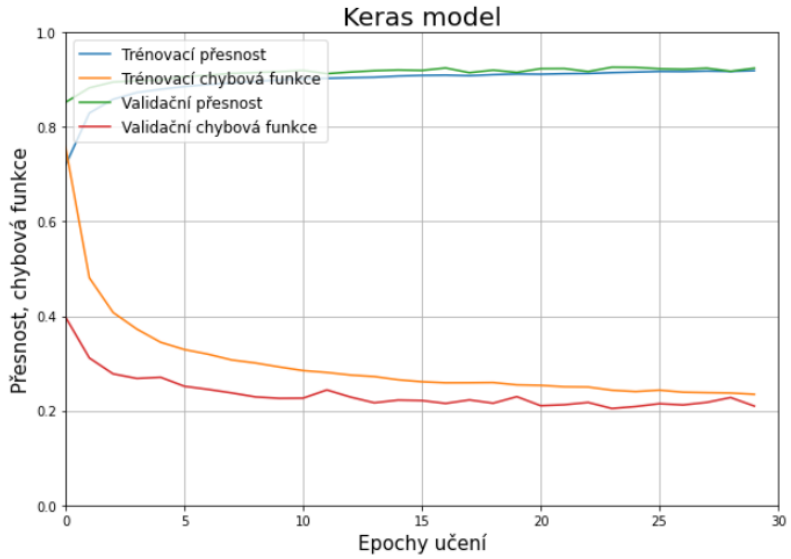
Obrázek 19: Graf průběhu učení mé vlastní neuronové sítě pro Fashion MNIST



Zdroj: Vlastní zpracování

Model vytvořený pomocí TensorFlow Keras je komplexnější a má daleko více parametrů, takže k zamezení přeučení jsem musel použít vrstvu `tf.keras.layers.Dropout(rate=0.5)`. Určitá logika je ve zvyšování počtu konvolučních masek za každou vrstvou pro pooling. Jelikož parametr `pooling_size` je nastaven na hodnotu `(2, 2)`, takže šířka a délka snímku se podělí dvěma, můžu si dovolit zdvojnásobit počet konvolučních masek, aniž bych se bál vysokého počtu parametrů a výpočetní náročnosti.[4] S výjimkou první konvoluční vrstvy obecně lépe fungují menší konvoluční masky, které používají méně parametrů. V první vrstvě si můžeme dovolit větší konvoluční masku, protože zmenšíme rozměry snímku, aniž bychom přišli o velké množství informací.[4]

Obrázek 20: Graf průběhu učení Keras neuronové sítě pro Fashion MNIST



Zdroj: Vlastní zpracování

6.4 KLASIFIKACE GTSRB

Tento datový soubor jsem importoval pomocí služby Activeloop Deeplake,(která mi umožnila:

1. jednoduše nahrát data z cloudové databáze tak, abych se nemusel příliš zaobírat jejich vstupním formátem a následným zpracováním.
2. vizualizovat datový soubor formou aplikace, která poskytuje možnost data interaktivně prozkoumat přímo v Jupyter notebooku.

Má vlastní neuronová síť pro klasifikaci dopravních značek (viz obr. 21) má 2 skryté vrstvy o počtech neuronů 128 a 64. Vstupní vrstva má 2700 neuronů, protože rozměry barevných snímků jsem normalizoval na 30x30 px (x3 dimenze barvy). To znamená, že u větších snímků datového souboru (kolem 100x100 px a více) jsem poměrně hodně zredukoval jejich rozměry. Při větších velikostech totiž docházelo k většímu přeučení a výsledný model nedosahoval lepší přesnosti, protože komplexnější vstupní informace vyžadují komplexnější neuronovou síť, než kterou jsem vytvořil pomocí Keras. Velikost 30x30 px se ukázala jako dostatečná pro dosáhnutí mého cíle a zamezení velkému přeučení.

Obrázek 21: Má vlastní neuronová síť pro klasifikaci GTSRB

```
custom_model = Artificial_Neural_Network((2700, 128, 64, 43))
custom_model.stochastic_gradient_descent(X_train_flattened, y_train_encoded,
                                         epochs=30, mini_batch_size=10, learning_rate=0.1,
                                         validation_data=(X_valid_flattened, y_valid_encoded))
```

Zdroj: Vlastní zpracování pomocí https://snappify.com/

Vrstvy a jejich vlastnosti pro konvoluční neuronovou síť vytvořenou pomocí TensorFlow Keras lze opět vyčíst z obrázku 22.

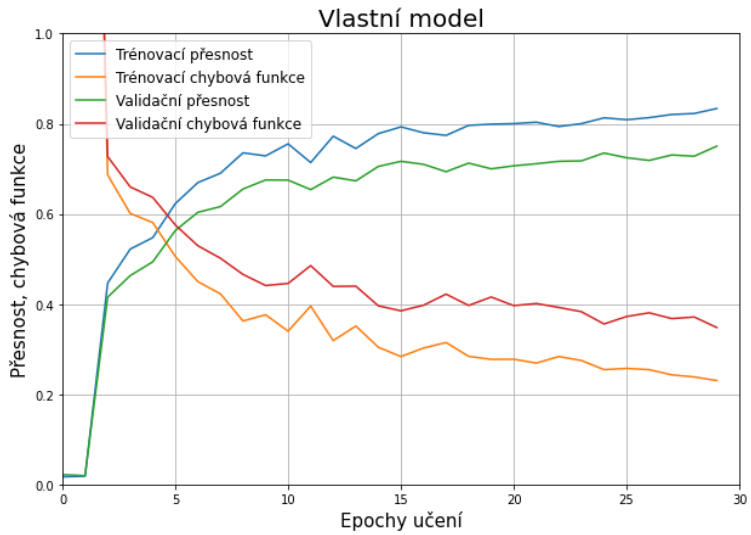
Obrázek 22: Keras neuronová síť pro klasifikaci GTSRB

```
keras_model = tf.keras.models.Sequential([
    tf.keras.layers.Conv2D(64, (3, 3), activation="relu",
                           input_shape=(30, 30, 3)),
    tf.keras.layers.Conv2D(64, (3, 3), activation="relu"),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Conv2D(128, (3, 3), activation="relu"),
    tf.keras.layers.Conv2D(128, (3, 3), activation="relu"),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Conv2D(256, (3, 3), activation="relu"),
    tf.keras.layers.MaxPooling2D((2, 2)),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(128, activation="relu"),
    tf.keras.layers.Dropout(0.5),
    tf.keras.layers.Dense(43, activation="softmax")
])
```

Zdroj: Vlastní zpracování pomocí https://snappify.com/

Zde jsou lépe vidět následky chybějící konvoluce a neoptimálně zvolených aktivačních funkcí, chybové funkce a optimalizačního algoritmu, protože tento model dosáhl testovací přesnosti jen 75,83 %. Neschopnost vytvořit přesný model lze vidět i v grafu na obr. 23. Model by šel trochu zpřesnit zvětšením mini-batch množin, což by ale ještě více zpomalilo už tak pro tento úkol hodně pomalou neuronovou síť.

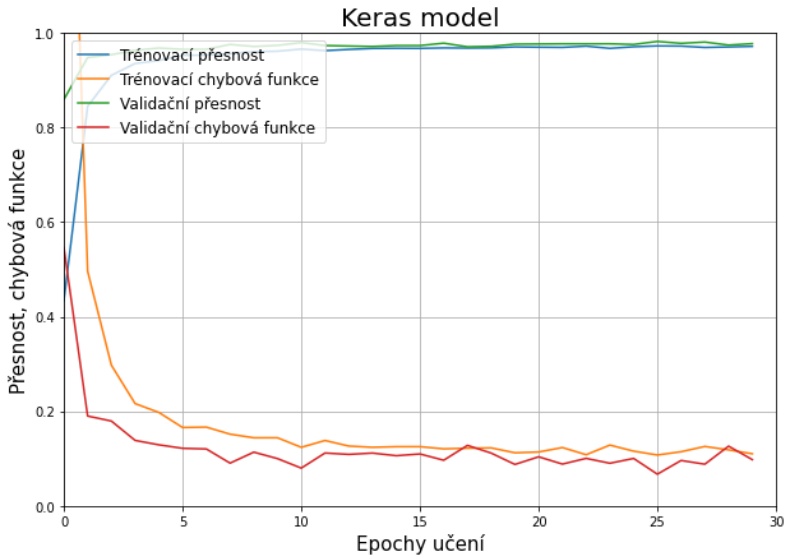
Obrázek 23: Průběh učení mé vlastní neuronové sítě na GTSRB



Zdroj: Vlastní zpracování

Výsledný model vytvořený pomocí TensorFlow Keras dosahuje testovací přesnosti 98,19%. K této neuronové síti jsem došel delší experimentací a zkoušením různých kombinací vrstev a jejich vlastností. Průběh učení lze vidět v grafu na obr. 24. Podobný model, jaký jsem použil pro GTSRB, je ve zdroji [4] znázorněný pro klasifikaci Fashion MNIST.

Obrázek 24: Průběh učení Keras neuronové sítě na GTSRB



Zdroj: Vlastní zpracování

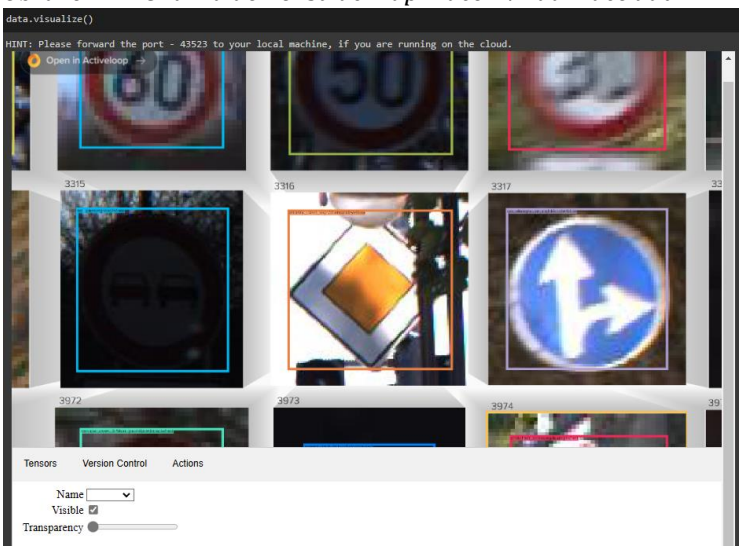
Ilustrační predikce všech modelů jsou zobrazeny v Jupyter noteboocích v příloze.

7. Demonstrace funkčního modelu pro klasifikaci dopravních značek

Abych ukázal, že výsledný model lze nyní v praxi použít, exportoval jsem ho pomocí metod `tf.keras.Model.save(filepath)`, kde `filepath` je cílová lokace uložení i se jménem exportovaného modelu. Poté jsem ho v demonstračním Jupyter notebooku importoval pomocí `tf.keras.models.load_model(filepath)`.

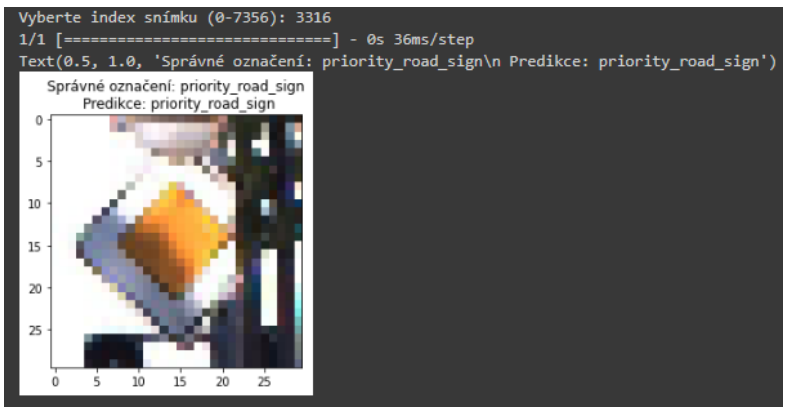
Po importování knihoven a načtení dat si uživatel může najít určitý snímek dopravní značky ve vizualizačním rozhraní zprostředkovaném modulem Deeplake. Poté může na dotaz skriptu další buňky zadat index tohoto snímku a na výstupu může porovnat predikci modelu se správným označením:

Obrázek 26: Ukázka demonstrační aplikace - vizualizace dat



Zdroj: Vlastní zpracování

Obrázek 25: Ukázka demonstrační aplikace - predikce modelu



Zdroj: Vlastní zpracování

8. Co dál?

Pokud bychom chtěli dosáhnout ještě větší přesnosti klasifikace dopravních značek nebo vytvořit model, který dokáže rozpoznat více objektů dopravy, mohli bychom použít vyspělé architektury konvolučních neuronových sítí (jako je např. ResNet), které byly úspěšné v soutěži ILSVRC (image-net.org). Tyto architektury jsou popsané ve zdroji [4] a jsou přístupné k použití a modifikaci v modulu `tf.keras.applications`.

Pokud bychom chtěli vytvořit všestranného agenta, který se dokáže orientovat v dopravě, museli bychom implementovat:

- Lokalizaci a klasifikaci objektů ve snímaných vizuálních informacích (tento úkol se celkově označuje jako *object detection*). Viz architektura YOLO popsaná ve zdroji [4].
- Sémantickou segmentaci - klasifikaci každého pixelu podle toho, jakému objektu ve snímku náleží, čili nalezení přesných ohraničení objektů na snímku.

Pokud pracujeme se sekvencí dat, jako například při problémech rozpoznání mluveného slova (tedy práce se zvukem), při práci s videem nebo při problémech vyžadujících modely zpracování jazyka (viz GPT-4), budeme potřebovat tzv. *rekurentní neuronové sítě* (*RNN*), které mohou svůj výstup použít jako další informaci pro svůj vstup a tím pádem také produkovat sekvenci dat na svém výstupu.

Pro generování dat, například generování snímků (viz DALL-E, MidJourney, StableDiffusion) nebo videa (viz GEN-2) z textového zadání, bychom potřebovali tzv. *generativní adversariální sítě* (*GAN*).

Pokud bychom chtěli, aby se náš agent dokázal v dopravě také správně rozhodovat, mohli bychom použít neuronové sítě pro tzv. *zpětnovazební učení*, které je inspirováno behavioristickou psychologií.

ZÁVĚR

Podle bodů zadání:

- Technologii neuronových sítí jsem popsal v kapitole č. 2, 3 a 4. Tento bod byl z hlediska vysvětlených konceptů společně s tvorbou vlastní neuronové sítě podle těchto konceptů nejsložitější a zabral nejvíce času. Prvotní inspiraci pro tvorbu mé vlastní neuronové sítě jsem dostal po přečtení první kapitoly knihy [9]. V této knize je taktéž implementována ukázková neuronová síť pouze pomocí Pythonu a NumPy, ale během programování mé vlastní neuronové sítě jsem ji použil pouze k porovnání výstupů. Před mou vlastní implementací jsem viděl i podobné výtvory jiných lidí, ale má vlastní neuronová síť byla celá realizována pouze na základě matematických vztahů popsaných v tomto dokumentu. Původní neuronová síť byla velice pomalá, protože jsem implementoval původní gradientní sestup, parametry jsem aktualizoval skalárně a algoritmus zpětného šíření jsem také počítal skalárně. Pro zrychlení jsem tedy vykonal již dříve zmíněné změny – naprogramoval jsem SGD a všechny výpočty realizoval maticově, přičemž vztahy (24), (25) a (26) jsem na maticové vztahy (27), (28) a (29) zobecnil sám. Jako nedostatek vidím to, že jsem nerealizoval časování stochastického gradientního sestupu, takže jsem nemohl výsledné neuronové sítě porovnat z hlediska času učení. Můj cíl 80% přesnosti klasifikace Fashion MNIST pomocí mé vlastní neuronové sítě jsem překonal asi o 4 %. Samotnou funkci stochastického gradientního sestupu považuji za úspěch.
- Charakteristiku nástroje TensorFlow jsem zpracoval v kapitole č. 5. Nešel jsem do hloubky této knihovny, ale zaměřil jsem se na vysokoúrovňovou API Keras, na které jsem vysvětlil základní postup tvorby, učení, zhodnocení a konečného použití neuronových sítí.
- Tvorbu v TensorFlow s implementací vlastní neuronové sítě jsem porovnal v kapitole č. 6. Při tvorbě neuronových sítích v TensorFlow jsem se inspiroval ukázkovými neuronovými sítěmi v knize [4], ale než jsem došel ke konečným verzím, musel jsem vyzkoušet několik jiných možností (cca kolem desíti pro

každý datový soubor), ve kterých jsem měnil počet konvolučních vrstev, počet a velikost konvolučních masek, počet klasických skrytých vrstev a počet neuronů v těchto vrstvách a počet vrstev pro dropout a jejich parametr `rate`. Je zajímavé, že pro datový soubor Fashion MNIST je uvedenou základní tvorbou v Keras velice složité dostat se nad přesnost 92 %, zatímco pro datový soubor GTSRB tuto hranici překonáme velice snadno. Zároveň ale pro překonání cca 80% přesnosti při klasifikaci datového souboru Fashion MNIST nám stačí základní a dnes již zastaralé metody použité v mé vlastní neuronové síti, zatímco pro překonání této hranice při klasifikaci GTSRB potřebujeme použít vyspělejší metody jako je konvoluce, regularizace a vhodnější aktivační funkce, chybovou funkci a optimalizační algoritmus. To všechno je pravděpodobně způsobeno tím, že i když jsou snímky ze souboru GTSRB komplexnější, takže vyžadují komplexnější neuronovou síť, snímky ze souboru Fashion MNIST vykazují větší variaci uvnitř jednotlivých kategorií, takže je složitější dosáhnout opravdu vysoké přesnosti. Jako nedostatek modelu pro GTSRB vidím to, že validační množina dat se standardně vybírá z trénovací množiny dat, ale když jsem náhodně promíchal trénovací množinu dat a vybral z ní validační data, z nějakého důvodu validační data reprezentovala více trénovací data než ty testovací, takže přesnost na validačních datech byla větší než na tetovacích datech. Proto jsem validační množinu dat dostal rozpůlením testovací množiny dat, čímž jsem ale zvětšil poměr trénovací množiny k testovací množině a zmenšil objem testovacích dat. Tím jsem možná způsobil lehce větší nepřesnost při zhodnocení modelu. Konečná testovací množina dat ale i po půlení obsahuje 3000 snímků, což by mělo být dostačující. Můj cíl 90% přesnosti klasifikace GTSRB pomocí neuronové sítě vytvořené pomocí TensorFlow Keras jsem překonal asi o 8 %.

- Vlastní aplikaci vybrané technologie jsem zdokumentoval v kapitole č. 7. V demonstrační aplikaci jsem ukázal, že výsledný model lze úspěšně exportovat a kdokoliv ho může použít v praxi.

SEZNAM POUŽITÉ LITERATURY

[1] BACH, Joscha. Synthetic Intelligence. In: *[YouTube - DigitalFUTURES DOCTORAL CONSORTIUM]* [online]. 13. 2. 2022 [cit. 2023-03-20]. Dostupné z: https://www.youtube.com/watch?v=pB-pwXU0I4M

[2] HUTTER, Marcus. Foundations of Intelligent Agents. In: *[YouTube - AI: Singularity Summit 2010]* [online]. [cit. 2023-03-20]. Dostupné z: https://www.youtube.com/watch?v=x8btbKaRfoc

[3] RUSSELL, Stuart J. a Peter NORVIG. *Artificial Intelligence: a modern approach*. 4th ed. Harlow: Pearson Education, 2021. ISBN 9781292401133.

[4] GÉRON, Aurélien. *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: concepts, tools, and techniques to build intelligent systems*. Second edition. Beijing: O'Reilly, 2019. ISBN 9781492032649.

[5] BROWNLEE, Jason. Difference Between Algorithm and Model in Machine Learning. *Machine Learning Mastery* [online]. 29. 4. 2020 [cit. 2023-03-18]. Dostupné z: https://machinelearningmastery.com/difference-between-algorithm-and-model-in-machine-learning/

[6] BROWNLEE, Jason. What is a Hypothesis in Machine Learning?. *Machine Learning Mastery* [online]. [cit. 2023-03-18]. Dostupné z: https://machinelearningmastery.com/what-is-a-hypothesis-in-machine-learning/

[7] XIAO, Han, Kashif RASUL a Roland VOLLGRAF. *Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms* [online]. 15. 9. 2017 [cit. 2023-03-18]. Dostupné z: https://arxiv.org/abs/1708.07747

[8] STALLKAMP, Johannes, Marc SCHLIPSING, Jan SALMEN a Christian IGEL. *Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition* [online]. 20. 2. 2012 [cit. 2023-03-18]. Dostupné z: https://www.sciencedirect.com/science/article/abs/pii/S0893608012000457

[9] NIELSEN, Michael A. *Neural Networks and Deep Learning* [online]. Determination Press, 2015 [cit. 2023-03-18]. Dostupné z: http://neuralnetworksanddeeplearning.com/

[10] SANDERSON, Grant. Neural Networks. In: *[YouTube - 3Blue1Brown]* [online]. 2017 [cit. 2023-03-18]. Dostupné z: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi

[11] YU, Brian a David J. MALAN. Neural Networks - Lecture 5 - CS50's Introduction to Artificial Intelligence with Python 2020. In: *[YouTube - CS50]* [online]. 10. 4. [cit. 2023-03-18]. Dostupné z: https://www.youtube.com/watch?v=mFZazxxCKbw&list=PLBw9d_OueVJS_084gYQexJ38LC2LEhpR4&index=6

[12] KIRSANOV, Artem. Dendrites: Why Biological Neurons Are Deep Neural Networks. In: *[YouTube - Artem Kirsanov]* [online]. 29. 1. 2023 [cit. 2023-03-20]. Dostupné z: https://www.youtube.com/watch?v=hmtQPrH-gC4

[13] Artificial Neural Networks. In: *Brilliant* [online]. [cit. 2023-03-27]. Dostupné z: https://brilliant.org/courses/artificial-neural-networks/

[14] BODEN, Margaret A. *Artificial Intelligence: A Very Short Introduction*. New York: Oxford University Press, 2018. ISBN 978019960291

[15] FRIDMAN, Lex. Deep Learning Basics: Introduction and Overview. In: *MIT Deep Learning and Artificial Intelligence Lectures* [online]. [cit. 2023-03-18]. Dostupné z: https://deeplearning.mit.edu/

[16] *[TensorFlow dokumentace]* [online]. [cit. 2023-03-18]. Dostupné z: https://www.tensorflow.org/api_docs

[17] *[Keras API dokumentace]* [online]. [cit. 2023-03-18]. Dostupné z: https://keras.io/api/

[18] GOODFELLOW, Ian, Yoshua BENGIO a Aaron COURVILLE. *Deep learning* [online]. Cambridge, Massachusetts: The MIT Press, [2016] [cit. 2023-03-20]. ISBN 978-0262035613. Dostupné z: https://www.deeplearningbook.org/

[19] DANCUK, Milica. PyTorch vs TensorFlow: In-Depth Comparison. In: *PhoenixNAP* [online]. 23. 2. 2021 [cit. 2023-03-27]. Dostupné z: https://phoenixnap.com/blog/pytorch-vs-tensorflow

SEZNAM OBRÁZKŮ A TABULEK

PŘÍLOHY

Příloha č. 1: Poster k maturitní práci.

„DMP_Neuronove_site_poster"

Příloha č. 2 : Program mé vlastní neuronové sítě.

„custom_ann.py“

Příloha č. 3: Jupyter notebooky pro učení a porovnání neuronových sítí

„fashion_minst_classification.ipynb“

„fashion_mnist_classification.pdf“

„gtsrb_classification.ipynb“

„gtsrb_classification.pdf“

Příloha č. 4: Jupyter notebook demonstrační aplikace konečného modelu a samotný konečný model.

„keras_gtsrb_model.h5“

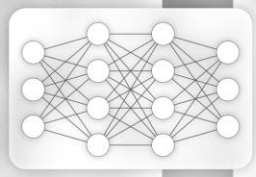
„gtsrb_model_demonstration.ipynb“

Neuronové sítě

Jindřich Machka, VOŠ a SPŠE Olomouc, 4.A, 2023
Vedoucí práce: Mgr. Jaroslav Krbec

Úvod

Ať už jde o vyvozování užitečných informací z vizuálních či zvukových vstupů, autonomní chování dosáhnuté zpětnovazebním učením, generování vlastních snímků z textového zadání nebo čím dál všestrannější vlastnosti velkých modelů pro zpracování jazyka, díky schopnosti neuronových sítí výpočetně reprezentovat vzory v datech s ohledem na určitý cíl, a tak aproximovat jakoukoliv funkci s libovolnou přesností, se umělá inteligence stala středobodem technologického vývoje 21. století.



Cíl práce

Cílem mé práce bylo v programovacím jazyce Python s pomocí knihovny NumPy pro zprostředkování maticových operací vytvořit vlastní neuronovou síť a na problému klasifikace snímků ji porovnat s neuronovou sítí vytvořenou pomocí nejpoužívanějšího nástroje pro tvorbu neuronových sítí v praxi - pomocí knihovny TensorFlow a API Keras.

Zhodnocení

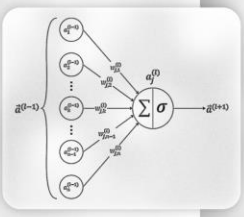
Má vlastní neuronová síť při klasifikaci malých snímků oblečení z datového souboru Fashion MNIST dosahuje na testovací množině dat přesnosti 84 %, ale kvůli chybějící konvoluci, neoptimálně zvoleným aktivačním funkcím a optimalizačnímu algoritmu výsledný model nestačí na klasifikaci dopravních značek z datového souboru GTSRB. Na tento úkol jsem tedy využil konvoluční neuronovou síť vytvořenou pomocí TensorFlow Keras, která při klasifikaci dopravních značek dosahuje přesnosti 98 %.

Funkce neuronových sítí

Umělý neuron

Umělý neuron je paralelní výpočetní jednotka, jejíž výstup, který je poslán na vstup všech neuronů následující vrstvy neuronové sítě, je dán aktivační funkcí, která na vstup přijímá vážený součet vstupů daného neuronu s přičteným prahem neuronu.

- Váhy, které udávají sílu spojení daného neuronu s neurony z předešlé vrstvy, a práh, který způsobuje posun aktivační funkce, jsou parametry neuronové sítě.
- Aktivační funkce: skoková, logistická, ReLU, Softmax

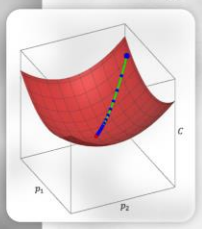


Dopředná propagace

Dopředná propagace je postup šíření a transformace vstupních informací skrze neuronovou síť produkující výstup neuronové sítě.

Optimalizační algoritmus

Optimalizační algoritmus zajišťuje učení neuronové sítě tím, že na základě chybovosti neuronové sítě iterativně upravuje parametry jednotlivých neuronů na optimální hodnoty, a tak minimalizuje chybovou funkci. S klesající chybovou funkcí by měla růst přesnost predikcí neuronové sítě.



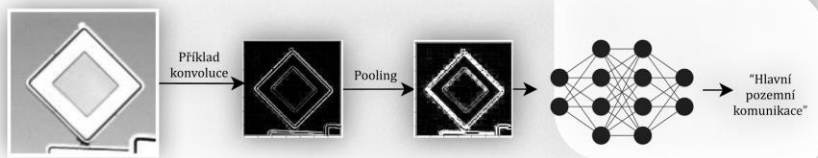
- Optimalizační algoritmy: Stochastický gradientní sestup, Nadam
- Chybové funkce: střední kvadratická chyba, diskrétní křížová entropie

Zpětná propagace

Zpětná propagace je zpětný postup zjišťování citlivosti změny výstupu neuronové sítě na změnu každého parametru každého neuronu, díky které může optimalizační algoritmu parametry náležitě upravit.

Konvoluce a pooling

Metody umožňující neuronové síti učit se na základě extrahovaných vzorů ve snímku a jeho vlastností, aby neuronová síť nemusela hledat strukturu v původním dlouhém číselném vektoru.



Použil jsem v mé vlastní neuronové síti. Použil jsem při tvorbě konvoluční neuronové sítě pomocí Tensorflow Keras.