ČESKÉ VYSOKÉ
UČENÍ TECHNICKÉ
V PRAZE

F3

Fakulta elektrotechnická
Katedra počítačů

Bakalářská práce

Neuronové sítě pro SŠ

Antonín Mašek

Květen 2020
Vedoucí práce: Ing. Božena Mannová, Ph.D.

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Mašek** Jméno: **Antonín** Osobní číslo: **466194**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Softwarové inženýrství a technologie**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Neuronové sítě pro SŠ - výukový program

Název bakalářské práce anglicky:

Neural Networks for High Schools - educational program

Pokyny pro vypracování:

Cílem práce je vytvoření systému pro podporu výuky tématu „neuronové sítě“ na středních školách. Vytvořený systém bude demonstrovat implementaci konkrétního řešení neuronové sítě na rozpoznávání ručně psaných číslic.
Jako data pro zpracování ručně psaných číslic použijte již existující dataset MNIST. Dataset rozdělte na tréninkovou a testovací část.
Seznamte se s problematikou neuronových sítí a datasetem MNIST. Navrhněte výukový systém a zvolte vhodné implementační prostředky. Systém implementujte a ověřte. Přesnost řešení ověřte na testovací části datasetu

Seznam doporučené literatury:

[1] Co je strojové učení. https://www.oracle.com/cz/artificial-intelligence/what-is-machine-learning. html.
[2] Helena, Kučerová. KTD: Česká terminologická databáze knihovnictví a informační vědy (TDKIV) [online]. Praha : Národní knihovna ČR.
[3] Goodfellow, Ian. Deep Learning. MIT Press, 2017. ISBN 9780262035613. http://www.deeplearningbook.org/.
[4] LeCun, Yann, Corinna Cortes a Chris Burges. MNIST Handwritten Digit Database. http://yann.lecun.com/exdb/mnist/.

Jméno a pracoviště vedoucí(ho) bakalářské práce:

Ing. Božena Mannová, Ph.D., kabinet výuky informatiky FEL

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

Datum zadání bakalářské práce: **13.02.2020** Termín odevzdání bakalářské práce: **22.05.2020**

Platnost zadání bakalářské práce: **30.09.2021**

Ing. Božena Mannová, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

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

. | Datum převzetí zadání

Podpis studenta

Poděkování / Prohlášení

Děkuji Ing. Boženě Mannové, PhD. za to, že se ujala mého tématu a doufám, že bude s výsledkem mého snažení spokojena.

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

V Praze dne 21. 5. 2020

.....

Abstrakt / Abstract

Tento dokument má za cíl vytvořit výukový materiál, který by objasnil základy fungování neuronových sítí primárně pro čtenáře (nejen středních škol), kteří mají základy diferenciálního počtu. V průběhu práce bude vysvětlena teorie, která bude na závěr využita v praxi na reálné implementaci neuronové sítě. Přesnost sítě bude následně ověřena.

Klíčová slova: neuronová síť, strojové učení, mnist dataset, rozpoznávání číslic, backpropagation

This document aims to create a study material that would clarify the basics of Neural Networks primarily for readers that have a basic understanding of calculus. The basic theory behind neural networks will be explained and at the later chapters, it will be put to practice. Finally, the precision will be checked.

Keywords: neural network, machine learning, mnist dataset, recognizing digits, backpropagation

Obsah /

Tabulky / Obrázky

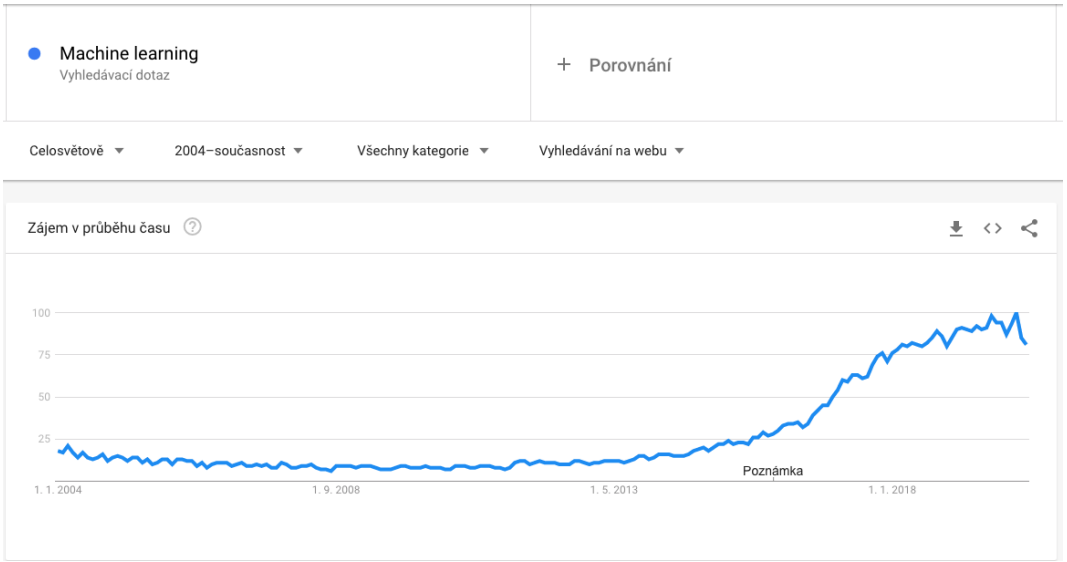
Kapitola 1
Úvod

V průběhu posledních let je stále více slyšet o umělé inteligenci (AI) a to nejen v odborných kruzích. Technologické společnosti zmiňují využití strojového učení (ML) ve svých produktech, montují podpůrné čipy a dokonce připravují API pro vývojáře. Ovšem co to neuronové sítě jsou, proč je o nich tak často slyšet ve spojitosti s AI, ML a co tyto pojmy vlastně znamenají a jak se od sebe liší? A především jak vlastně samotná neuronová síť v nejzákladnějším provedení funguje, tak na tyto otázky se pokusí tento dokument odpovědět srozumitelným jazykem určeným především pro čtenáře, kteří se v této tématice chtějí teprve zorientovat a nechtějí, aby pro ně slovní spojení „Neuronové sítě“ bylo pouhou „magií“.

1.1 Pro koho je tato práce určena

Ač je název této práce „Neuronové sítě pro SŠ“, tak při jejím psaní nebylo rozhodně zamýšleno se omezit pouze na středoškolské publikum. Práce je vhodná pro kohokoliv, kdo má alespoň základy diferenciálního počtu a nejlépe nějakého programovacího jazyka.

Cílem práce je využít již značných kvalitních zdrojů, které jsou naneštěstí většinou v anglickém jazyce, což je škoda pro české prostředí už jen kvůli tomu, že popularita neuronových sítí a strojového učení v poslední době vysoce stoupá.



Obrázek 1.1. Trend vyhledávání pojmu Machine Learning na Google (2004 - 2020)

1.2 Co by měl čtenář po přečtení umět

Po přečtení by měl mít čtenář přehled o základních pojmech, které se vyskytují kolem neuronových sítí a strojového učení. Zároveň si napíše vlastní neuronovou síť, takže by

měl být schopen porozumět principu, na kterém může stavět při nadcházejícím studiu nejen z teoretické perspektivy, ale i praktické.

1.3 Struktura práce a prostředí

Práce se skládá ze dvou základních komponent a to:

- Teoretická
- Praktická

Teoretická část je hlavní text, ve kterém bude vysvětlena teorie a bude čtenáře provázet. Nicméně pro přehlednost a interakci čtenáře budou teoretické části sekundovat **části praktické**, které budou v podobě Jupyter notebooků (viz níže 1.3.4), díky kterým si může čtenář osahat kód vlastníma rukama. Níže je popis různých technologií, se kterými se čtenář v průběhu četby práce setká:

1.3.1 Python 3

Jazyk Python ve verzi 3 (dále pouze Python) byl zvolen nejen pro jeho jednoduchou syntaxi, ale především proto, že velká část zdrojů jej již využívá a je tedy poměrně snadné najít potřebné knihovny, ukázkové kódy a pokračovat ve vlastním studiu. Informace potřebné k instalaci Pythonu lze najít na oficiálních stránkách[1].

1.3.2 Pickle

Pro serializaci dat je využita knihovna pickle[2]. Díky tomu máme naše transformovaná data uložena v souboru a při příštím použití nám pouze stačí je načíst, jelikož jsou již v námi požadovaném formátu.

1.3.3 NumPy

NumPy je matematická knihovna pro jazyk Python, která poskytuje podporu při výpočtech. Nemá nic společného s logikou tvořené neuronové sítě, nýbrž je to pomocný nástroj, který ulehčí práci například s vícerozměrnými poli a vektorovými výpočty a podobně. Pro více informací může čtenář navštívit oficiální stránky[3].

Více viz [1]

1.3.4 Jupyter Notebook

Jupyter Notebook je používán k prezentování praktických částí této práce. Je využíván v podstatě jako interaktivní učebnice, kde je samozřejmostí nejen zvýrazňování syntaxe psaného kódu, ale kód lze libovolně editovat, spustit a uživatel uvidí výstup programu po jeho úpravách. Krom psaní samotného kódu jde do notebooku psát i klasický text. Díky tomu se z něj stává kvalitní nástroj pro vysvětlování problematiky psaného programu.

Pokud by čtenář neměl zájem používat Jupyter notebooky, pak samozřejmě **nemusí**. Všechny programy, které budou napsány v rámci této práce, budou samozřejmě přiloženy jako klasické přílohy, které lze spustit a upravovat. Zdrojem pro více informací o projektu Jupyter jsou oficiální stránky[4].

Více viz [2]

[1] `https://www.python.org`
[2] `https://docs.python.org/3/library/pickle.html`
[3] `https://numpy.org/`
[4] `https://jupyter.org/`

1.4 Neuronová síť

[3] *Počítačová aplikace nebo systém využívající k řešení úloh model funkcí biologického neuronu (tzv. procesor, výkonný prvek, perceptron). Typický procesor má více vstupů, které dokáže klasifikovat a na jejich základě generovat výstup. Procesory jsou navzájem propojeny do sítí ohodnocenými vazbami, což umožňuje nealgoritmické a paralelní zpracování složitých úloh. Sítě mohou mít různou topologii (asociativní, rekurentní, vrstvená). Činnost sítě je založena na procesu učení, tj. adaptace na konkrétní úlohu za pomoci vnějšího činitele (síť s učitelem) nebo na základě stimulů (samoorganizující se síť).*

Tolik k formální definici. Neformálně můžeme neuronové sítě chápat jako model, který lze využít k řešení problémů. Nejde tedy o konkrétní implementaci[1], nýbrž o přístup k řešení. Na rozdíl od tradičního přístupu k programování, kde jsou jasně definována pravidla, jak problém řešit. Neuronové sítě si naopak samy z dat najdou a naučí důležité vlastnosti, které jsou nutné k vyřešení daného problému.

1.5 Machine learning

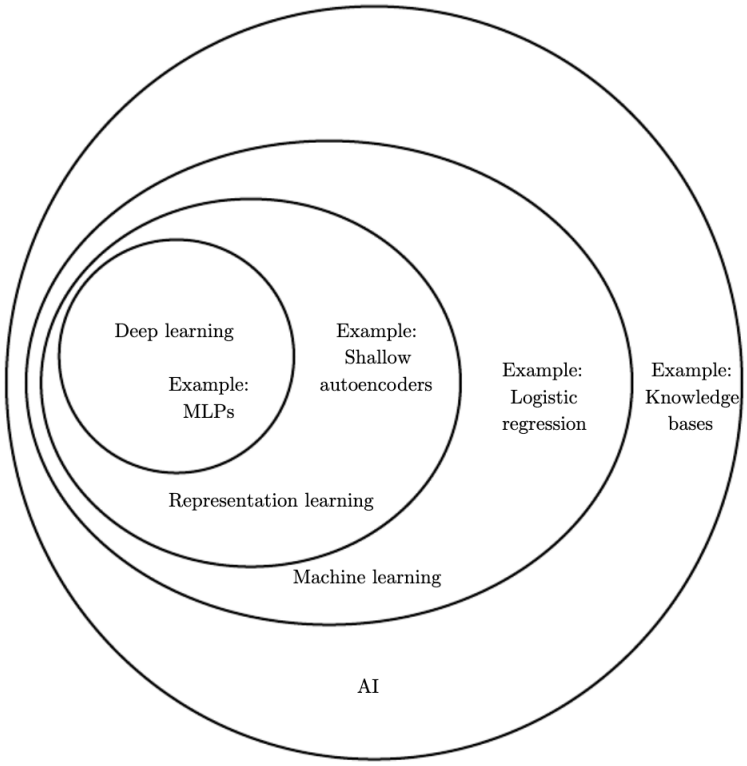
[4] *Strojové učení je podskupina umělé inteligence, která se zaměřuje na vývojové systémy, které se učí – nebo zvyšují svou výkonnost – na základě dat, se kterými pracují. Umělá inteligence je široký pojem, který označuje systémy nebo stroje, které napodobují lidskou inteligenci. Strojové učení a umělá inteligence jsou často diskutovány společně a tyto pojmy jsou někdy používány zaměnitelně, ale neznamenají totéž. Důležitým rozdílem je to, že ačkoli veškeré strojové učení je umělou inteligencí, není veškerá umělá inteligence strojovým učením.*

1.6 Artificial intelligence

[5] *Mezioborová vědní disciplína na pomezí kognitivních věd, kybernetiky a počítačové vědy, která zkoumá a modeluje inteligenci s cílem vyvinout software a hardware, který bude při řešení úloh používat postupy považované za projev lidské inteligence.*

Na obrázku níže lze vidět, že je mnoho přístupů k AI a ML je jen jedním z nich a je tedy podmnožinou AI.

[1] `https://slovnik-cizich-slov.abz.cz/web.php/slovo/implementace`



Obrázek 1.2. Vennův diagram demonstrující jednotlivé přístupy k AI a jak spolu souvisí [6]

Kapitola 2
Definice problému

V této práci budeme demonstrovat implementaci konkrétního řešení neuronové sítě na „hello world“[1] neuronových sítí a to rozpoznávání ručně psaných číslic. Tento úkol byl zvolen nejen kvůli nižší komplexitě, ale zároveň díky výbornému, již existujícímu, datasetu MNIST (viz 2.1), který využijeme, abychom nemuseli sami sbírat data, což je mimo rozsah této práce.

Dále díky excelentní knize Michaela Nielsena, která je výborným zdrojem a z implementace sítě v této knize vychází kód této práce. Bližší popis MNIST datasetu je k nalezení v kapitole 5 a práci Michaela Nielsena lze najít zde [7].

2.1 MNIST

MNIST dataset je databáze ručně psaných číslic čítající celkem 70 000 popsaných příkladů. Popsaným příkladem je myšlen jeden konkrétní obrázek číslice, který u sebe obsahuje i korektní hodnotu číslice, díky které víme, o kterou se jedná a jsme schopni posléze určit, zda-li byl obrázek správně klasifikován. [8]

2.1.1 Soubory

Samotný dataset je tvořen následujícími soubory a je již rozdělen na tréninkovou a testovací sadu:

- **train-images-idx3-ubyte.gz**: Tréninková množina obrázků
- **train-labels-idx1-ubyte.gz**: Tréninková množina popisků
- **t10k-images-idx3-ubyte.gz**: Testovací množina obrázků
- **t10k-labels-idx1-ubyte.gz**: Testovací množina popisků

Další vlastnost MNIST datasetu, kterou s výhodou využijeme je, že byl již normalizován a číslice byly vycentrovány, což nám značně ulehčí práci s preprocesingem (viz 2.1.2).

Dataset je ke stažení na stránkách `http://yann.lecun.com/exdb/mnist`.

2.1.2 Preprocesing

Preprocesing je krok, ve kterém jsou data transformována do takového stavu, aby byla snadno strojově zpracovatelná [9].

[1] `https://www.thesoftwareguild.com/blog/the-history-of-hello-world/`

Kapitola 3
Notace

Jelikož se v rámci této práce setkáme s mnoha matematickými zápisy, je vhodné mít vše na jednom místě, aby se měl čtenář kam vrátit v případě, že by tápal a nebyl si jist významem nějakého zápisu. **Není nutné tuto kapitolu procházet ihned, jelikož se čtenář jistě těší, až se dozví více o neuronových sítích. Je tedy možné tuto kapitolu přeskočit a vracet se dle potřeby.**

Jak již název napovídá, tak neuronové sítě se skládají z jednotlivých neuronů, které jsou uspořádány do vrstev. Tím se ovšem nekončí, jelikož každý neuron se skládá z dalších komponent, které si v této kapitole představíme spolu s jejich notací, kterou budeme v průběhu práce využívat.

3.1 Vrstvy

Jak jsme zmínili výše, tak jednotlivé neurony jsou uspořádány do vrstev spolu s jejich vahami (viz 3.7) a biasy (viz 3.6). Abychom byli schopni rozlišovat, o které vrstvě se bavíme, budeme využívat exponentu, kde do závorky napíšeme číslo, o kterou vrstvu se jedná. Například:

$$\mathbf{b}^{(1)}$$

Toto by byl vektor biasů pro první vrstvu. Pokud se budeme ovšem bavit o obecných vzorcích, pak místo konkrétního čísla budeme vrstvu značit jako (l).

Jako poslední se v souvislosti s vrstvami budeme setkávat s velkým (L). To již není obecný zápis, jelikož je to konstanta a je tím myšlena poslední vrstva. Pokud bychom tedy měli síť o pěti vrstvách, pak $L = 5$.

3.1.1 Vstupní vrstva

Vstupní vrstva (*anglicky input layer*) je první vrstva nalevo v síti a není ve skutečnosti tvořena neurony, i přes to, že dle nákresů to tak může vypadat. Vstupní vrstva jsou ve skutečnosti data, která do sítě dodáváme, se kterými již není v rámci dané vrstvy nikterak dále manipulováno.

3.1.2 Skryté vrstvy

Skryté vrstvy (*anglicky hidden layers*), ač se mohou dle názvu zdát misteriózní, nejsou nic jiného, než jakákoliv vrstva nacházející se mezi vstupní a výstupní vrstvou.

3.1.3 Výstupní vrstva

Výstpní vrstva (*anglicky output layer*) je poslední vrstva naší sítě (tedy první zprava) a výstupní aktivace jejích neuronů jsou výstupem naší sítě. Jednotlivé číselné hodnoty pak interpretujme jako pravděpodobnosti. Ten neuron, který má nejvyšší aktivaci = nejvyšší pravděpodobnost, je brán jako volba sítě.

3.2 Vektory

Toto opět navazuje na fakt, že síť je uspořádána do vrstev. Abychom nemuseli počítat s jednotlivými čísly a vypisovat jejich konkrétní indexy, pak často sáhneme po vektorové/maticové notaci. Notaci pro jednotlivá čísla se budeme věnovat níže, nicméně aby se nám na první pohled lépe orientovalo, kdy se bavíme o vektoru a kdy ne, tak vektory budeme značit tučným fontem. Toto by byl vektor biasů třetí vrstvy:

$$\mathbf{b}^{(3)}$$

3.3 Vážený vstup

Vážený vstup, který značíme jako $z_j^{(l)}$, je potřebný k výpočtu samotné aktivace neuronu. O aktivacích si řekneme v zápětí. Vypočíst jej lze následujícím způsobem:

$$z_j^{(l)} = \sum_j w_j^{(l)} a_j^{(l-1)} + b^{(l)}$$

Teorie 3.1. $\sum$ viz 4.1

Pokud přejdeme na vektorovou formu, pak se bavíme o vektoru vážených vstupů pro danou vrstvu. Jeho výpočet a zápis $l-tou$ vrstvu by vypadal následovně:

$$\mathbf{z}^{(l)} = \mathbf{w}^{(l)} \cdot \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}$$

Teorie 3.2. Skalární součin viz 4.3

3.4 Aktivace

Aktivací je myšlen výstup z konkrétního vektoru po aplikaci sigmoid funkce na vážený vstup. Aktivaci značíme pomocí malého a. Pokud bychom se bavili zápisu jedné konkrétní aktivace, pak budeme využívat následující zápis, kterým je myšlena $j-tá$ aktivace v $l-té$ vrstvě:

$$a_j^{(l)}$$

Většinou budeme sahat po vektorové formě. Pak mluvíme o vektoru aktivací pro danou vrstvu. Vektor aktivací například pro $l-tou$ vrstvu by tedy vypadal takto:

$$\mathbf{a}^{(l)}$$

Jako poslední informace k aktivacím stojí za zmínku dva vektory, které jsou specifické. A to $\mathbf{a}^{(0)}$ a $\mathbf{a}^{(L)}$. $\mathbf{a}^{(0)}$ je specifický v tom, že je to ve skutečnosti náš vstup do sítě - tedy data, která síti dodáme. Naopak $\mathbf{a}^{(L)}$ je výstup ze sítě - pro nás to tedy je odhad sítě o kterou číslici se jedná.

3.5 Sigmoid funkce

Sigmoid funkci značíme pomocí σ a má následující předpis:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Poznámka 3.1. Pokud je vstup do funkce vektor, tedy $\mathbf{z}$, pak je operace provedena na každý element zvlášť.

Příklad 3.1. Aplikujme sigmoid funkci na vektor $\mathbf{z}$:

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_n \end{bmatrix}$$

$$\sigma(\mathbf{z}) = \begin{bmatrix} \sigma(z_1) \\ \sigma(z_2) \\ \vdots \\ \sigma(z_n) \end{bmatrix}$$

3.6 Biasy

Každý vektor má svůj bias. Pokud se budeme chtít bavit o konkrétních biasech, pak sáhneme po následující notaci:

$$b_j^{(l)}$$

Tímto je myšlen $j-tý$ bias v $l-té$ vrstvě. Opět s výhodou budeme využívat vektorovou formu zápisu. Pak se bavíme o vektoru biasů pro danou vrstvu. Ukažme si příklad pro vektor biasů $l-té$ vrstvy:

$$\mathbf{b}^{(l)}$$

3.7 Váhy

Každý neuron má zároveň váhy, které jej spojují se všemi neurony předchozí vrstvy. Z toho vyplývá, že každý neuron má více, než jednu váhu, a proto zde sáhneme po maticovém zápisu. Nicméně netřeba se znepokojovat. Nejprve si ukážeme, jak budeme značit, pokud se bavíme o jedné konkrétní váze:

$$w_{jk}^{(l)}$$

Takto by byla označena váha, která spojuje $j-tý$ neuron v $l-té$ vrstvě a $k-tý$ neuron v $(l-1)$ vrstvě. Toto značení se může zdát naopak, než by bylo přirozené. Nicméně jako pomůcka může posloužit, že (l) a j jsou nad sebou v jednom sloupečku a tedy jde o stejnou vrstvu. Naopak k je mimo a tedy je to vrstva $(l-1)$.

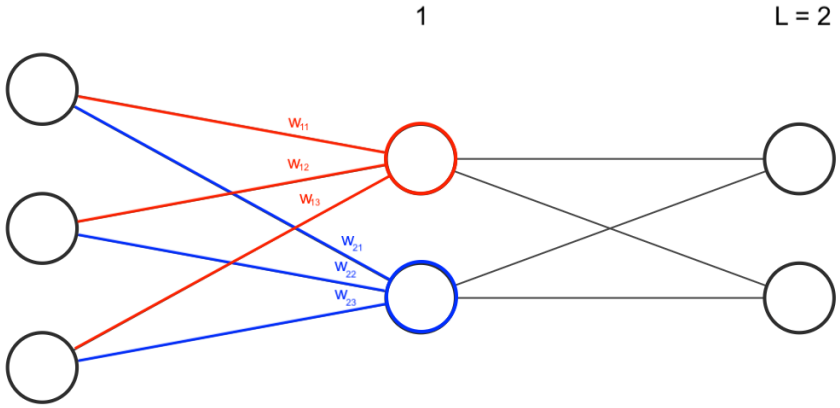
Abychom mohli zapisovat váhy kompaktněji, sáhneme po maticích. Matice vah pro vrstvu (l) bude tedy vypadat následovně:

$$\mathbf{w}^{(l)}$$

Abychom se podívali na konkrétní příklad a ujasnili si indexování, pojďme se bavit o síti na obrázku 3.1. Zde si obecně vypíšeme obsah matice pro 1. vrstvu. Tedy:

$$\mathbf{w}^{(1)} = \begin{bmatrix} w_{11}^{(1)} & w_{12}^{(1)} & w_{13}^{(1)} \\ w_{21}^{(1)} & w_{22}^{(1)} & w_{23}^{(1)} \end{bmatrix} \tag{1}$$

Lidsky řečeno můžeme každý řádek matice vnímat jako veškerá propojení neuronu s předchozí vrstvou. Tedy první řádek obsahuje váhy pro první neuron $l-t\acute{e}$ vrstvy (červeně). Druhý řádek obsahuje váhy pro druhý neuron (modře) atd... Matice má vždy tolik řádků, kolik má daná vrstva neuronů a tolik sloupců, kolik neuronů má vrstva předchozí.



Obrázek 3.1. Vizuální ukázka co ve skutečnosti reprezentuje matice vah pro vrstvu (l) v tuto chvíli tedy $\mathbf{w}^{(1)}$

Příklad 3.2. Nechť si čtenář zkusí sám odvodit rozměr a následně sepsat matici vah pro L, tedy $\mathbf{w}^{(L)}$ stejně jako jsme to udělali pro $l = 1$ zde (1).

Kapitola 4
Matematika

Ačkoli by bylo báječné vysvětlit tématiku neuronových sítí čistě lidsky bez matematiky, ve skutečnosti to bohužel možné není. Neuronové sítě jsou plné matematiky, nicméně není třeba se jí bát. Naopak, jakmile je pochopena může působit až přirozeně.

V této kapitole se může čtenář seznámit s různými matematickými aparáty, které budou v rámci této práce využívány. Nicméně je potřeba zmínit následující upozornění:

Jakékoliv vysvětlivky matematických metod v této práci nejsou exaktní. Naopak přístup, kterým jsou vysvětlovány je volen úmyslně tak, aby byl pochopitelný lidským jazykem a bylo pokud možno srozumitelné co daný aparát dělá a k čemu nám slouží, jelikož to je podstata této práce.

Nebudou tedy poskytovány důkazy, nicméně vždy bude odkázáno na další literaturu, jejíž podstatou naopak je tyto aparáty exaktně vysvětlit a dokázat.

Nyní, když máme upozornění za sebou, si uvedeme naopak doporučení. **Tuto kapitolu je doporučeno přeskočit, pokračovat rovnou kapitolou 5 a vracet se postupně v případě potřeby. Pokud to bude nutné, pak na tuto kapitolu bude odkázáno a čtenář bude mít kde čerpat.** Proč tomu tak je? Cílem této práce je přiblížit tuto tématiku především studentům středních škol, které jistě zajímají neuronové sítě a nikoli matematický aparát skrývající se v pozadí. Než by se student kapitolou prokousal, mohl by také ztratit motivaci - což by byla škoda. Naopak, pokud již bude znát důvod, kvůli kterému se daný aparát učí, bude mít jistě větší motivaci pokusit se jej pochopit.

4.1 Suma

Již z názvu lze odvodit, že pomocí sumy můžeme sečíst prvky či funkce, které jsou za sumou uvedeny. Sumu značíme pomocí symbolu $\sum$. Pojďme si demonstrovat užití sumy na příkladu.

Příklad 4.1. Mějme vektor $\mathbf{a} = (1, 2, 3)$. Vypočtěme sumu jeho členů:

$$\sum_{j=1}^{3} a_j = (1 + 2 + 3) = 6$$

V některých případech se vynechávají číselné indexy. Je tím myšleno zkrátka přes všechny prvky. Tedy:

$$\sum_{j} a_j = (1 + 2 + 3) = 6$$

A ještě jeden příklad:

$$\sum_{j=1}^{3} a_j^2 + a_j = (1^2 + 1) + (2^2 + 2) + (3^2 + 3) = 20$$

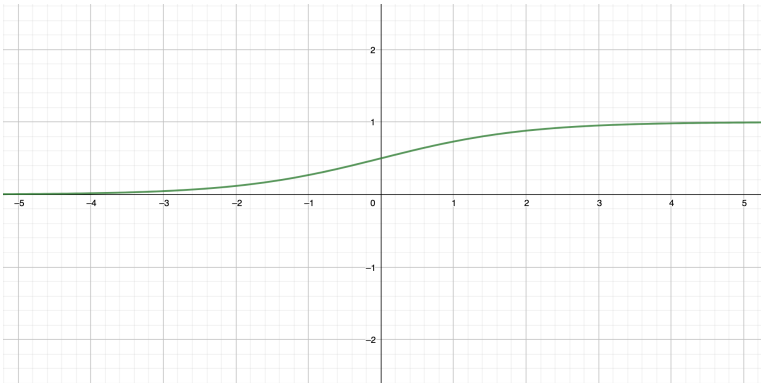
Dále může čtenář čerpat například zde [10].

4.2 Sigmoid funkce

Sigmoid funkce, nebo také logistická funkce, či logistický sigmoid je v našem kontextu využívána pro výpočet aktivace neuronu na základě váženého vstupu. Ač jsme si předpis sigmoid funkce ukázali již v 3.5, tak si jej připomeneme zde společně s vysvětlivkou, co pro nás sigmoid funkce dělá.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Pokud bychom si sigmoid funkci vyobrazili na grafu, pak dostaneme graf následující:



Obrázek 4.1. Graf sigmoid funkce

Z grafu můžeme vidět, že funkce σ nám vlastně převede celé $\mathbb{R}$ do intervalu $(0, 1)$.

Více viz [11] str. 65

4.3 Skalární součin

Se znalostí sumy, jsme schopni definovat skalární součin:

Definice 4.1. Mějme vektory $\mathbf{x}$ a $\mathbf{y}$ o n prvcích, pak jejich skalární součin je dán jako:

$$\mathbf{x} \cdot \mathbf{y} = \sum_{j} x_j y_j = x_1 y_1 + x_2 y_2 + \ldots + x_n y_n$$

Poznámka 4.1. Berme na vědomí, že „np.dot“ se v knihovně numpy používá i k maticovému násobení.

Více viz [12] str. 124

4.4 Násobení matic

U násobení matic si pouze zmíníme podmínku pro násobení a jakým způsobem lze zjistit rozměry výsledné matice. Mějme matici A a B, kde matice A má rozměry $m \times n$ a matice B má rozměry $k \times j$. Kde první rozměr vždy značí počet řádků a druhý počet sloupců:

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}; B = \begin{bmatrix} b_{1,1} & b_{1,2} & \cdots & b_{1,j} \\ b_{2,1} & b_{2,2} & \cdots & b_{2,j} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k,1} & b_{k,2} & \cdots & b_{k,j} \end{bmatrix}$$

Pak tyto dvě matice lze násobit pouze, pokud $n = k$ a výsledný rozměr matice bude $m \times j$.

Příklad 4.2. Mějme následující matice A a B:

$$A = \begin{bmatrix} a & b \\ c & d \\ e & f \end{bmatrix}; B = \begin{bmatrix} g & h & i \\ j & k & l \end{bmatrix}$$

Vidíme, že jejich rozměry jsou 3×2 a 2×3 respektive. Dále lze vidět, že vnitřní členy se rovnají - matice tedy lze násobit a vnější členy nám dají výsledný rozměr vynásobené matice - tedy 3×3.

Více viz [12] str. 49

4.5 Transpozice

Transpozicí, neboli transponováním je myšlen proces, kdy se v matici prohodí řádky za sloupce a naopak. Tedy:

$$[a_1, a_2, \ldots, a_n]^T = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

Tato operace se dá jednoduše pospat krátkým vzorcem, kde i je číslo řádku v matici a j číslo sloupce. a_{ij} je potom prvek na $i-tém$ řádku a v $j-tém$ sloupci.

$$a_{ij}^T = a_{ji}$$

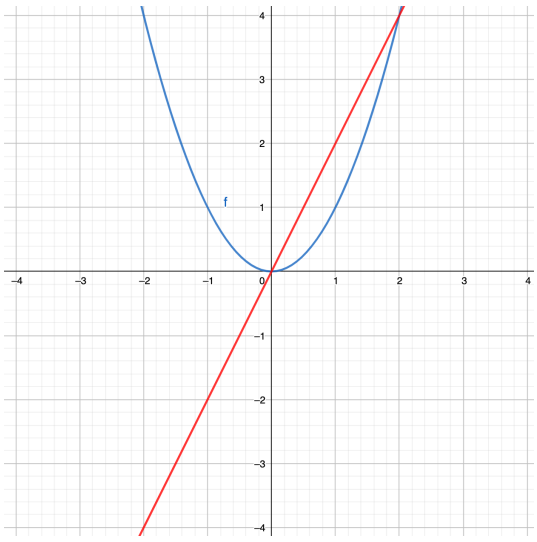
$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix}^T = \begin{bmatrix} a_{1,1} & a_{2,1} & \cdots & a_{m,1} \\ a_{1,2} & a_{2,2} & \cdots & a_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1,n} & a_{2,n} & \cdots & a_{n,m} \end{bmatrix}$$

Více viz [12] str. 48

4.6 Derivace funkce

Poznámka 4.2. Bohužel je mimo rozsah této práce kompletně vysvětlit tak komplexní téma, jako jsou právě derivace. Od toho jsou matematická skripta. Nicméně si pokusíme derivaci vysvětlit intuitivně a nebudeme se starat o samotný proces, jak derivovat. V případě potřeby, nechť se čtenář obrátí na potřebná skripta. Ač znalost derivací pomůže, pro pochopení problematiky neuronových sítí stačí chápat podstatu, kterou si pokusíme vysvětlit.

Co to tedy vlastně ta derivace je? Buďme konkrétní a řekněme, že máme funkci $f(x) = x^2$. Derivací funkce $f(x)$ je funkce $f'(x) = 2x$. A pokud si obě funkce vyobrazíme na grafu, pak získáme takový jako na obrázku 4.2



Obrázek 4.2. Graf funkce $f(x) = x^2$ (modře) a její derivace $f'(x) = 2x$ (červeně)

Ti čtenáři, kteří za sebou mají základy diferenciálního počtu jistě ví, že derivace funkce nám říká něco o průběhu funkce původní. Zkusme se podívat na obrázek 4.2 a vypozorovat vztah mezi funkcí $f(x) = x^2$ a její derivací $f'(x) = 2x$.

Z grafu je patrné, že původní funkce (modrá) klesá v částech, kde je její derivace (červená) záporná. V bodě 0, kde je její minimum (tedy neroste, ani neklesá) je i její derivací nulová a ve zbylé části, kde původní funkce roste je derivace kladná.

Příklad 4.3. Mějme výše zadanou funkci a ověřme, že na intervalu $(-\infty, 0)$ klesá, v bodě 0 je konstantní a na intervalu $(0, \infty)$ roste.

Zvolíme si tři body v každém z požadovaných intervalů:

$$a = -2; a \in (-\infty, 0)$$

$$b = 0;$$

$$c = 2; c \in (0, \infty)$$

A nyní dosadíme do derivace funkce a ověříme s grafem:

$$f'(x) = 2x$$

$$f'(-2) = -4$$

$$f'(0) = 0$$

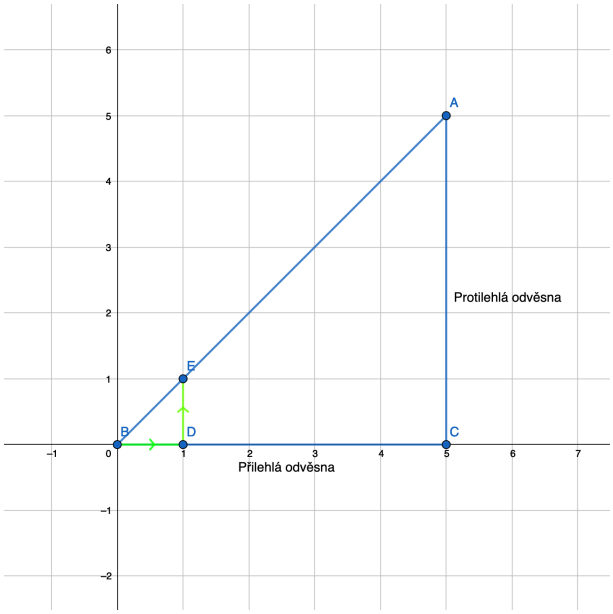
$$f'(2) = 4$$

Čtenář si nyní může ověřit, že funkce v bodech a, b, c skutečně klesá, je konstantní a roste respektive. Může ovšem vyvstat otázka, proč se tomu tak děje? Pro exaktní definici je opět nejlepší nahlédnout do patřičných skript. My si ovšem pokusíme popsat intuitivní představu.

Jistě je čtenář obeznámen s goniometrickými funkcemi a konkrétně pak s funkcí $\tan(x)$ jež platí v pravoúhlém trojúhelníku a vypočítá se jako poměr odvěsen. Konkrétně:

$$\frac{\textit{Délka protilehlé odvěsny}}{\textit{Délka přilehlé odvěsny}}$$

Dále víme, že pomocí této funkce lze vypočítat protilehlý úhel. Ale to pro nás není tak důležité jako samotný fakt, že je tan definován jako poměr odvěsen trojúhelníku. Co nám toto vlastně říká? Jde si to vyložit následujícím způsobem: Pokud popojdu o jeden dílek na ose x, pak na ose y popojdu o $tan(\alpha)$. Pojďme být konkrétní a podívejme se na obrázek 4.3.



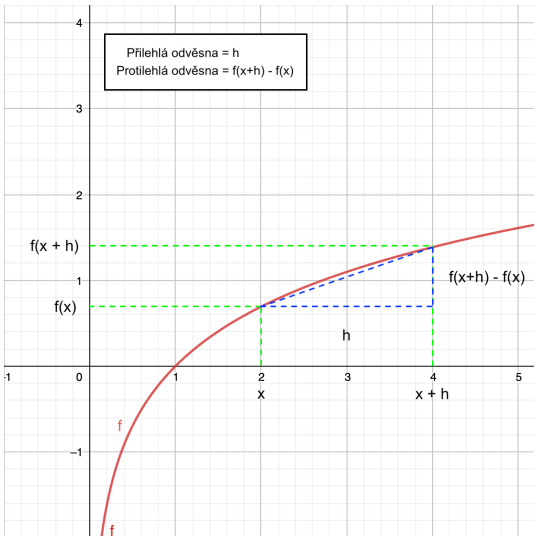
Obrázek 4.3. Pravoúhlý trojúhelník

Je zřejmé, že nás bude zajímat úhel ABC a pojmenujeme si jej α. Dále z obrázku vidíme, že $\tan(\alpha) = \frac{5}{5} = 1$. Přeformulujeme si to dle výše zmíněné pomůcky: Pokud se posuneme o jeden dílek na ose x, pak se posuneme o $\tan(\alpha)$ na ose y. A jelikož jsme vypočítali, že $\tan(\alpha) = 1$, pak můžeme skutečně z obrázku ověřit, že tomu tak je.

Poznámka 4.3. Spokojíme se tedy s intuitivní představou, že $\tan(\alpha)$ nám říká, jak moc se daná funkce zvětší, či zmenší, pokud se posuneme o jeden dílek. Samozřejmě není nutné se pohybovat přesně o jeden dílek, ale lze jej jakkoli měnit pokud ovšem změníme stejným způsobem i výsledný $\tan(\alpha)$.

Proč tato odbočka k funkci tan? Protože jsme si již říkali, že derivace nám říká, jak moc funkce roste či klesá a ukázali jsme si na příkladu 4.3, jak zjistit derivaci v daném bodě. Pojďme si nyní společně odvodit oficiální definici derivace, kterou najdete i ve skriptech.

Příklad 4.4. Odvození definice derivace. Mějme nějakou funkci $f(x)$ a zvolíme si konstantu h. Mějme následující obrázek: 4.4



Obrázek 4.4. Intuitivní popis derivace funkce

Zvolíme si libovolné x v definičním oboru funkce a dále h, které může být libovolně z $\mathbb{R}$. Z těchto údajů již známe, či jsme schopni vypočítat následující hodnoty: x, $x+h$, $f(x)$, $f(x+h)$. Pokud se podíváme na obrázek 4.4, pak si můžeme povšimnout, že nám tyto hodnoty společně s funkcí f tvoří v podstatě pravoúhlý trojúhelník. Samozřejmě, že přepona není přesná. Ale víceméně máme trojúhelník. Dále si můžeme povšimnout, že jsme získali obě odvěsny a jejich délku můžeme vypočítat takto:

$$přilehlá\ odvěsna = (x+h) - x = x + h - x = h$$

$$protilehlá\ odvěsna = f(x+h) - f(x)$$

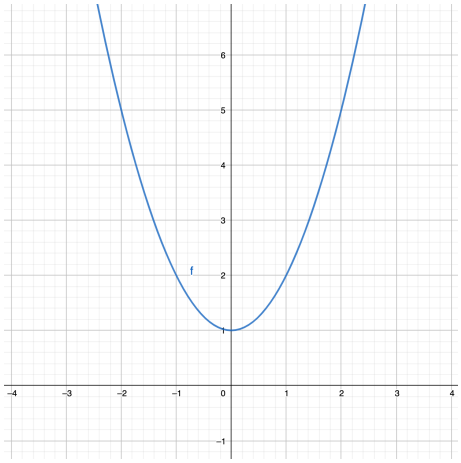
A konečně tedy $\tan(\alpha)$ můžeme tedy zapsat jako:

$$\tan(\alpha) = \frac{f(x+h) - f(x)}{h}$$

Pokud něco není jasné, nechť čtenář zkonzultuje obrázek 4.4, kde jsou odvěsny vyznačeny. Nyní nám už zbývá poslední problém - přepona není úsečkou a může mít tedy libovolný tvar. Z definice můžeme ovšem vyvodit, že pokud zvolíme h menší, pak se přiblížíme k původnímu bodu a odhad bude přesnější. Tedy by se nám hodilo mít h nejmenší co to jde. Naštěstí přesně takový aparát máme k dispozici a tím je limita. Pokud bychom tedy zmenšovali h tak dlouho, až by bylo skoro nulové, pak bychom získali derivaci v bodě x. Pokud tedy doplníme definici, pak získáme:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Více viz [13] str. 63



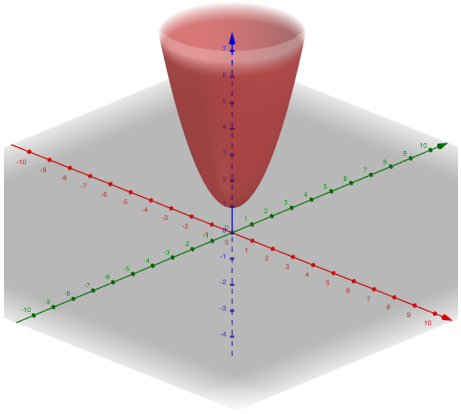
Obrázek 4.5. Graf funkce $f(x) = x^2 + 1$

4.7 Funkce více proměnných

Na středních školách se většinou učí funkce o jedné proměnné. Mějme tedy funkci jedné proměnné například $f(x) = x^2 + 1$ a její graf vypadá takto viz obrázek 4.5

Jak jistě čtenář ví, předpis $f(x)$ značí, že do funkce dosadíme za x libovolné číslo, které je pro danou funkci v definičním oboru a funkce nám na oplátku vrátí výslednou hodnotu. V tomto případě, pokud $x = 2$, pak $f(2) = 2^2 + 1 = 5$.

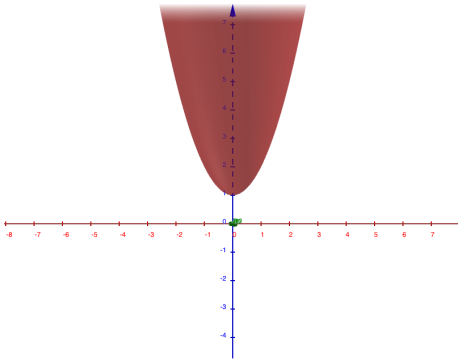
U funkce více proměnných je postup obdobný. Vezměme si opět nějakou jednoduchou funkci například $f(x, y) = x^2 + y^2 + 1$ a opět se podívejme na její graf na obrázku 4.6



Obrázek 4.6. Graf funkce $f(x, y) = x^2 + y^2 + 1$

Jak můžeme vidět, tentokrát po nás funkce nepožaduje pouze jedno číslo, tak jako doposud, ale je potřeba dosadit čísla dvě. Pokud si dosadíme například takto: $x = 2, y = 3$, pak $f(2, 3) = 2^2 + 3^2 + 1 = 4 + 9 + 1 = 14$. Pro čtenáře, který se s takovýmto grafem setkává poprvé, nemusí být vše hned jasné. Nicméně je to stejné jako s funkcí jedné proměnné.

Na obrázku vidíme osu x vyznačenou červenou barvou. Pokud bychom se na tento graf podívali tak, že bychom viděli pouze osu x, pak dostaneme následující graf viz obrázek 4.7



Obrázek 4.7. Pohled na osu X grafu funkce $f(x, y) = x^2 + y^2 + 1$

Pokud porovnáme s grafem 4.5, pak můžeme vidět, že se jedná o totožný tvar. K funkci o více proměnných se budeme, alespoň v rámci této práce, chovat v podstatě stejně, jako k funkci jedné proměnné s tím rozdílem, že krom dosazení za jednu proměnnou (v grafu si ji představme jako osu) dosadíme za dvě či více proměnných.

V našem případě funkce o dvou proměnných tedy dosadíme prvně za $x = 2$ (po ose x se posuneme o dva dílky), poté za $y = 3$ (po ose y se posuneme o tři dílky). Poté bychom se již mohli podívat na graf a odečíst, že na ose $z = f(x, y)$ je výsledné číslo 14, což lze ověřit i výpočtem.

Stejným způsobem fungují i funkce o n proměnných. U nich si bohužel graf již představit nelze. Existují metody, jak si pomoci se zobrazením grafu pro funkci o třech neznámých. Tam nám pomáhá například barva, nicméně pro funkce o 4 a více proměnných, tam je již potřeba si vypěstovat určitou intuici.

Poznámka 4.4. I přes to, že si tyto funkce již nelze představit vizuálně, jde vypěstovat určitá intuice. Jedna z možností je nepřemýšlet příliš nad vlastním tvarem funkce, ale brát v potaz, že vše funguje obdobně jako v našem trojrozměrném světě. Pokud bychom měli nějakou funkci $f(a, b, c, d)$, kde $a = 1, b = 2, c = 3, d = 4$, pak si představíme, že bychom se posunuli například o 1m ve směru a (rovně), o 2m ve směru b (doprava) o 3m ve směru c (nahoru) a o 4m ve směru d. U posledního rozměru jsme již v našem trojrozměrném světě nahraní, nicméně si to představme analogicky a posunuli bychom se zkrátka o již zmíněné 4m ve směru d.

Více viz [14] str. 17

4.8 Parciální derivace

Teorie 4.1. Derivace viz 4.6

Nyní již víme, jak chápat derivaci. A to tak, že nám říká jakým způsobem se daná funkce chová v libovolném bodě jejího definičního oboru. Nicméně již víme, že existují i funkce o mnohem více proměnných, u kterých bychom také potřebovali zkoumat jejich derivace. A abychom toho mohli docílit, tak si musíme představit parciální derivace.

Jelikož u funkcí jedné proměnné je proměnná pouze jedna, pak není nutné se zaobírat tím, ve kterém směru funkci zkoumáme. A jak jsme si již dříve řekli, jednotlivé vstupní

proměnné lze brát každou jako určitý směr. Pokud chceme tedy derivovat funkci o více proměnných, je potřeba si určit, ve kterém směru ji zkoumáme.

To by se slovně dalo popsat jako „Vůči které proměnné derivujeme“. Nyní si pro derivaci představíme alternativní značení, které je ovšem rovnocenné s tím pro jednu proměnnou. Abychom si to demonstrovali, tak ještě na chvilku u funkce jedné proměnné zůstaneme.

Příklad 4.5. Mějme $f(x) = 3x^2 + 2$, pak:

$$f'(x) = \frac{\partial f}{\partial x} \tag{1}$$

Prosím čtenáře, aby se nezalekl značení a kdykoliv uvidí symbol ∂, aby jej zkrátka vnímal jako signalizaci, že jde o derivaci. Nyní převedeme na slova pravou stranu vzorce (1). „Derivace funkce f vzhledem k proměnné x“. Jak je vidět, skutečně se nejedná o nic složitého, ale pouze o jiný způsob zápisu.

Nyní se již pojďme podívat na derivace parciální. Důvod, proč lze derivaci funkce jedné proměnné zapsat takto $f'(x)$ je ten, že máme pouze jednu proměnnou a tak je jasné dle které chceme derivovat.

Mějme ovšem funkci, například $f'(x, y) = 2x + 2y$. Představme si nyní, že dostaneme příklad, kde bude následující zadání: $f'(x, y)$. Máme derivovat podle x? Či podle y? Jak je vidět, zápis není jednoznačný, a proto se u parciálních derivací sahá po druhé formě zápisu.

Mějme nyní stejný příklad, avšak zadaný tímto způsobem: $\frac{\partial f}{\partial y}$. Nyní jasně vidíme, že chceme derivovat vzhledem k proměnné y a směle se pustíme do práce.

Poznámka 4.5. Jak je patrné z vzorce (1) - oba zápisy jsou ekvivalentní. Tedy pokud by čtenář rád zapisoval derivace funkce jedné proměnné způsobem na pravé straně od rovnítka, může to zcela jistě udělat.

A nyní hlavní otázka: **Jak parciálně derivovat**? Čtenář, který se s parciálními derivaci doposud nesetkal jistě očekává zradu, avšak naštěstí bude potěšen, neboť parciální derivace nejsou o nic složitější, než derivace klasické. Zopakujeme si jedno z pravidel pro počítání derivací:

$$c' = 0; c = libovolná\ konstanta \tag{2}$$

Jak je z tohoto pravidla vidět, derivace konstanty je **vždy** nulová a to je pro nás dobrá zpráva! Mějme například následující složitě vyhlížející funkci o čtyřech neznámých a vypočteme si její derivaci vůči proměnné z:

Příklad 4.6. Nechť $f(x, y, z, w) = 3x^2 - 12y^7 + z - w^2$, kde $x, y, z, w \in \mathbb{R}$, pak:

$$\frac{\partial f}{\partial z} = 1$$

Jelikož derivujeme dle proměnné z, tak můžeme všechny ostatní proměnné vnímat jako konstanty. Jako pomůcku lze využít, že si místo proměnných, dle kterých nederivujeme napíšeme libovolné konstantní číslo. **Ovšem pozor. Nezapomínejme na pravidla derivování a pokud je nějaká jiná proměnná v součinu s tou, dle které derivujeme pak nám nevypadne jako v tomto případě.** Ale neděsme se, pravidla pro derivování jsou skutečně všude stejná, nicméně příklad 4.6 byl účelně zvolen takový, aby derivace vyšla hezky, avšak vždy to tak být nemusí.

Příklad 4.7. Se znalostmi ze sekce 4.6 nechť čtenář zkusí sám ověřit, proč platí pravidlo (2).

Více viz [14] str. 53

4.9 Gradient

Teorie 4.2. Parciální derivace viz 4.8

S parciálními derivacemi souvisí ještě jeden, nejen pro nás, důležitý pojem a tím je **gradient funkce**. Gradient funkce je **vektor parciálních derivací** a značí se následovně ∇. Pokud se čtenář s tímto pojmem setkává poprvé, může opět znít složitě, ale není se čeho obávat. Vše si opět názorně ukážeme na jednoduchém příkladu:

Příklad 4.8. Mějme funkci více proměnných $f(x, y) = x^2 + 2y$. Najděme gradient funkce f:

$$f(x, y) = x^2 + 2y$$

$$\frac{\partial f}{\partial x} = 2x$$

$$\frac{\partial f}{\partial y} = 2$$

A tedy gradient funkce f je:

$$\nabla f = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} 2x \\ 2 \end{bmatrix}$$

Jak jsme si pověděli, gradient funkce je tedy vektor parciálních derivací a říká nám velice důležitou věc a to **směr, kterým daná funkce roste nejvíce**. Představme si, že jsme v bodě $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ a zajímá nás, kterým směrem se vydat, abychom šli co nejvíce do kopce. To se znalostí gradientu není žádný problém, neboť nám stačí dosadit a dostaneme $\begin{bmatrix} 2 \cdot 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$ a vidíme, že se musíme vydat ve směru $\begin{bmatrix} 2 \\ 2 \end{bmatrix}$.

Nyní si ukážeme tři grafy. Na grafu 4.8 je vyobrazena naše funkce f. Graf 4.9 vyobrazuje směry gradientu v jednotlivých bodech a konečně graf 4.10 vyobrazuje rovněž směry gradientu, ale pouze v rovině, kde $f(x, y) = 0$.

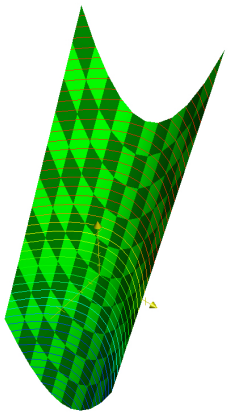
Více viz [14] str. 56

4.10 Derivace složených funkcí

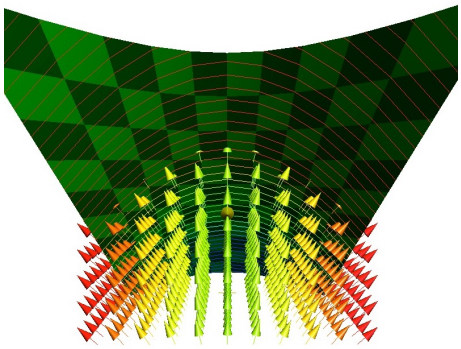
I přes to, že jsme si řekli, že se zde nebudeme učit konkrétní způsoby jak derivovat, tak si alespoň toto pravidlo společně připomeneme, neboť je klíčové pro skutečné pochopení matematiky skrývající se za neuronovými sítěmi. Pojďme si pravidlo vypsat a poté si o něm povíme více:

$$(f(g(x)))' = f'(g(x)) \cdot g'(x) \tag{3}$$

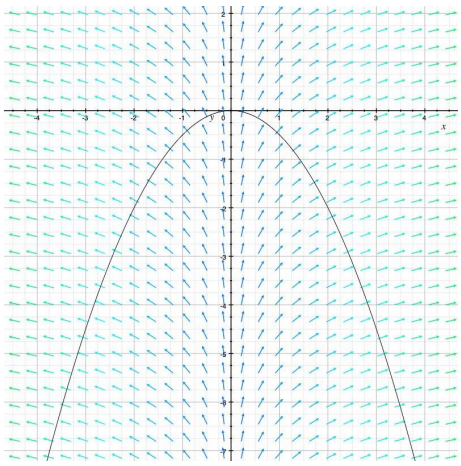
Nenechme se zastrašit zápisem, jelikož nám říká něco logického. Pojďme si to společně demonstrovat.



Obrázek 4.8. Graf funkce $f = x^2 + 2y$.



Obrázek 4.9. Graf funkce $f = x^2 + 2y$ s vektorovým polem dle gradientu.



Obrázek 4.10. Graf funkce $f = x^2 + 2y$ s vektorovým polem dle gradientu v rovině.

Příklad 4.9. Mějme funkce $g(x) = x^3$ a $f(x) = 3x^2 + 4$. Spočítejme derivaci funkce $f(g(x))$. Nejprve se pokusíme si sami odvodit, jak na to a poté se podíváme na výše zmíněné pravidlo dle vzorečku (3). Máme tedy:

Odvození

$$g(x) = x^3$$

$$f(x) = 3x^2 + 4$$

A chceme získat:

$$f'(g(x))$$

Dále, pokud si čtenář vzpomíná, jsme probírali v (1), že následující zápisy jsou ekvivalentní. Pro intuitivnější vysvětlení tohoto pravidla využijeme druhý zmíněný. Tedy:

$$f'(g(x)) = \frac{\partial f}{\partial x}$$

Výše zmíněný přepis na pravé straně rovnítka nám říká úplně to samé „Zderivujme funkci f vůči proměnné x“. Jak bychom na to tedy šli? Jako pomůcku si nejdřív představíme, že chceme pouze vypočítat výslednou hodnotu funkce $f(g(x))$.

Z logické úvahy vyvodíme, že nejdříve musíme obdržet hodnotu pro vstupní proměnnou x. Až poté jsme schopni vypočítat výstup z $g(x)$ a konečně můžeme dosadit výstup z $g(x)$ jak vstup do $f(g(x))$. Tento myšlenkový pochod bychom si mohli znázornit takto:

$$x \to g(x) \to f(g(x)) \tag{4}$$

Ovšem stejnou optikou je nutné se dívat na derivaci složených funkcí. x nám ovlivní „rychlost růstu/klesání“ funkce $g(x)$, která na oplátku ovlivní „rychlost růstu/klesání“ funkce $f(g(x))$. Tedy, pokud chceme vypočítat následující derivaci:

$$\frac{\partial f}{\partial x}$$

Pak je nutné nejprve zjistit, jakým způsobem ovlivňuje proměnná x funkci $g(x)$ viz (4). Tedy:

$$\frac{\partial g}{\partial x}$$

Nyní, když disponujeme touto informací, můžeme dále zjistit, jak moc ovlivňuje výstup funkce g funkci $f(g(x))$:

$$\frac{\partial f}{\partial g}$$

V tuto chvilku prosím čtenáře, aby skutečně vnímal g jako obyčejnou proměnnou dle které derivujeme. Stejně, jako tomu je u obyčejného x. Koneckonců x je pouhé číslo, avšak v moment, kdy dosadíme toto číslo do $g(x)$, tak se z $g(x)$ rovněž stane pouhé číslo.

Nyní již máme oba členy, které nám ovlivňují změny ve funkci $f(g(x))$, tak si pojďme vypsat kompletní derivaci níže:

$$\frac{\partial f}{\partial x} = \frac{\partial g}{\partial x}\frac{\partial f}{\partial g} \tag{5}$$

Pokud zkonzultujeme vzoreček (4), pak si můžeme povšimnout, že jsme dospěli obdobného výsledku. Pokud jdeme zleva, pak vidíme, že nejprv musíme zjistit, jak nám

x ovlivní $g(x)$ tedy $x \to g(x)$. Tento fakt nám zde reprezentuje tento člen $\frac{\partial g}{\partial x}$. Poté pokračujeme dále a zjišťujeme, jakým způsobem nám $g(x)$ ovlivňuje funkci $f(g(x))$ tedy $g(x) \to f(g(x))$. Toto je reprezentováno druhým členem $\frac{\partial f}{\partial g}$. A z těchto dvou členů nám konečně vyjde, jak ovlivňuje proměnná x naši funkci $f(g(x))$.

Pojďme se na to podívat prakticky. S výše zadanými funkcemi si vypočítáme oba členy:

$$g(x) = x^3$$

$$\frac{\partial g}{\partial x} = 3x^2$$

Opět nechť čtenář vnímá $g(x)$ jako obyčejnou proměnnou a stejně tak se k ní chová. Pokud to pomůže, je možné si místo $g(x)$ představit například y - tomuto se říká substituce. Ale pozor, po dokončení derivace je nutné dosadit zpět. My se substituci vyhneme a čtenář sám zváží, který přístup je mu více po chuti.

$$f(x) = 3x^2 + 6$$

$$f(g(x)) = 3g(x)^2 + 6$$

$$\frac{\partial f}{\partial g} = 6g(x)$$

Se znalostí obou členů si dosadíme do vzorečku (5) a získáme tedy:

$$\frac{\partial f}{\partial x} = 3x^2 \cdot 6g(x)$$

A jelikož znění $g(x)$ známe, pak si můžeme opět dosadit a získáme:

$$\frac{\partial f}{\partial x} = 3x^2 \cdot 6x^3 = 18x^5$$

Pomocí vzorečku (3)

Již nebudeme znovu počítat derivace funkcí f a g, neb jsou spočítány výše.

$$f'(g(x)) = f'(g(x)) \cdot g'(x)$$

$$f'(g(x)) = 6g(x) \cdot 3x^2$$

$$f'(g(x)) = 6x^3 \cdot 3x^2 = 18x^5$$

Více viz [13] str. 68

4.11 The Hadamard product

Hadamardův součin, značený $\odot$ je méně obvyklá operace se dvěma maticemi o stejném rozměru.

Příklad 4.10. Mějme matice A a B o stejém rozměru. Najděme $A \odot B$:

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix}; b = \begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

$$A \odot B = \begin{bmatrix} ae & bf \\ cg & dh \end{bmatrix}$$

Jednoduše řečeno vynásobíme položky na stejných pozicích.

Více viz [11] str. 32

Kapitola 5
Vstupní data

V této kapitole si podrobně popíšeme vstupní data z MNIST datasetu a naimplementujeme si program, který je bude schopen načíst a posléze transformovat do formátu, se kterým se nám bude později lépe pracovat.

Program pro načtení je součástí, jakožto příloha A.

5.1 Formát dat

Data MNIST datasetu nejsou uložena v žádném standardizovaném obrazovém formátu a napsat, či užít nějaký program, který nám je převede do námi potřebného formátu, je tedy nutností. Samotný formát souborů je popsán na stránkách MNIST[1]. My si jej ovšem projdeme společně.

Jedná se o celkem 4 soubory, kde tréninková sada obsahuje **60 000 vzorků** a testovací sada obsahuje **10 000 vzorků**. Pro úplnost si uvedeme soubory, ze kterých se dataset skládá.

- **train-images-idx3-ubyte.gz**: Tréninková množina obrázků
- **train-labels-idx1-ubyte.gz**: Tréninková množina popisků
- **t10k-images-idx3-ubyte.gz**: Testovací množina obrázků
- **t10k-labels-idx1-ubyte.gz**: Testovací množina popisků

Pozorný čtenář si jistě povšiml, že vzorky a popisky jsou ve dvou různých souborech. Jak je tedy budeme párovat? Odpověď na tuto otázku je velice jednoduchá, neboť v obou souborech je stejný počet položek, a tedy prvnímu vzorku odpovídá první popisek a tak dále.

5.1.1 Obrázky

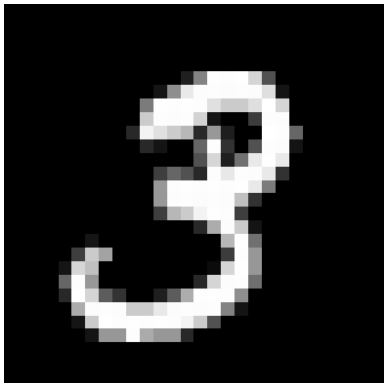
Nejprve si popíšeme, jakým způsobem je obrázek reprezentován. Každý obrázek je černobílý, čtvercový a má délku strany 28 pixelů. Jeden obrázek tedy tvoří $28 \cdot 28 = 784$ čísel, která jsou v rozsahu 0 - 255.

Pokud má pixel hodnotu 0, pak to znamená, že barva je bílá. Naopak pokud pixel nabývá hodnoty 255, pak je barva černá. A jakákoliv hodnota mezi je stupeň šedi.

Poznámka 5.1. Pro někoho, kdo ví, jakým způsobem funguje barevný model RGB se to může zdát matoucí, nicméně v těchto souborech byl zvolen tento formát a my jej musíme respektovat.

Pro lepší představu s čím vlastně budeme pracovat a co je myšleno **obrázkem** si jeden rovnou ukážeme viz obrázek 5.1

[1] `http://yann.lecun.com/exdb/mnist/`



Obrázek 5.1. Vykreslený vzorek čísla 3

5.1.2 Popisky

Soubory s popisky obsahují pro každý obrázek právě jeden byte, který nabývá hodnot 0 - 9. Toto číslo je popisek a koresponduje s tím, která číslice je skutečně napsána na obrázku. Pokud bychom tedy vzali v potaz výše uvedený příklad viz 5.1, pak bychom obdrželi ze souboru s popisky hodnotu **3**.

Upozornění: Pro pochopení problematiky neuronových sítí není nutné následující sekce procházet, jelikož je v nich popsána pouze implementace transformačního programu a nikterak více s tématem nesouvisí. Je tedy možno pokračovat kapitolou 6.

5.2 Struktura souborů

Nyní již víme, jak jsou data reprezentována, tak se můžeme podívat na strukturu souborů, ve kterých jsou uložena. Soubory nejsou žádného standardního formátu a jsou v binární formě. Po stažení prohlížečem je nutné provést jejich dekompresi a až poté je možno je zpracovat.

- Je možné, že prohlížeč provedl dekompresi, aniž by o tom informoval. V takovém případě stačí ignorovat koncovku .gz, či ji odstranit
- Veškerá čísla typu integer v souboru jsou uložena v MSB neboli (high endian / big endian) formátu

5.2.1 Popisky

Soubory: **train-labels-idx1-ubyte**, **t10k-labels-idx1-ubyte**
Popisek je číslice od 0 do 9, kdy jeho hodnota určuje korektní označení pro číslici napsanou na daném obrázku. Struktura je k vidění v tabulce 5.1.

Posun (v bytech)	Datový typ	Hodnota	Popis
0000	32bit integer	0x00000801 (2049)	Magické číslo (MSB)
0004	32bit integer	60000	Počet položek
0008	unsigned byte	??	Popisek
0009	unsigned byte	??	Popisek
...			
xxxx	unsigned byte	??	Popisek

Tabulka 5.1. Struktura souboru s trénovacími popisky MNIST datasetu. [15]

Posun (v bytech)	Datový typ	Hodnota	Popis
0000	32bit integer	0x00000803 (2051)	Magické číslo (MSB)
0004	32bit integer	60000	Počet položek
0008	32bit integer	28	Počet řádků (Výška obrázku)
0012	32bit integer	28	Počet sloupců (Šířka obrázku)
0016	unsigned byte	??	Pixel
0017	unsigned byte	??	Pixel
...			
xxxx	unsigned byte	??	Pixel

Tabulka 5.2. Struktura souboru s trénovacími vzorky MNIST datasetu. [15]

5.2.2 Obrázky

Soubor: **train-images-idx3-ubyte**, **t10k-images-idx3-ubyte**
Pixel je číslo od 0 do 255, kdy 0 znamená bílá, 255 černá a cokoli mezi je stupeň šedi. Pixely jsou uspořádány po řádcích. Struktura je k vidění v tabulce 5.2.

5.3 Program pro transformaci a načtení dat

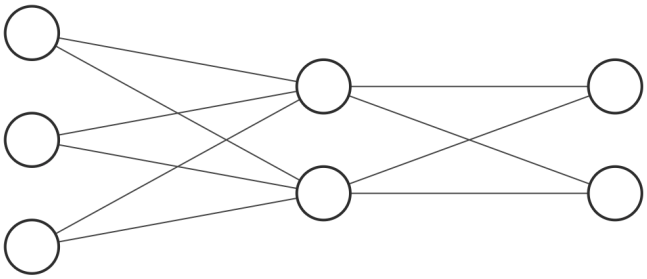
Jelikož program pro transformaci a načítání dat z MNIST datasetu není primární zaměření této práce, tak jej zde v teoretické části pouze zmíníme. Program je samozřejmě k dispozici jako příloha a v nadcházejících částech této práce jej budeme využívat jako černou skříňku k načtení dat.

Poznámka 5.2. Avšak, aby práce byla kompletní, tak je program v příloze A včetně verze napsané v Jupyter notebooku s vysvětlivkami.

Kapitola 6
Co to vlastně je Neuronová síť

Již jsme se hrubě seznámili s definicí našeho problému a především se vstupními daty, která budeme využívat. Nyní se tedy můžeme konečně pustit do hlavní části této práce a to co vlastně neuronová síť je, jak se učí a z čeho se skládá.

Začneme typickým obrázkem, který je většinou prezentován ve spojitosti s neuronovými sítěmi a řekneme si z čeho se taková síť skládá.



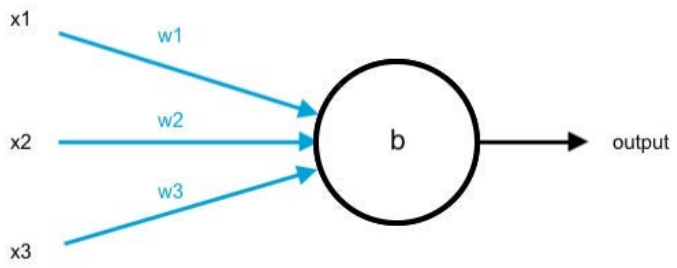
Obrázek 6.1. Typické znázornění neuronové sítě. Kolečka reprezentují neurony a spojnice reprezentují váhy.

6.1 Neuron

Jak již název sítě napovídá, tak primární komponentou sítě je neuron a na obrázku 6.1 jich můžeme vidět hned 7 a jsou reprezentovány „kolečky“.

Co tedy neuron vlastně dělá? Neurony jsou ve skutečnosti pouze jednoduchými funkcemi a v rámci sítě jsou uspořádány po vrstvách. Síť na obrázku 6.1 má pro představu vrstvy tři. Samotný neuron můžeme vidět na obrázku 6.2. V tuto chvíli se budeme bavit konkrétně o tomto neuronu, avšak obecně může mít vstupů kolik je potřeba, tedy n vstupů.

- Neuron má vstupy: x_1, x_2, x_3
  - Vstupy mohou nabývat libovolné hodnoty v intervalu $\langle 0, 1 \rangle$
- Neuron má takzvané váhy w_1, w_2, w_3
- Neuron má takzvaný bias b
- Neuron má výstup *output*
  - Výstup je pouze jeden, ač to například na obrázku 6.1 vypadá, že jich má více. Takto se neuronové sítě často kreslí, avšak znamená to, že jeden a ten samý výstup jde jako vstup do vícero neuronů. Nikoliv, že by měl neuron více výstupů.



Obrázek 6.2. Neuron

- Výstup může nabývat libovolné hodnoty v intervalu $\langle 0, 1 \rangle$

Co tyto hodnoty vlastně jsou? Každá z nich není nic jiného než pouhé číslo. A jak jsme si již řekli - neuron je čistě matematická funkce. Pojďme se tedy podívat, jak bychom výstup z neuronu mohli spočítat.

$$output = \sum_j (w_j x_j) + b \tag{1}$$

Teorie 6.1. $\sum$ viz 4.1

Vyzbrojeni znalostí sumy můžeme ze vzorce vyčíst, že každý vstup vynásobíme jeho vahou, vše sečteme a na závěr přičteme bias. Tím získáme výstup z neuronu. Abychom ovšem zaručili, že výstup - tedy *output* - z našeho neuronu bude vždy v intervalu $\langle 0, 1 \rangle$, tak *output* ještě „proženeme“ takzvanou sigmoid funkcí:

$$\sigma(output) = \frac{1}{1 + e^{-output}}$$

Teorie 6.2. Sigmoid funkce viz 4.2

Již jsme si vysvětlili jak takový neuron nakládá se svými vstupy a co generuje za výstup. Dále jsme si představili konkrétní vzorečky, ovšem s drobnými nepřesnostmi. Pojďme si nyní představit verzi s korektní notací pro vrstvu (l) ve formě složek:

$$z_j^{(l)} = \sum_j w_j^{(l)} a_j^{(l-1)} + b^{(l)} \tag{2}$$

$$a^{(l)} = \sigma(z_j^{(l)}) = \frac{1}{1 + e^{-z_j^{(l)}}} \tag{3}$$

Teorie 6.3. Vrstvy viz 3.1

Můžeme si povšimnou pár změn, které je nejlepší prostudovat v kapitole 3. Nicméně hlavní změnou je, že vzorcem (1) ve skutečnosti vypočítáme vážený vstup 3.3, tedy z, který následně vložíme jako vstup do sigmoid funkce. Až poté obdržíme výstup, neboli aktivaci neuronu 3.4, kterou značíme pomocí a.

Viz [7] kapitola 1, sekce Sigmoid neurons

6.2 Výstup sítě

Nyní již víme, jakým způsobem jednotlivé neurony generují své výstupy. Jak si to ovšem zasadit do kontextu sítě jako celku, jak nám generuje výstup samotná síť a jak jej interpretovat?

Pro začátek si upravíme notaci vzorečků (2) a (3) do vektorové podoby, která nám zpřehlední notaci. Tedy:

$$\mathbf{z}^{(l)} = \mathbf{w}^{(l)} \cdot \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \tag{4}$$

Teorie 6.4. Skalární součin viz 4.3

$$\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)}) = \frac{1}{1 + e^{-\mathbf{w}^{(l)}}} \tag{5}$$

Teorie 6.5. Aplikace sigmoid funkce na vektor viz 3.5

Se znalostí těchto vzorečků předvedeme, jakým způsobem síť dodává výstup. Pokud se podíváme zpět na obrázek 6.1, pak víme, že první vrstva nalevo je vrstva vstupní, kterou můžeme značit jako vektor vstupních aktivací $\mathbf{a}^{(0)}$. Tato data musíme síti dodat my a v našem případě jde tedy o obrázek z MNIST datasetu, který tvoří 784 čísel viz 5.1.1. Vstupní vrstva naší sítě by tedy měla 784 neuronů.

Co se týče vah a biasů, tak ty jsou na počátku **náhodně určeny** a postupně se trénují. Tedy - máme je rovněž zadány.

Tím máme již vše co je potřeba, jelikož vzorečky (4) a (5) nic víc nevyžadují a my je budeme pouze řetězově aplikovat, dokud se nedostaneme až po $\mathbf{a}^{(L)}$ - tedy výstup.

$$\mathbf{z}^{(1)} = \mathbf{w}^{(1)} \cdot \mathbf{a}^{(0)} + \mathbf{b}^{(1)}$$

$$\mathbf{a}^{(1)} = \sigma(\mathbf{z}^{(1)})$$

$$\mathbf{z}^{(2)} = \mathbf{w}^{(2)} \cdot \mathbf{a}^{(1)} + \mathbf{b}^{(2)}$$

$$\mathbf{a}^{(2)} = \sigma(\mathbf{z}^{(2)})$$

$$\vdots$$

$$\mathbf{z}^{(L)} = \mathbf{w}^{(L)} \cdot \mathbf{a}^{(L-1)} + \mathbf{b}^{(L)}$$

$$\mathbf{a}^{(L)} = \sigma(\mathbf{z}^{(L)})$$

6.2.1 Interpretace výstupní vrstvy

Ač je interpretace výstupní vrstvy zmíněna v 3.1.3, bude jistě lepší si ji projít přímo zde v kontextu toho, co jsme si ukázali. Vektor aktivací $\mathbf{a}^{(L)}$ můžeme tedy považovat za výstup ze sítě. Jak takový vektor poté interpretovat?

Z povahy naší úlohy, kdy chceme rozpoznat číslice $0-9$ je patrné, že máme celkem 10 kategorií, do kterých chceme naše vstupní data klasifikovat. Ve výstupní vrstvě naší sítě bude tedy celkem 10 neuronů, kdy první neuron bude korespondovat číslici nula, druhý číslici jedna, ..., desátý číslici devět. Poté ten neuron s nejvyšší aktivací bude předpověď naší sítě. Tedy odhad sítě poznáme jako index s nejvyšší hodnotou v našem vektoru $\mathbf{a}^{(L)}$.

6.3 Neuronová síť

Nyní již máme určité znalosti k tomu, abychom si mohli přiblížit neuronovou síť jakožto celek. Již víme, že se skládá z jednotlivých vrstev, které tvoří neurony. Tyto vrstvy neuronů jsou následně mezi sebou propojeny vahami. Dále víme, že každý neuron zvlášť je jednoduchá funkce, která má nějaké vstupy a **svůj výstup může ovlivnit úpravou buď váhy nebo biasu**. S touto znalostí lze vyvodit, že celá neuronová síť je rovněž pouze matematická funkce, která je složena z jednotlivých funkcí neuronů.

Poznámka 6.1. Nechť si čtenář povšimne tučného textu v předchozím odstavci. Pro někoho tato informace nemusí být nikterak překvapivá, druhému naopak nemusí dávat smysl. Je důležité si uvědomit, že ovlivnit výstup naší sítě lze pouze analogicky k ovlivnění výstupu neuronu. U neuronu je to viditelné jasně. Díky tomu lze ovšem vyvodit, že i výstup ze sítě půjde ovlivnit pouze změnou váhy či biasu, neboť je složena z jednotlivých neuronů.

Jako další důkaz lze brát to, že pokud bychom chtěli měnit přímo aktivace, pak bychom museli měnit i $\mathbf{a}^{(0)}$ = vstupní obrázek, což vychází z faktu, že aktivace se počítají řetězově. Což samo o sobě nedává smysl, jelikož my chceme, aby se síť pro daný vstup sama naučila generovat odpovídající výstup. Nikoliv, abychom my upravovali vstup tak dlouho, dokud by jej síť korektně neklasifikovala. (Nehledě na to, že by tou dobou ze vstupního obrázku byl pravděpodobně šum)

Kapitola 7
Jak se síť učí

Již jsme si představili obecný náhled na to, co vlastně neuronová síť je. Že je to složená funkce ze spousty menších funkcí (neuronů) a že se dělí na vrstvy a má nějaké vstupy a výstupy. Jak se ale síť ve skutečnost učí? V této kapitole si získané znalosti převedeme na náš konkrétní příklad s rozeznáváním psaných číslic.

7.1 Konkrétní podoba sítě

Ještě, než se vrhneme do vysvětlování, bude vhodné si konkrétně představit síť, kterou budeme společně implementovat, abychom se nemuseli stále pohybovat v obecné rovině, což by pro některé čtenáře mohlo být zbytečně abstraktní. Pojďme si tedy ukázat, jakou bude mít naše síť podobu.

Jak jsme se již bavili v kapitole 5, tak každý obrázek z datasetu MNIST má rozměry $28 \cdot 28 = 784$ pixelů, kde každým pixelem je myšleno číslo v rozmezí $\langle 0, 255 \rangle$. Náš vstupní program ovšem tato data transformuje a každé číslo převede do intervalu $\langle 0, 1 \rangle$, kde $0 = \check{c}ern\acute{a}$, $1 = b\acute{i}l\acute{a}$, cokoliv mezi je stupeň šedi.

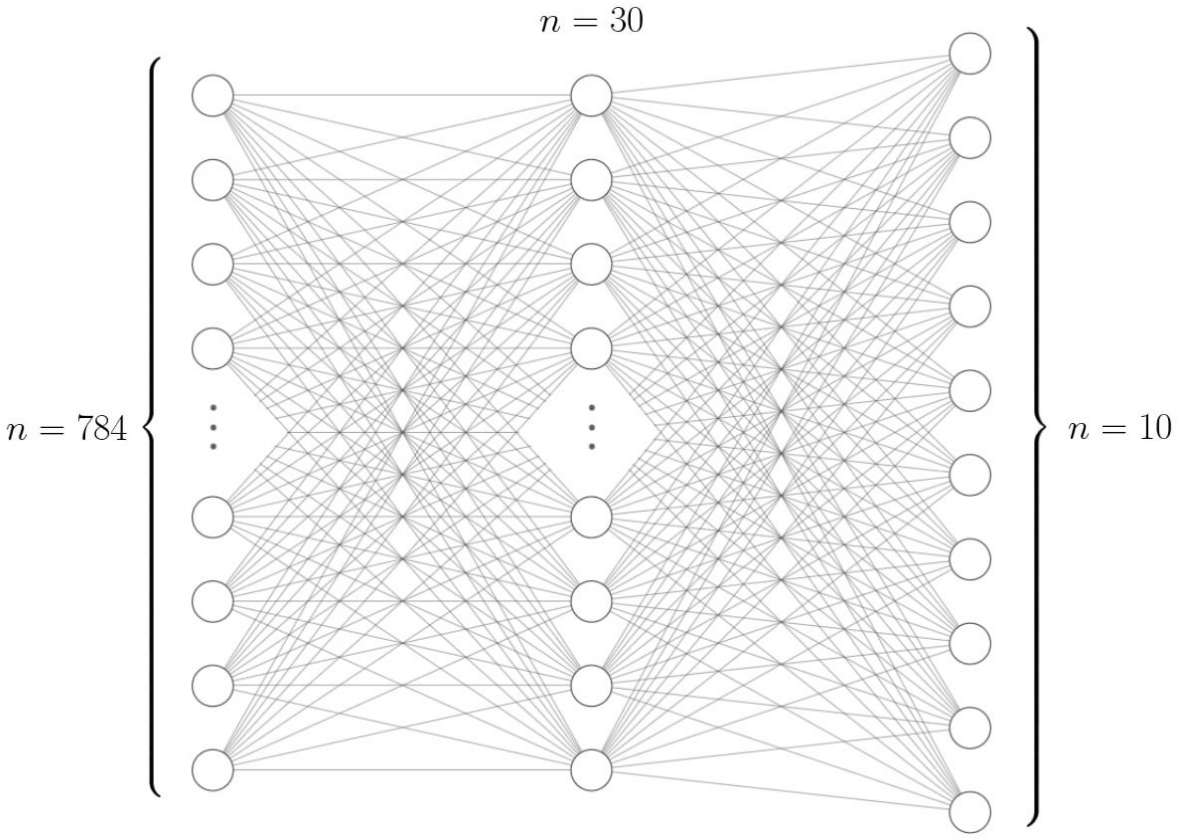
A jelikož chceme, aby naše síť rozpoznala číslici na obrázku, pak je daný obrázek zároveň vstupem do sítě. Proto naše síť bude mít **784 vstupních neuronů**. Dále víme, že na obrázku může být číslice od nuly do devíti - celkem máme tedy 10 možností. Z tohoto faktu nám vychází, že výstupní vrstva bude mít **10 neuronů**, kde výstup z každého neuronu bude značit pravděpodobnost s jakou si síť myslí, že na vstupním obrázku je daná číslice. A neuron s nejvyšší hodnotou, tedy nejvyšší pravděpodobností, bude oficiální odhad sítě. Pokud tedy bude mít první neuron nejvyšší hodnotu, pak odhad sítě bude číslice 0. Je vhodné připomenout, že prvním neuronem je myšlena nula a desátým neuronem se myslí číslice devět. A konečně skrytou vrstvu budeme mít pouze jednu a o velikosti **30 neuronů**. Počet a velikost skrytých vrstev je zvolen náhodně a je možno s těmito údaji experimentovat. Naše síť je znázorněna na obrázku 7.1

7.2 Dopředný průchod

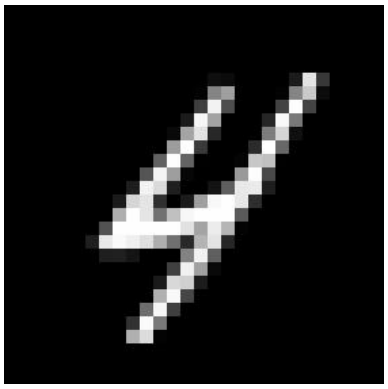
Dopředný průchod (*anglicky feed forward*) a jakým způsobem získáme výstup ze sítě jsme se již bavili zde: 6.2. Pojďme si toto téma ovšem projít s naší konkrétní sítí a vizuálně si ukázat, jak si představit obrázek, jakožto vektor aktivací $\mathbf{a}^{(0)}$. Jak převést vzorce, které jsme si představili v minulé kapitole do počítačové programu v jazyce Python a zároveň si ukážeme implementaci metody, která nám řekne jaký je výsledný odhad sítě.

7.2.1 Vstupní vrstva

Teorie 7.1. Vstupní vrstva viz 3.1.1



Obrázek 7.1. Diagram naší neuronové sítě, kde n značí počet neuronů v dané vrstvě.



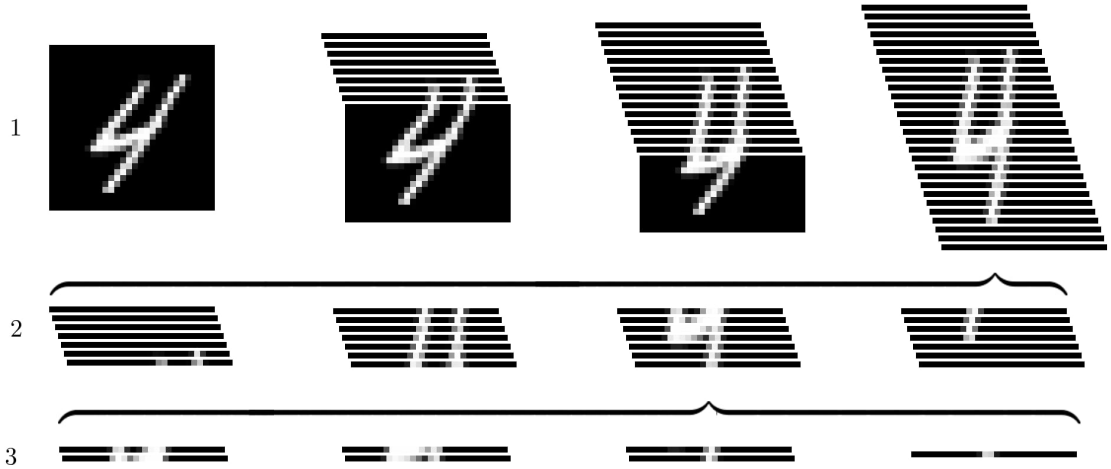
Obrázek 7.2. Ukázka obrázku, který využijeme pro náš příklad vstupu do sítě

Již víme, že vstupní vrstva je náš vstup, který síti dodáme. V našem případě obrázek z MNIST datasetu. Jak si ale takový obrázek představit, jako vektor $\mathbf{a}^{(0)}$? Nejprve se podívejme, jak může takový obrázek vypadat viz obrázek 7.2

Tento obrázek, jako všechny v MNIST datasetu, má $28 \cdot 28 = 784$ pixelů. V tento okamžik jsou uspořádány do mřížky. Pojďme si ovšem představit, že bychom je přeorganizovali do jedné dlouhé řady čísel. V tom nám pomůže obrázek 7.3. Na obrázku máme několik kroků:

1. Zde je vyobrazeno, že obrázek lze vnímat jakoby byl složen z řádků.
2. Jednotlivé řádky pak lze přeuspořádat, aby byly vedle sebe.

3. Takto lze pokračovat, dokud nebudeme mít jednu dlouhou řadu složenou ze 784 čísel. (Kvůli nedostatku místa je v tomto kroku rozdělen pouze jeden dílek.)



Obrázek 7.3. Ukázka, jak si představit obrázek, který je reprezentovaný jako pouhá řada čísel

Tímto způsobem je obrázek ve skutečnosti (po transformaci dat naším programem) v paměti počítače i reprezentován. Jako řada čísel a jenom pro to, abychom ho mohli vidět my, lidé, je přeuspořádán do mřížky. Půjdeme ještě o krok dál a výslednou řadu čísel transponujeme. Tím nám vznikne sloupcový vektor (neboli matice se 784 řádky a 1 sloupcem), který už v podstatě odpovídá vstupu do sítě.

Teorie 7.2. Transpozice viz 4.5

7.2.2 Implementace metody feed forwad

Samotný dopředný průchod je tedy proces, kdy vezmeme vstupní vektor aktivací $\mathbf{a}^{(0)}$ (naše vstupní data) a postupně vrstvu po vrstvě je „proženeme“ sítí, dokud nedostaneme vektor aktivací $\mathbf{a}^{(L)}$ - tedy výstup sítě. Jak toho docílit matematicky jsme probírali zde 6.2. Pojďme si nyní představit možný algoritmus pro tuto metodu v jazyce Python viz obrázek 7.4:

```
def feedforward(a):
    for l in range(L):
        a = sigmoid(np.dot(w, a) + b)
    return a
```

Obrázek 7.4. Implementace metody feedforward v jazyce Python

Jak můžeme vidět, na vstupu má metoda parametr a, kterým je myšlen vstup, tedy $\mathbf{a}^{(0)}$. Dále můžeme ve funkci vidět L, kde se předpokládá, že dříve v programu je L definováno a je tím myšlen počet vrstev.

Poté jíž provedeme cyklus pro každou vrstvu (l), kde postupujeme dle vzorečku (5) na straně 29. A konečně „np.dot“ je implementace skalárního součinu v knihovně numpy. Funkce feedforward vrátí výstup naší sítě - tedy vektor aktivací $\mathbf{a}^{(L)}$, který lze interpretovat jako vektor pravděpodobností, kde ta největší je výsledný odhad.

Poznámka 7.1. Povšimněme si, že ovlivnit výstup sítě lze pouze manipulací s vektory $\mathbf{w}$ a $\mathbf{b}$. Tedy není možné přímo měnit aktivace jednotlivých vrstev. Ty se budou lišit v důsledku změn v již zmíněných vektorech. Více viz 6.1.

Teorie 7.3. Informace o „np.dot“ viz 4.1

7.3 Chybová funkce

Nyní již máme nástroje k tomu, abychom ze sítě dostali pro nějaký konkrétní vstup odpovídající výstup. Bohužel, v tuto chvíli bude výstup zcela náhodný, což není žádoucí. Co v takovém případě dělat a jak síť naučit, aby její výstupy byly relevantní?

7.3.1 Intuice

Pojďme na chvilku odbočit a představme si, že chceme kamaráda naučit na snowboardu, na kterém sami umíme. Máme tedy nějakou představu o tom, jak vypadá kvalitní technika. První krok k úspěchu je oznámit kamarádovi, že něco dělá špatně. Nicméně nám to nebude co platné, jelikož pokud bychom na něj pouze povykovali, že jezdí špatně, tak se pravděpodobně moc nezlepší a zřejmě bychom si svou pozici učitele také moc dlouho neužili.

Druhý, dost možná i klíčový, krok je samozřejmě kamarádovi sdělit, co přesně dělá špatně a především **co má udělat, aby to zlepšil**.

Vezměme si tuto analogii a zkusme ji převést do našeho matematického světa. Musíme tedy naší síti sdělit, že něco dělá špatně **a co má udělat**, aby si vedla lépe. Jak ale síti sdělit, že je něco špatně, aby byla schopna příště odhadovat jinak? Představíme si takzvanou chybovou funkci, jejíž výstup bude velikost chyby, které se síť dopouští. A jelikož my známe onu „kvalitní techniku“, kterou jsou v tomto případě popisky k danému obrázku se správnou číslicí, tak můžeme síti sdělit, jak špatný daný odhad byl.

7.3.2 Definice chybové funkce

Definice 7.1. Nechť C je chybová funkce (*anglicky cost function*) neuronové sítě a je dána jako:

$$C(\mathbf{w}, \mathbf{b}) = \frac{1}{2n} \sum_{x} \|\mathbf{y}(\mathbf{x}) - \mathbf{a}^{(L)}\|^2 \tag{1}$$

Poznámka 7.2. Berme na vědomí, že chybová funkce může mít různé podoby. Pro naše účely bude ovšem použita výše definovaná.

Nenechme se zastrašit tím, jak chybová funkce vypadá a pojďme si ji společně rozklíčovat. $\mathbf{w}$ značí vektor všech vah a $\mathbf{b}$ je vektor všech biasů v síti. $\mathbf{x}$ je jeden konkrétní vstup (obrázek), pak tedy $\mathbf{y}(\mathbf{x})$ je požadovaný výstup z naší sítě (předem známý popisek) pro vstup $\mathbf{x}$. Jak si pozorný čtenář jistě všiml, tak je $\mathbf{y}(\mathbf{x})$ značeno jakožto vektor, i přes to, že jsme si dříve pověděli, že popisek je pouhé číslo. Nechť čtenář v tuto chvíli pokračuje v četbě, neboť se k vysvětlení dostaneme v poznámce 7.4. Výstupem z naší sítě je $\mathbf{a}^{(L)}$. T termín $C(\mathbf{w}, \mathbf{b})$ nám říká, že chybová funkce závisí na jednotlivých vahách a biasech. $\|$ je klasická velikost vektoru (tedy norma) a konečně $\frac{1}{2n}\sum_x$ nám říká, že průměrujeme přes všechny tréninkové vstupy.

Poznámka 7.3. Povšimněme si, že výstupem funkce C je skalár. Tedy obyčejné číslo, které indikuje jak dobře, či špatně si síť vede. Čím je číslo vyšší, tím je chyba vyšší.

Z rovnice je patrné, že $\mathbf{w}$ a $\mathbf{b}$ jsou vektory. Nicméně si můžeme představit, že do funkce C parametry zadáme následujícím způsobem $C(w_1, w_2, \ldots, w_j, b_1, b_2, \ldots, b_k)$. Tedy jak je vidět, jde pouze o funkci mnoha proměnných a vektorový zápis je pouze kompaktnější.

Teorie 7.4. Funkce více proměnných viz 4.7

7.3.3 Co nám chybová funkce vlastně říká

A co nám tedy ta chybová funkce (1) vlastně říká? Mějme pro vstup $\mathbf{x}$ výstupní vektor $\mathbf{a}^{(L)}$ a odpovídající popisek $\mathbf{y}(\mathbf{x})$.

$$\mathbf{a}^{(L)} = [0.2, 0.33, 0.03, 0.12, 0.9, 0.83, 0.32, 0, 0.5, 0.1]^T$$

$$\mathbf{y}(\mathbf{x}) = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]^T$$

Poznámka 7.4. Čtenář si může povšimnout mírně odlišného formátu popisku. Klasicky jsme využívali pouze číslici. To je dostačující pro testovací účely. Nicméně pro trénink sítě využijeme stejného formátu, jako nám síť vrací - tedy vektor o 10 složkách, kde jsou všechny složky nulové, krom té na indexu značící číslici, o kterou se jedná. Hodnota na tomto indexu bude rovna jedné. Což, jak si jistě pozorný čtenář povšiml, odpovídá přesně ideálnímu výstupu sítě, který bychom požadovali.

Vektor $\mathbf{y}(\mathbf{x})$ pak značí, číslici 3. Síť si však „myslí“, že jde o číslici 4, neboť nejvyšší aktivace je na indexu korespondujícímu právě číslici 4. Společně si nyní dosadíme a vypočteme velikost chyby.

Poznámka 7.5. Pro tuto chvíli se zafixujeme pouze na jeden vstupní obrázek x, díky čemuž si pro zjednodušení přepíšeme cost funkci.

$$C(\mathbf{w}, \mathbf{b}) = \frac{1}{2}\|\mathbf{y} - \mathbf{a}^{(L)}\|^2$$

Příklad 7.1. Následující příklad ukazuje dosazení a výpočet do cost funkce. Máme výstup $\mathbf{a}^{(L)}$ a očekávaný výstup $\mathbf{y}$:

$$\mathbf{a}^{(L)} = [0.2, 0.33, 0.03, 0.12, 0.9, 0.83, 0.32, 0, 0.5, 0.1]^T$$

$$\mathbf{y} = [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]^T$$

Pokud dosadíme, pak získáme:

$$C(\mathbf{w},\mathbf{b}) = \frac{1}{2}\|[0,0,0,1,0,0,0,0,0,0]^T - [0.2,0.33,0.03,0.12,0.9,0.83,0.32,0,0.5,0.1]^T\|^2$$

$$C(\mathbf{w},\mathbf{b}) = \frac{1}{2}\|[-0.2,-0.33,-0.03,0.88,-0.9,-0.83,-0.32,0,-0.5,-0.1]^T\|^2$$

$$C(\mathbf{w},\mathbf{b}) \approx \frac{1}{2}1.669^2$$

$$C(\mathbf{w},\mathbf{b}) \approx \frac{1}{2}2.7855$$

$$C(\mathbf{w},\mathbf{b}) \approx 1.3927$$

Příklad 7.2. Nyní provedeme obdobný příklad, avšak dosadíme tak, že síť uhodne správně. Mějme tedy:

$$\mathbf{a}^{(L)} = [0.2,0.3,0.03,0.96,0.02,0.3,0.32,0.1,0.2,0.1]^T$$
$$\mathbf{y} = [0,0,0,1,0,0,0,0,0,0]^T$$

Opět dosadíme:

$$C(\mathbf{w},\mathbf{b}) = \frac{1}{2}\|[0,0,0,1,0,0,0,0,0,0]^T - [0.2,0.3,0.03,0.96,0.02,0.3,0.32,0.1,0.2,0.1]^T\|^2$$

$$C(\mathbf{w},\mathbf{b}) = \frac{1}{2}\|[-0.2,-0.3,-0.03,0.04,-0.02,-0.3,-0.32,-0.1,-0.2,-0.1]^T\|^2$$

$$C(\mathbf{w},\mathbf{b}) \approx \frac{1}{2}0.6207^2$$

$$C(\mathbf{w},\mathbf{b}) \approx \frac{1}{2}0.3853$$

$$C(\mathbf{w},\mathbf{b}) \approx 0.1927$$

Pokud porovnáme oba výsledky, můžeme pozorovat, že v moment, kdy síť odhadla správně, byla chyba řádově menší. Pokud bychom se nyní zase vrátili k původní variantě, kde se nebudeme fixovat pouze na jeden vstup, pak je postup analogický s tím rozdílem, že pouze uděláme průměr chyb pro všechny vstupy.

Poznámka 7.6. Povšimněme si, že jako vstupní parametry do funkce C jsou brány vektory $\mathbf{w}$ a $\mathbf{b}$ i přes to, že v těle funkce se na první pohled nevyskytují. A naopak jako parametr nejsou zadány vektor $\mathbf{y}(\mathbf{x})$ ani $\mathbf{a}^{(L)}$. Důvodem je to, že se vždy počítá chybová funkce pro daný vstupní obrázek, tedy C_x. Tím, že počítáme pro daný obrázek se nám zafixuje $\mathbf{y}(\mathbf{x})$ a síť nám vrátí $\mathbf{a}^{(L)}$. A co se týče $\mathbf{w}$ a $\mathbf{b}$, tak ty jediné lze ovlivnit a jejich role na výstupu sítě se projevuje ve vektoru $\mathbf{a}^{(L)}$, jelikož ten je vypočítán pomocí vzorce (3) na straně 28, kde již tyto vektory figurují.

Závěrem tedy je, že naším cílem je mít co nejmenší možnou chybu, tedy minimalizovat chybovou funkci. A na otázku, jakým způsobem lze minimalizovat funkci je jednoduchá odpověď: derivace.

Derivace viz 4.6.

```
def cost():
    n, cost_sum = len(training_data), 0
    for i in range(n):
        y, a = training_data[i]
        a = feedforward(a)
        cost_sum += np.lingalg.norm(y - a)**2
    return 1 / (2 * n) * cost_sum
```

Obrázek 7.5. Implementací metody cost v jazyce Python 3

7.3.4 Implementace chybové funkce

I přes to, že zatím nejsme v praktické části, se může občas hodit ukázat reálný kód. Díky tomu je možné dobře vidět, jak tyto doposud abstraktní znalosti reálně uplatnit. Implementaci lze prozkoumat na obrázku 7.5.

Proměnnou *training data* je myšlen soubor dat pro trénování sítě, který obdržíme pomocí dodávaného programu, jež načte MNIST dataset. Každá položka v této kolekci pak odpovídá dvojici (obrázek, vektorizovaný popisek). Kód k *feed forward* jsme probrali již zde 7.2.2. A závěrem „np.linalg.norm“ je metoda pro velikost vektoru v knihovně numpy.

7.4 Učení

V tuto chvíli jsme si řekli, jakým způsobem „povykovat na kamaráda“, pokud bychom opět užili analogii z 7.3.1. Jak jsme si ovšem pověděli, tímto bychom se daleko nedostali. Je zapotřebí najít způsob, jak skutečně učit. Pojďme si tedy představit algoritmus, který nám s tím pomůže.

7.5 Gradient descent

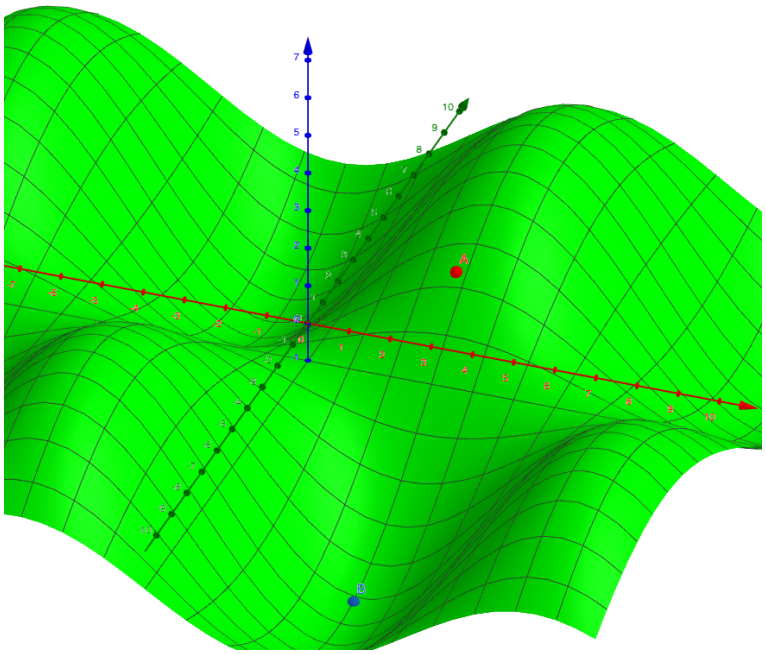
Teorie 7.5. Gradient viz 4.9

Vyzbrojeni znalostí gradientu jej nyní můžeme využít v praxi. Pozorného čtenáře již jistě napadlo, jakým způsobem budeme této znalosti využívat. Naším cílem je získat **gradient naší chybové funkce**, tedy vektor derivací podle všech jednotlivých vah a biasů, abychom věděli, jaký vliv má každý prvek na výslednou chybovou funkci a zároveň tedy směr, kterým se pohnout, abychom funkci co nejvíce minimalizovali.

7.5.1 Intuice

Naše chybová funkce bude mnoha proměnných, dokonce i malinká ukázková síť viz obrázek 6.1 má celkem 10 vah a 4 biasy. Její chybová funkce by tedy závisela na $14 - ti$ vstupních parametrech. Nicméně my si pro tuto chvíli představíme, že bychom měli vstupní parametry pouze dva, díky čemuž si budeme moci zakreslit graf. Mějme tedy nějakou chybovou funkci $C(w, b)$, která má na vstupu pouze jednu váhu a jeden bias. Představme si její povrch například takto, jako je znázorněno v grafu 7.6:

V grafu 7.6 můžeme zároveň vidět osy, které máme popsány v popisku grafu.



Obrázek 7.6. Ukázka možného grafu chybové funkce dvou proměnných. Červená osa značí váhu w, zelená osa bias b a modrá výstup funkce $C(w, b)$.

Dále můžeme v grafu spatřit dva body a to bod $A(červený)$ a bod $B(modrý)$. Tyto body nám reprezentují aktuální nastavení vah a biasů (bod A) a jedno z optimálních nastavení vah a biasů (bod B). Pokud by měla chybová funkce takovýto povrch, pak je snadné určit, kterým směrem se pohnout, abychom ji minimalizovali. Ovšem v praxi to tak zpravidla není, proto si zde ukážeme postup, který lze analogicky aplikovat i ve funkcích o více proměnných.

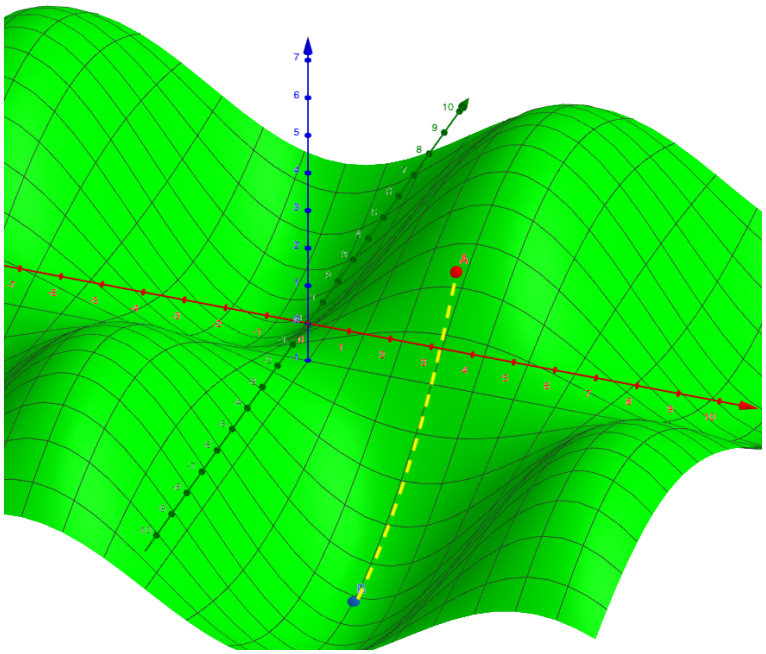
Jsme tedy v bodu A a zároveň jsme schopni získat ∇C. Ten nám říká směr, kterým funkce v daném bodě nejvíce roste. My ovšem chceme klesat, čehož lze docílit snadno. Stačí jít v záporném směru gradientu. Kdybychom tedy opět použili analogii jízdy na snowboardu, tak pokud bychom byli v bodě A a chtěli se co nejrychleji dostat do bodu B, pak bychom pravděpodobně zvolili následující trasu, jako je na obrázku 7.7:

Více viz [11] str. 82 a pro intuitivnější vysvětlivky viz [16]

7.5.2 Learning rate

Nyní si ovšem představme, že jedeme v noci a vidíme pouze kousek před sebe. Pokud bychom byli v bodě A a bezhlavě se vydali směrem, který se zdá nejvíce z kopce, pak bychom si mohli pěkně naběhnout. Co když bude na svahu zatáčka? V tuto chvíli přichází do hry tzv. **Learning rate**, kterou značíme pomocí η. Tato hodnota se nikterak nevypočítá, avšak je volena při spouštění sítě a je možné s ní experimentovat a zjistit, která na daný problém funguje nejlépe.

Co pro nás tedy dělá? Představme si, že naše η je dosah svítilny na svahu. Pokud bude dosah malý, tak pojedeme skutečně pomalu. Sice vždy uvidíme, kterým směrem máme jet, abychom nevyjeli ze svahu a zároveň abychom jeli co nejvíce z kopce, ale za cenu, že dojedeme za delší čas.



Obrázek 7.7. Ukázka cesty dle gradientu do lokálního minima

Opačný případ bychom si mohli představit tak, že naše svítilna sice dosvítí daleko, takže můžeme v klidu jet rychlou jízdou, ale vidíme pouze před sebe a můžeme přehlédnout například zkratku, která nás dolů zavede rychleji. V případě naší chybové funkce, bychom mohli udělat moc velký krok, a tedy „přestřelit“ naše chtěné minimum. Pokud bychom tyto kroky dělali stále velké, pak je možné, že bychom do minima nikdy nedorazili. Demonstraci, jak by se mohla chovat příliš vysoká hodnota paramteru η si můžeme prohlédnout na obrázku 7.8:

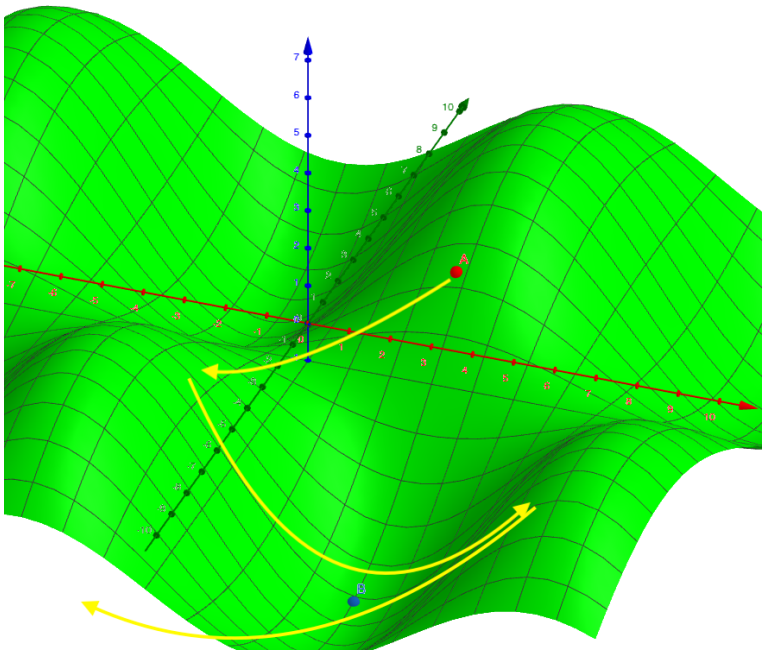
Jak je vidět, je zde prostor pro experimentování a žádná hodnota η není korektní. Existují pouze lepší a horší hodnoty pro určité problémy. Je tedy nutné najít nějaký rozumný kompromis, aby učení neprobíhalo příliš pomalu a naopak, abychom nebyli natolik rychlí, že bychom se nedostali do cíle.

7.6 Stochastic gradient descent

Stochastic gradient descent vychází z klasické verze tohoto algoritmu a v praxi pomáhá k rychlejšímu nalezení minima chybové funkce. Abychom pochopili jak nám pomáhá, pak musíme začít u předpisu chybové funkce, který si nyní osvěžíme:

$$C(\mathbf{w}, \mathbf{b}) = \frac{1}{2n} \sum_x \|\mathbf{y}(\mathbf{x}) - \mathbf{a}^{(L)}\|^2$$

Věnujme nyní pozor především členu $\sum_x$. Ten nám říká, že abychom vypočítali chybovou funkci, pak nejprve musíme vypočítat sumu chyb **pro všechna tréninková data** a z toho poté udělat průměr. To se na první pohled nemusí zdát problematické, zcela určitě z pohledu matematického. Naneštěstí nežijeme v teoretickém světě, ale praktickém a v tom na nás číhají nástrahy jako určitý hardwarový limit atd... I v našem



Obrázek 7.8. Ukázka příliš vysoké hodnoty η

případě máme z MNIST datasetu k dispozici 60 000 tréninkových dat. Pokud bychom si to převedli na čísla, která tvoří pouze obrázky, pak máme 47 040 000 vstupních čísel. Všechna tato čísla musíme dát na vstup do naší sítě, získat výstup, poté chybu a až teprve nyní uděláme průměr.

Doufám, že si čtenář udělal obrázek o tom, že tato operace může být skutečně hardwarově náročná. A to se bavíme o naší malé síti pro rozpoznávání číslic s 60 000 tréninkovými daty. Co teprve, pokud bychom měli miliony tréninkových dat. A právě zde přichází na scénu algoritmus **stochastic gradient descent**.

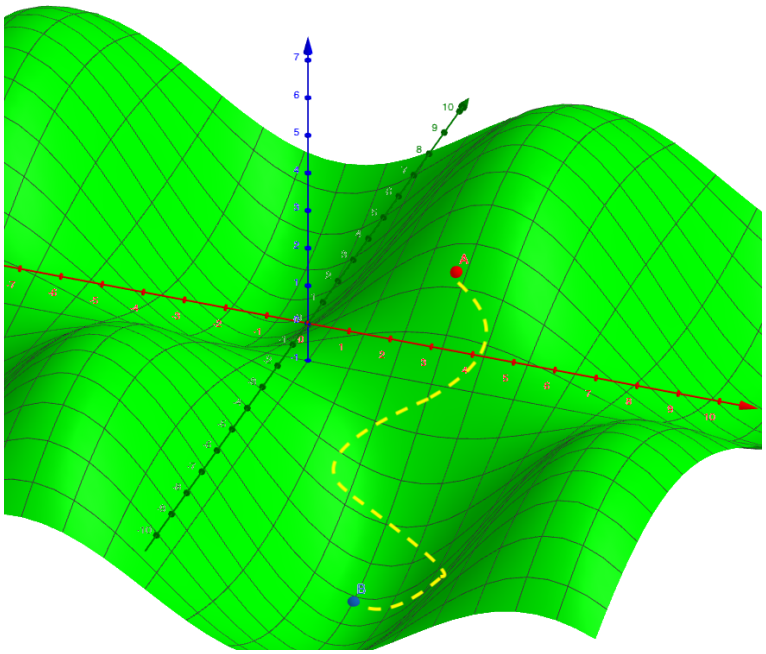
7.6.1 Princip

Výhoda je, že pokud čtenář chápe smysl algoritmu gradient descent, pak stochastic gradient descent není o nic složitější. Ovšem jeho síla panuje v tom, jakým způsobem budeme počítat onu chybovou funkci.

Naše tréninková data si totiž rozdělíme po menších částech do takzvaných mini-batchů. Velikost mini-batche je opět volitelná a lze s ní experimentovat. Výhoda pro nás nastává v tom, že poté budeme počítat gradient chybové funkci pouze z dat v tomto mini-batchi. Ano, je pravda, že gradient nebude zcela přesný. Bude ovšem dost přesný na to, abychom odhadli hrubý směr.

Opět se vrátíme k analogii snowboardu. Představme si, že místo, abychom jeli přímo dolů budeme dělat obloučky. Jelikož si nejsme jisti kudy přesně jet, tak jedeme chvilku sem, pak zjistíme, že jsme měli zatočit, tak se otočíme a tento proces opakujeme. Naše cesta nebude tedy přímá, ale spíše obloukovitá. Nicméně k cíli se dostaneme. Tento přístup znamená v praxi významné zkrácení času potřebného k vytrénovaní sítě. Podívejme se pro představu na obrázek 7.9:

Více viz [11] str. 286



Obrázek 7.9. Ukázka průběhu algoritmu stochastic gradient descent

7.6.2 Epocha

Jak jsme si řekli, z jednoho mini-batche jsme schopni spočítat hrubý gradient. Nicméně naše trénovací data jsou rozdělena do spousta mini-batchů a algoritmus gradient descent je iterativní. Tedy jakmile spočítáme jeden gradient a adekvátně upravíme váhy a biasy, pak přejdeme na další mini-batch a proces opakujeme.

V moment, kdy vyčerpáme veškeré mini-batche říkáme, že jsme dokončili **epochu** učení. V tento okamžik můžeme vstupní data náhodně promíchat, opět rozdělit a pokračovat v učení.

7.7 Aktualizace vah a biasů

Nyní již víme intuitivně, co musíme dělat, abychom chybovou funkci minimalizovali. Pojďme si proto sepsat nějaké konkrétní vzorce, dle kterých se budeme moci řídit. Z předchozího výkladu vyplynulo, že je nutné algoritmus aplikovat krokově a postupovat po kouskách. Velikost tohoto kroku nám určuje η a již víme, že pokud bude příliš velké můžeme „přestřelit“. Naopak pokud bude malé, pak nám může minimalizace funkce trvat skutečně dlouho. Z těchto postřehů nám tedy vyplývají následující poznatky:

- Potřebujeme gradient chybové funkce, tedy ∇C.
- Musíme si zvolit adekvátně velikou hodnotu pro η.
- Opakovaně musíme upravovat váhy a biasy tak, abychom se neustále přibližovali nějakému minimu.

Poznámka 7.7. Již jsme zmínili, že ve skutečnosti půjdeme ve směru záporného gradientu. Tento fakt budeme nadále značit, že se budeme posouvat tímto směrem: $-\eta\nabla C$. V tomto výrazu máme zahrnut korektní směr (pomocí $-\nabla C$) a zároveň velikost kroku, jelikož gradient proporcionálně upravíme dle η.

Bez dalšího odkládání si představme vzorec pro toho iterativní pravidlo k nalezení minima:

$$w_i \to w_i' = w_i - \eta \frac{\partial C}{\partial w_i} \tag{2}$$

$$b_j \to b_j' = b_j - \eta \frac{\partial C}{\partial b_j} \tag{3}$$

Pojďme si popsat vzorec pro aktualizaci váhy. Druhý vzoreček funguje naprosto stejným způsobem. Pomocí $w_i \to w_i'$ značíme aktualizaci z původní váhy w_i na novou hodnotu w_i'. Na druhé straně rovnítka máme postup. Máme původní váhu w_i, kterou posuneme ve směru dle záporného gradientu a velikost kroku nám určuje η. Pokud bychom se podívali na náš graf funkce dvou proměnných 7.6, kde červená osa značí váhu, pak by tento krok znamenal se posunout dle červené osy tak, aby se výsledná funkce C zmenšila. V tuto chvíli je více patrné, kterým směrem se posunout spíše na zelené ose, která reprezentuje bias. Nechť se čtenář v tomto kroku pozastaví a zamyslí se, kterým směrem by bylo potřeba se na červené ose posunout.

Poznámka 7.8. I přes to, že jsme se až do teď zabývali chybovou funkcí dvou proměnných, tak stejný postup se aplikuje u chybové funkce o libovolně velkém počtu proměnných. My jsme zvolili dvě, kvůli možnosti si danou funkci představit a vybudovat si určitou intuici.

Kapitola 8
Zpětná propagace

Pojďme společně pokračovat tam, kde jsme skončili v minulé kapitole. V té jsme zjistili, že abychom mohli síť učit, pak jí musíme sdělit jak špatně na tom je a především - **co má udělat, aby výsledky byly lepší**. Pokud si krátce zrekapitulujeme co víme:

1. Máme sadu trénovacích a testovacích dat. Každá položka, která se v těchto datech nachází je dvojice **(obrázek, popisek)**, kde obrázek je vektor vstupních dat a popisek je korektní číslice či vektorizovaný popisek značící odpovídající číslici.
2. Pomocí vzorce (3) ze strany 28 jsme schopni postupně vypočítat aktivace každé vrstvy, dokud nezískáme $\mathbf{a}^{(L)}$ - tedy výstup
3. Pro daný vstup $\mathbf{x}$, ke kterému jsme obdrželi výstup $\mathbf{a}^{(L)}$ máme ze vstupních dat zároveň odpovídající popisek $\mathbf{y}(\mathbf{x})$.
4. Jakmile máme $\mathbf{a}^{(L)}$ a $\mathbf{y}(\mathbf{x})$, pak jsme schopni vypočítat chybu dle chybové funkce (1) ze strany 34.

V této kapitole si povíme, jak pokračovat dále a intuici, kterou jsme si v minulé kapitole vypěstovali, využít v praxi.

Více viz např. [17]

8.1 Minimalizace chybové funkce

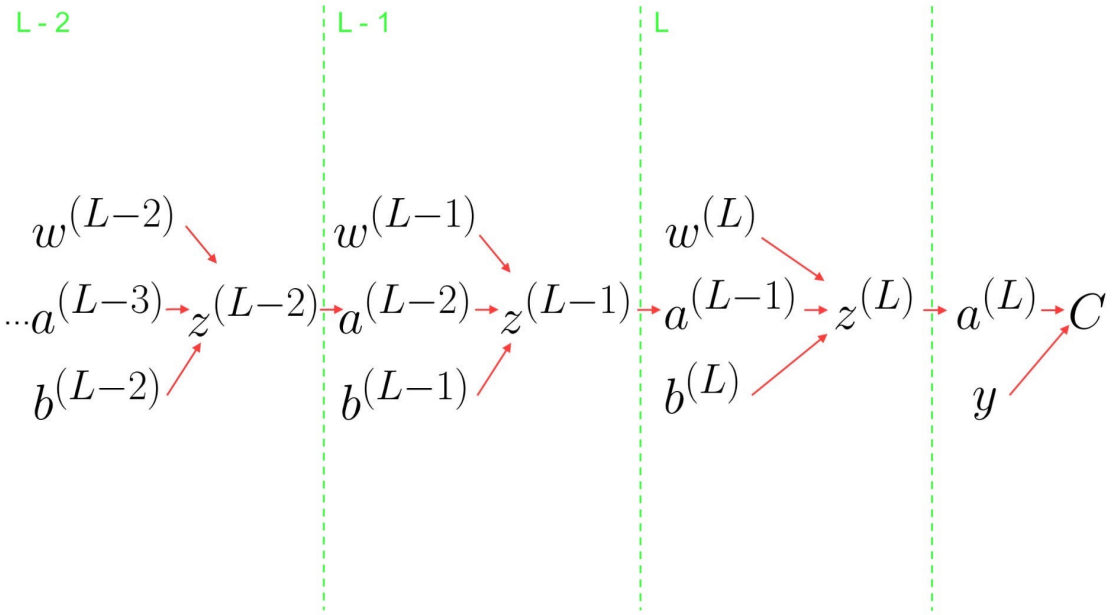
Jak již víme, a zároveň jsme na to upozornili i v poznámce 7.1, tak přímo měnit aktivace jednotlivých vrstev není možné → nelze přímo ovlivnit výstup sítě. Nicméně v poznámce je dále upozorněno na fakt, že toho docílit lze a to pomocí změn vektorů $\mathbf{w}$ a $\mathbf{b}$. Pojďme si připomenout chybovou funkci:

$$C(\mathbf{w}, \mathbf{b}) = \frac{1}{2n} \sum_x ||\mathbf{y}(\mathbf{x}) - \mathbf{a}||^2$$

Jak je patrné z levé strany zápisu, tak chybová funkce C závisí na vektorech $\mathbf{w}$ a $\mathbf{b}$, které si bere jako vstupní parametry. V minulé kapitole jsme se dozvěděli, že tuto funkci potřebujeme minimalizovat → zmenšit chybu celé sítě. Zároveň jsme představili algoritmus gradient descent, který nám s minimalizací této funkce pomůže. Nicméně jak již samotný název napovídá k aplikaci tohoto algoritmu musíme mít k dispozici gradient funkce C. A ten se tedy skládá z derivací podle jednotlivých vah a biasů. **V této kapitole si představíme další algoritmus, který nám umožní tento gradient vypočítat. Jeho název je back propagation, neboli zpětná propagace.**

8.1.1 Derivace vůči váze

Teorie 8.1. Derivace složených funkcí viz 4.10



Obrázek 8.1. Znázornění závislostí pro vypočítání chybové funkce

V této sekci si společně odvodíme, jak derivovat chybovou funkci C vůči libovolné váze $w_j^{(l)}$. Zajímá nás tedy $\frac{\partial C}{\partial w_j^{(l)}}$. Pro lepší představu si opět znázorníme závislosti, jak dochází k výpočtu cyhbové funkce viz obrázek 8.1:

Z diagramu lze vyčíst, že vypočítání $\frac{\partial C}{\partial w_j^{(l)}}$ by se provedlo následujícím způsobem:

$$\frac{\partial C}{\partial w_j^{(l)}} = \frac{\partial z^{(l)}}{\partial w^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial C}{\partial a^{(l)}} \tag{1}$$

8.1.2 Derivace vůči biasu

Problém, jak získat derivaci vůči biasu je obdobný tomu, jak získat derivaci vůči váze. Budeme opět vycházet z diagramu 8.1, ze kterého jsme schopni vyčíst, že $\frac{\partial C}{\partial b_j^{(l)}}$ je následující:

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial z^{(l)}}{\partial b^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial C}{\partial a^{(l)}} \tag{2}$$

8.1.3 Praktická ukázka

Jelikož zmíněné vzorečky pro výpočet váhy, či biasu v jejich obecné formě můžou působit nejasně, tak si prakticky ukážeme jejich využití. Jak můžeme vidět, tak podoba výsledných derivací se odvíjí od tvaru chybové funkce C. Její tvar je možno si libovolně zvolit, nicméně my jsme si její tvar již stanovili viz (1) na straně 34 a budeme se jej tedy držet pouze s tím rozdílem, že se omezíme na jeden vstupní obrázek x, čímž nám odpadne sumace.

Poznámka 8.1. Při odvozování budou vynechávány spodní indexy jk u vah a biasů z důvodu úspornější notace. Bude tomu tak pouze v této sekci. Nejde tedy o vektory, ale o jednotlivé váhy, či biasy. Pokud bude obdobný krok použit i v pozdějších částech práce, bude na to patřičně upozorněno.

Příklad 8.1. Nechť C je chybová funkce. Najděme derivace pro libovolné $w_j^{(l)}$ a $b_j^{(l)}$.

$$C = \frac{1}{2}(y_j - a_j^{(L)})^2 \tag{3}$$

Poznámka 8.2. Připomeňme, že $\mathbf{a}^{(L)}$ je zkrátka výstup ze sítě, kde L je konstanta značící počet vrstev v síti. Na druhou stranu l je proměnná, za kterou lze dosadit číslo kterékoliv vrstvy v síti - a tedy klidně i L. Ve vzorečku (3) je uvedeno $a_j^{(L)}$ a to proto, že L jak již bylo zmíněno znamená poslední vrstvu a pro výpočet chyby potřebujeme výstup ze sítě porovnat s očekávaným výstupem.

Nejprve budeme hledat $\frac{\partial C}{\partial w_j^{(l)}}$. Pokud se podíváme na (1), pak vidíme, že na této derivaci se podílí následující členy: $\frac{\partial z^{(l)}}{\partial w^{(l)}}$, $\frac{\partial a^{(l)}}{\partial z^{(l)}}$ a $\frac{\partial C}{\partial a^{(l)}}$. Z vzorečku (3) již víme, jaký má tvar C a tedy pro člen $\frac{\partial C}{\partial a^{(l)}}$ nám zbývá zjistit podobu jmenovatele, tedy $a^{(l)}$. Což je dobrá zpráva, jelikož jsme si již zmiňovali vzoreček (3) na straně 28, který nám přesně toto říká.

$$a^{(l)} = \sigma(z^{(l)})$$

Již víme, že abychom získali $a^{(l)}$, pak potřebujeme $z^{(l)}$. Což opět není problém, jelikož vzoreček jsme opět zmiňovali viz (2) na straně 28.

$$z^{(l)} = w^{(l)} \cdot a^{(l-1)} + b^{(l)}$$

Zde již můžeme vidět, že $w^{(l)}$ ani $b^{(l)}$ nebudou problematické, jelikož ty máme zadány. Dále $a^{(l-1)}$ rovněž není problém, neb nás obecně zajímá $a^{(l)}$. Je tedy patrné, že se ho lze dobrat řetězovou aplikací od vstupu ($a^{(0)}$), až po $a^{(l)}$.

Se znalostí následujících členů:

$$C = \frac{1}{2}(y_j - a_j^{(L)})^2$$

$$a_j^{(l)} = \sigma(z_j^{(l)})$$

$$z_j^{(l)} = w_j^{(l)} \cdot a_j^{(l-1)} + b_j^{(l)}$$

Již víme vše potřebné k nalezení (1). Tak se do toho společně pustíme:

$$\frac{\partial C}{\partial w_j^{(l)}} = \frac{\partial z_j^{(l)}}{\partial w_j^{(l)}} \frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} \frac{\partial C}{\partial a_j^{(l)}}$$

$$\frac{\partial z_j^{(l)}}{\partial w_j^{(l)}} = a_j^{(l-1)}$$

$$\frac{\partial a_j^{(l)}}{\partial z_j^{(l)}} = \sigma'(z_j^{(l)})$$

$$\frac{\partial C}{\partial a_j^{(l)}} = (y_j - a_j^{(l)}) \cdot (-1) = (a_j^{(L)} - y_j)$$

Dáme vše dohromady a máme derivaci:

$$\frac{\partial C}{\partial w_j^{(l)}} = a_j^{(l-1)} \cdot \sigma'(z_j^{(l)}) \cdot (a_j^{(L)} - y_j)$$

Nyní pojďme najít $\frac{\partial C}{\partial b_j^{(l)}}$. Jak můžeme vidět z (2), tak derivace se liší pouze prvním členem. V tuto chvíli nám tedy stačí zjistit následující člen $\frac{\partial z^{(l)}}{\partial b^{(l)}}$:

$$\frac{\partial z^{(l)}}{\partial b^{(l)}} = 1$$

A po dosazení do (2) získáme:

$$\frac{\partial C}{\partial b_j^{(l)}} = 1 \cdot \sigma'(z_j^{(l)}) \cdot (a_j^{(L)} - y_j) = \sigma'(z_j^{(l)}) \cdot (a_j^{(L)} - y_j)$$

8.2 Hlavní vzorce pro zpětnou propagaci

Poznámka 8.3. V této sekci budou opět vynechávány spodní indexy jk u vah a biasů z důvodu úspornější notace. Nejde tedy o vektory, ale o jednotlivé váhy, biasy, atd... Pokud bude obdobný krok použit i v pozdějších částech práce, bude na to upozorněno.

Již jsme si odvodili, jakým způsobem získáme derivace podle konkérních vah, či biasů. Pojďme si je zopakovat:

$$\frac{\partial C}{\partial w_j^{(l)}} = \frac{\partial z^{(l)}}{\partial w^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial C}{\partial a^{(l)}}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \frac{\partial z^{(l)}}{\partial b^{(l)}} \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial C}{\partial a^{(l)}}$$

Pokud se podíváme pozorně, můžeme si povšimnout, že v obou vzorečcích se nám opakují poslední dva členy. Pojďme si je označit jako $\delta^{(l)}$. Pokud bychom se podívali zpět na 8.1, pak lze odvodit, že tento člen se rovná derivaci funkce C, dle $z^{(L)}$. Dle již provedených příkladů si může čtenář zkusit ověřit. Máme tedy následující rovnost:

$$\delta^{(l)} = \frac{\partial a^{(l)}}{\partial z^{(l)}} \frac{\partial C}{\partial a^{(l)}}$$

A ta nám vlastně říká, jaký vliv má vážený vstup $z^{(l)}$ na výslednou funkci C. Avšak zde je důležité zpozornět a uvědomit si, že tento vzorec není zcela obecný. Naopak, přesněji napsaný by byl takto:

$$\delta^{(L)} = \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C}{\partial a^{(L)}} \tag{4}$$

Můžeme ověřit dle 8.1, že pokud bychom chtěli odvodit $\delta^{(L-1)}$ neboli $\frac{\partial C}{\partial z^{(L-1)}}$, pak dostaneme:

$$\delta^{(L-1)} = \frac{\partial C}{\partial z^{(L-1)}} = \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C}{\partial a^{(L)}}$$

Pojďme ještě o krok dále a odvodíme si $\delta^{(L-2)}$

$$\delta^{(L-2)} = \frac{\partial C}{\partial z^{(L-2)}} = \frac{\partial a^{(L-2)}}{\partial z^{(L-2)}} \frac{\partial z^{(L-1)}}{\partial a^{(L-2)}} \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C}{\partial a^{(L)}}$$

Takto bychom mohli pokračovat až po první vrstvu. Co z toho však lze vypozorovat? Povšimněme si dvou posledních členů $\frac{\partial a^{(L)}}{\partial z^{(L)}}$ a $\frac{\partial C}{\partial a^{(L)}}$, jejichž součin se mimochodem přímo rovná $\delta^{(L)}$ dle (4). Ať už chceme počítat vrstvu kteroukoli, pak se tam vždy tyto dva členy vyskytují a vždy jsou na konci.

Pokud prozkoumáme naše odvozené vzorce dále, pak si můžeme povšimnout, že s každou další vrstvou pouze přibývá dvojice členů $\frac{\partial a^{(l-1)}}{\partial z^{(l-1)}}$, $\frac{\partial z^{(l)}}{\partial a^{(L-1)}}$, kde se samozřejmě l mění dle toho, pro kterou vrstvu počítáme.

Co si z toho tedy můžeme odnést? Že abychom byli schopni vypočítat derivaci podle jednotlivých vah, pak bychom potřebovali jít od konce vrstvu po vrstvě, jelikož jedna závisí na druhé. A vzhledem k tomu, že dle (4) již víme, jak spočítat $\delta^{(L)}$, tak již jen musíme zjistit, jak spočítat obecně $\delta^{(l)}$, která bude vycházet z $\delta^{(L)}$.

Příklad 8.2. Pojďme si odvodit vzorec pro $\delta^{(l)}$:

Pojďme vycházet z výše odvozeného výpočtu pro $\delta^{(L-1)}$:

$$\delta^{(L-1)} = \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C}{\partial a^{(L)}}$$

Již víme, že součin posledních dvou členů tvoří $\delta^{(L)}$. Proto si je nahradíme:

$$\delta^{(L-1)} = \frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \delta^{(L)}$$

Nyní jdeme rozklíčovat jednotlivé členy. Začneme členem $\frac{\partial z^{(L)}}{\partial a^{(L-1)}}$. Vidíme, že jde o derivaci váženého vstupu. Proto si připomeneme vzoreček pro jeho výpočet:

$$z^{(l)} = w^{(l)} a^{(l-1)} + b^{(l)}$$

Z toho lze vyvodit, že derivace váženého vstupu $z^{(l)}$ dle $a^{(L-1)}$ je:

$$\frac{\partial z^{(L)}}{\partial a^{(L-1)}} = w^{(l)}$$

Nyní nám již zbývá člen $\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}}$. Opět vidíme, že budeme derivovat funkci $a^{(l)}$, proto si připomeneme rovněž vzorec pro výpočet:

$$a^{(l)} = \sigma(z^{(l)})$$

A derivace pro tento konkrétní případ je tedy rovna:

$$\frac{\partial a^{(L-1)}}{\partial z^{(L-1)}} = \sigma'(z^{(l-1)})$$

Pokud nyní vše poskládáme, pak:

$$\delta^{(L-1)} = \sigma'(z^{(l-1)}) w^{(l)} \delta^{(L)}$$

Tímto jsme získali vzorec, díky kterému můžeme vypočítat $\delta^{(L)}$ s pomocí $\delta^{(L-1)}$. Nyní si jej přepíšeme zcela obecně a zároveň prohodíme indexy, abychom nalevo od rovnítka měli pouze $\delta^{(l)}$. Čtenář si může sám zkusit ověřit, proč toto platí.

$$\delta^{(l)} = \sigma'(z^{(l)}) w^{(l+1)} \delta^{(l+1)}$$

Máme tedy čtyři vzorce, díky kterým jsme schopni vypočítat derivace dle jednotlivých vah a biasů. Tentokrát se již vrátíme k vektorovému zápisu:

Teorie 8.2. $\odot$ viz 4.11

$$\delta^{(L)} = \sigma'(\mathbf{z}^{(l)}) \odot \nabla_a C \tag{5}$$

$$\delta^{(l)} = \sigma'(\mathbf{z}^{(l)}) \odot ((\mathbf{w}^{(l+1)})^T \delta^{(l+1)}) \tag{6}$$

$$\frac{\partial C}{\partial b_j^{(l)}} = \delta_j^{(l)} \tag{7}$$

$$\frac{\partial C}{\partial w_{jk}^{(l)}} = a_k^{(l-1)} \delta_j^{(l)} \tag{8}$$

Více viz [18] a [19]

Kapitola 9
Implementace neuronové sítě v jazyce Python

V tuto chvíli již známe vše potřebné z matematického hlediska pro to, abychom mohli tyto znalosti převést v plně funkční kód. Jelikož tato práce staví na výborné knize Michaela Nielsena, jak již bylo zmíněno v úvodu, tak si budeme vysvětlovat kód právě z jeho knihy.

Poznámka 9.1. Zde si projdeme kód, tak aby korespondoval s teoretickou částí. Pokud čtenáře zajímá kód blíže do detailu z pohledu programátorského, nechť se podívá do přiloženého jupyter notebooku, kde je vše vysvětleno mnohem podrobněji, či ať nahlédne do implementovaného programu, který je dodán včetně komentářů.

Nejprve si pojďme stanovit „plán útoku“:

- Načteme tréninková a testovací data
- Náhodně inicializujeme váhy a biasy, abychom měli kde začít
- Tréninková data zamícháme a rozdělíme na mini-batche
- Pro každý mini-batch vypočteme gradient a dle něj posléze upravíme váhy a biasy

Nyní již máme náš plán útoku a dle něj si společně projdeme celý kód. Pokud by se čtenář rád podíval nejprve na kód jako celek, či by ho zajímal pouze kód, pak kompletní kód s komentáři se nachází v příloze.

9.1 Načtení dat

Načtení dat bylo blíže probíráno v kapitole 5. Avšak nejedná se o téma, které je nutné k pochopení problematiky. Proto si čtenář může buď již přetransformovaná data stáhnout, či program, který transformaci provede.

Nicméně samotné načtení dat provedeme takto:

- Importujeme modul pro načtení dat.
- Načteme tréninková a testovací data.

```
import mnist
training_data, test_data = mnist.load()
```

Obrázek 9.1. Načtení dat ze serializovaného souboru

9.2 Spuštění sítě

Zde si vyzkoušíme, jak vlastně síť spustit, abychom ji viděli v akci. Až poté se vrhneme na vysvětlení samotného kódu sítě.

- Importujeme si třídu, která implementuje neuronovou síť
- Vytvoříme si novou instanci sítě s potřebnými vrstvami
- Spustíme učení pomocí Stochastic Gradient Descent metody - SGD

```
from network import Network
n = Network([784, 30, 10])
n.SGD(training_data, 30, 20, 3.0, test_data=test_data)
```

Obrázek 9.2. Spuštění tréninku sítě pomocí algoritmu stochastic gradient descent

9.3 Samotný kód neuronové sítě

Zde budeme společně procházet implementaci neuronové sítě od Michaela Nielsena.

9.3.1 Inicializace

Jako parametr pro vytvoření sítě je potřeba pouze pole, které definuje počet neuronů v jednotlivých vrstvách. Jak můžeme vidět z příkladu, kde vstupní pole je [784, 30, 10], tak vstupní vrstva obsahuje 784 neuronů, což by mělo být vzhledem k povaze vstupních dat čtenáři jasné. Druhá vrstva bude mít 30 neuronů. S touto hodnotou lze experimentovat, neboť není pevně zadána a rozdílné hodnoty mohou přinést rozdílné výsledky. A poslední vrstva obsahuje 10 neuronů, která opět vychází z povahy úkolu, který po síti chceme.

```
def __init__(self, sizes):
  self.sizes = sizes
  self.num_layers = len(sizes)
  self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
  self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
```

Obrázek 9.3. Inicializace sítě

Nyní si popíšeme, co se děje v kódu dle 9.3.

- Uložíme si informaci o velikostech jednotlivých vrstev
- Uložíme si počet vrstev
- Náhodně inicializujeme biasy
- Náhodně inicializujeme váhy

9.3.2 SGD

Nyní si společně projdeme kód pro stochastic gradient descent. Ten, jak jsme si pověděli, spočívá v tom, že si rozdělíme tréninková data na mini-batche a poté počítáme gradient vzhledem k němu. Díky tomu dojde k významnému urychlení celého tréninkového procesu.

```
def SGD(self, training_data, epochs, mini_batch_size, eta, test_data = None):
  # 1
  if test_data:
    n_test = len(test_data)

  # 2
  n = len(training_data)

  # 3
  for j in range(epochs):
      # 4
      random.shuffle(training_data)
      mini_batches = [training_data[k : k + mini_batch_size] for k in range(0, n, mini_batch_size)]

      # 5
      for mini_batch in mini_batches:
          self.update_mini_batch(mini_batch, eta)

      # 6
      if test_data:
          print("Epoch {0}: {1} / {2}".format(j, self.evaluate(test_data), n_test))
      else:
          print("Epoch {0} complete".format(j))
```

Obrázek 9.4. Stochastic gradient descent

1. Pokud jsou na vstupu testovací data, pak si uložíme kolik jsme jich obdrželi.
2. Uložíme si počet tréninkových dat.
3. Provedeme specifikovaný počet epoch.
4. Náhodně promícháme tréninková data a rozdělíme je na počet mini batchů, specifikovaný při volání funkce.
5. Pro každý mini batch provedeme aktualizaci vah a biasů dle gradientu pro tento mini bacth.
6. Pokud jsou jako parametr zadána testovací data, pak ohodnotíme přesnost sítě.

9.3.3 Update mini batch

Tato metoda má na starosti najít gradient pro daný minibatch a na jeho základě upravit váhy a biasy.

1. Nejprve si připravíme prázdné matice pro jednotlivé váhy a biasy.
2. Poté pro každý vstup v daném mini-batchi:
    1. Vypočteme všechny $\delta^{(l)}$
    2. V těchto dvou řádcích se provádí suma ze vzorce (2) a (3) na straně 42.
3. V těchto dvou řádcích se využijí získané sumy a dopočítají se již zmíněné vzorce (2) a (3) na straně 42.

```python
def update_mini_batch(self, mini_batch, eta):

    # 1
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # 2
    for x, y in mini_batch:

        # 2.1
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)

        # 2.2
        nabla_b = [nb + dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw + dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]

    # 3
    self.weights = [w - (eta / len(mini_batch)) * nw for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b - (eta / len(mini_batch)) * nb for b, nb in zip(self.biases, nabla_b)]
```

Obrázek 9.5. Update mini batch

9.3.4 Back propagation

1. Opět si připravíme prázdné matice pro váhy a biasy
2. Zde si nastavíme aktuální vrstvu aktivací jako vstup x a dále si uložíme aktivaci do pole s aktivacemi. Z kapitol, kde jsme odvozovali vzorečky je zřejmé, proč se vyplácí si uchovávat jak aktivace, tak vážené vstupy pro jednotlivé vrstvy - jsou využívány pro vypočtení gradientu.
3. Zde se jedná o klasický dopředný průchod
4. Použití vzorečku (2) ze strany 28 a uložení váženého vstupu.
5. Vzoreček (3) ze strany 28 a uložení.
6. Výpočet $\delta^{(L)}$ dle (5) strana 48.
7. Vzorec (7) strana 48.
8. Vzorec (8) strana 48.
9. Řetězově budeme počítat derivace pro jednotlivé vrstvy. Jdeme od předposlední vrstvy, jelikož až od té je to cyklický proces. Viz odvozování 8.2 strana 46
10. Vzorec (6) strana 48.
11. Vzorec (7) strana 48.
12. Vzorec (8) strana 48.
13. Vrátíme výsledné derivace

9.3.5 Pomocné funkce

Dále se v kódu nachází několik pomocných funkcí, které ovšem nejsou klíčové pro samotné pochopení algoritmu, proto zde budou vynechány. Blíže popsané samozřejmě budou v přiloženém jupyter notebooku.

```python
def backprop(self, x, y):
    # 1
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]

    # 2
    activation = x
    activations = [x]
    zs = []

    # 3
    for b, w in zip(self.biases, self.weights):
        # 4
        z = np.dot(w, activation)+b
        zs.append(z)

        # 5
        activation = sigmoid(z)
        activations.append(activation)

    # 6
    delta = self.cost_derivative(activations[-1], y) * sigmoid_prime(zs[-1])

    # 7
    nabla_b[-1] = delta

    # 8
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())

    # 9
    for l in range(2, self.num_layers):
        z = zs[-l]
        sp = sigmoid_prime(z)

        # 10
        delta = np.dot(self.weights[-l + 1].transpose(), delta) * sp

        # 11
        nabla_b[-l] = delta

        # 12
        nabla_w[-l] = np.dot(delta, activations[- l - 1].transpose())

    # 13
    return (nabla_b, nabla_w)
```

Obrázek 9.6. Back propagation

Kapitola 10
Závěr

Cílem této práce bylo vytvořit studijní materiál pro žáky středních škol, kteří mají základní znalost diferenciálního počtu a pokusit se spíše lidským přístupem problematiku přiblížit. S tím se pojilo především to, že v anglickém jazyce již existuje několik excelentních zdrojů, avšak pokud by čtenář nebyl plně sžit s anglickou četbou, pak by komplexita, již tak náročného tématu, stoupla.

Druhou částí byla konkrétní implementace neuronové sítě pro rozeznávání ručně psaných číslic ve světle získaných znalostí a ověření na testovací části dat. Celá práce vycházela z výborně zpracované knihy Michaela Nielsena[1], na které stavěla a popisovaný kód neuronové sítě je čerpán právě z této knihy. Michael Nielsen se tomuto tématu věnuje ještě hlouběji, takže pokud je čtenář ochotný zkusit studium v anglickém jazyce, je mu více než doporučeno se na zmíněnou knihu podívat.

10.1 Výukový materiál

Je počítáno, že na první průchod prací pravděpodobně u čtenáře ona „heureka“ nenastane, jelikož neuronové sítě jsou téma skutečně obsáhlé a pro nově příchozí náročné nejen na představivost, ale především na to si zvyknout na matematickou notaci, která se na první pohled může zdát komplikovaná.

Pokud čtenář ovšem vydrží a práci si v klidu projde, či se dokonce vrátí k částem, které mu nebyly úplně jasné, pak mu určitě poskytne kvalitní intuitivní základ, se kterým se bude moci vrhnout do navazujícího studia.

10.2 Implementace sítě a ověření přesnosti

10.2.1 Implementace

Čtenář si může implementaci neuronové sítě buď pouze vyzkoušet, či dokonce projít s komentáři, jelikož její implementace je jak formou obyčejné přílohy, tak formou Jupyter notebooku 1.3.4, který umožňuje ke kódu psát i přehledné poznámky a komentáře.

10.2.2 Ověření přesnosti

Ověření přesnosti probíhá na testovací části datasetu MNIST[2], který je využíván rovněž k samotnému vytrénování sítě. Jak jsme si říkali v úvodu 9, tak na počátku jsou váhy a biasy náhodně inicializovány a až poté se začne síť zlepšovat. Zároveň se pokaždé tréninková data zamíchají jiným způsobem. Z toho lze vyvodit, že při každém spuštění našeho programu, kdy se bude síť učit, budeme dostávat mírně odlišnou přesnost.

[1] http://neuralnetworksanddeeplearning.com/
[2] http://yann.lecun.com/exdb/mnist/

Níže si uvedeme dvě spuštění naší sítě s následujícími parametry:

- Síť bude mít tři vrstvy
- Vrstvy budou mít $784, 30, 10$ neuronů respektive
- Budeme síť trénovat vždy po 10 epoch
- Velikost mini-batche stanovíme na 30
- $\eta = 3.0$

Výsledek prvního chodu programu na obrázku 10.1

```
→  bachelor-thesis git:(master) ✗ python net.py

    Loading file: data/data.pkl...
    Epoch 0: 8876 / 10000
    Epoch 1: 9101 / 10000
    Epoch 2: 9208 / 10000
    Epoch 3: 9289 / 10000
    Epoch 4: 9328 / 10000
    Epoch 5: 9336 / 10000
    Epoch 6: 9369 / 10000
    Epoch 7: 9384 / 10000
    Epoch 8: 9370 / 10000
    Epoch 9: 9432 / 10000
```

Obrázek 10.1. První průchod sítí pro ověření přesnosti

Výsledek druhého chodu programu na obrázku 10.2

```
→  bachelor-thesis git:(master) ✗ python net.py

    Loading file: data/data.pkl...
    Epoch 0: 8983 / 10000
    Epoch 1: 9227 / 10000
    Epoch 2: 9287 / 10000
    Epoch 3: 9304 / 10000
    Epoch 4: 9353 / 10000
    Epoch 5: 9358 / 10000
    Epoch 6: 9381 / 10000
    Epoch 7: 9378 / 10000
    Epoch 8: 9388 / 10000
    Epoch 9: 9402 / 10000
```

Obrázek 10.2. Druhý průchod sítí pro ověření přesnosti

Data na obrázcích 10.1 a 10.2 čteme následovně:

- Epoch je číslo epochy viz 7.6.2

- První číslo je počet správně uhodnutých číslic
- Druhé číslo je celkový počet testovacích číslic

Jak můžeme vidět, přesnost našeho řešení je vynikajících $\approx 94\%$ a víme, jak celý proces funguje od prvního po poslední číslo. Samozřejmě, že dnešní sítě nabízejí mnohem přesnější klasifikaci, nezapomínejme, že se bavíme o naprosto základní podobě neuronové sítě. Dále bychom se také mohli věnovat a zkoušet vyladit η, počet skrytých vrstev a jejich velikost, atd... To všechno má na výslednou přesnost vliv. Nicméně, pokud se zamyslíme, že namísto, abychom klasickým způsobem zkoušeli počítači vysvětlit, co to vlastně je číslo například 2 a jak jej má poznat, tak se to počítač z dat naučí sám a dokonce s takto vysokou přesností, pak je patrná významnost tohoto oboru.

Další ohromnou výhodou je, že s algoritmem jako takovým není třeba nic dělat. Pokud bychom poupravili vstupní a výstupní vrstvu a korektně naformátovali trénovací a testovací data, pak se síť může naučit rozeznávat například psy od koček.

Příloha A
Program pro transformaci a načítání MNIST

Tato příloha poskytuje program pro transformaci originálních data z MNIST datasetu na námi požadovaný formát. K dispozici je několik verzí programu. Formát originálních dat je popsán v kapitole 5.

A.1 Základní verze

Tato verze se skládá z klasických souborů s pythonovským kódem. **Očekává se, že tato verze bude případně užita pro experimentování s kódem.** Součástí jsou dva soubory.

A.1.1 Mnist

Hlavním souborem je **mnist.py**, který se nachází v umístění *code/mnist.py*. Tento soubor je využíván implementací neuronové sítě k načítání dat MNIST datasetu.

Tento program se podívá, zda-li se nachází soubor s daty na cestě *data/data.pkl* (relativně od umístění programu). Pokud soubor chybí, pak provede transformaci dat. Očekává se, že soubory s originálními daty se nachází relativně od souboru na cestě *data/mnist/**. Originální soubory MNIST datasetu jsou očekávány již rozbalené, bez koncovky a s originálními názvy.

Poznámka A.1. Bližší informace ke stažení souborů s daty jsou k nalezení v souboru na relativní cestě *data/mnist/note.txt* od souboru **mnist.py**.

Cesty ke všem souborům lze samozřejmě upravit. Slouží k tomu proměnné na začátku souboru **mnist.py**. Konkrétně jde o následující proměnné: **files** a **data_file_name**.

Poznámka A.2. Pro využití tohoto modulu je potřeba jej naimportovat a poté stačí již zavolat metoda *load()*. Program po každém importu sám zkontroluje, zda-li existují transformovaná data a v případě potřeby je tranfsformuje. Ukázka, jak modul importovat a data načíst je například na obrázku 9.1.

A.1.2 Exporter

Tento program je dodáván spíše jako pomůcka pro lepší představivost. Je díky němu možné si zobrazit obrázek na daném indexu spolu s korektním popiskem (v nadpisu okna), či dokonce daný obrázek uložit jako soubor. Součástí názvu bude korektní popisek a index číslice.

V programu jsou hlavní dvě metody a to *show* a *save*. Tyto metody a jejich použití je popsané v komentářích. Nicméně možné užití obou metod by mohlo vypadat například jako na obrázku A.3.

Poznámka A.3. Pro využití tohoto modulu k zobrazení číslic je potřeba mít nainstalovaný **matplotlib**. Více informací o matplotlibu lze najít například zde [20]. Pro ukládání je potřeba modul **skimage**[1].

[1] `https://scikit-image.org/`

```
# Načtení modulu
import mnist_exporter as me

# Zobrazení obrázku na indexu 10
me.show(10)

# Uložení obrázku na indexu 10
me.save(10)
```

Obrázek A.3. Zobrazení a uložení obrázku na indexu 10

A.2 Jupyter verze

Tato verze poskytuje vysvětlení základního programu a jak funguje transformace dat. Programy pro zobrazování a ukládání jsou dodávány pouze jako doplňkové. Jelikož nejsou pro pochopení tématu kritické jejich vysvětlivky zde tedy nejsou. Více informací o Jupyteru viz 1.3.4.

Poznámka A.4. Jupyter spustíme tak, že se v terminálu přesuneme pomocí příkazu **cd** do složky na relativním umístění *jupyter* a poté zadáme **jupyter notebook** a Jupyter by se měl nastartovat a otevřít v prohlížeči. Poté otevřeme notebook s názvem **MNIST.ipynb**.

Příloha B
Implementace neuronové sítě

Tato příloha poskytuje implementaci neuronové sítě, která je schopna naučit se rozeznávat ručně psané číslice. Kód je převzat z práce Michaela Nielsena [7] a jsou k němu poskynuty vysvětlivky. K dispozici jsou dvě verze programu.

B.1 Základní verze

Tato verze je primární a je určena pro testování a experimentování s neuronovou sítí. Kód pro spuštění samotného tréninku sítě se nachází na relativní cestě *code/net_training.py*. Samotná implementace sítě se pak nachází na *code/network.py*.

B.2 Jupyter verze

Tato verze poskytuje vysvětlení základního programu neuronové sítě. Více informací o Jupyteru viz 1.3.4.

Poznámka B.5. Jupyter spustíme tak, že se v terminálu přesuneme pomocí příkazu **cd** do složky na relativním umístění *jupyter* a poté zadáme **jupyter notebook** a Jupyter by se měl nastartovat a otevřít v prohlížeči. Poté otevřeme notebook s názvem **Net.ipynb**.

Příloha C
Slovníček

AI ■ Artificial Intelligence (Umělá inteligence)
API ■ Application Programming Interface
ML ■ Machine learning (Strojové učení)
NN ■ Neural Network (Neuronová síť)

Literatura

[1] Johnson, Justin. *Python Numpy Tutorial*.
https://cs231n.github.io/python-numpy-tutorial/.

[2] Pryke, Benjamin. *Jupyter Notebook for Beginners Tutorial*.
https://www.dataquest.io/blog/jupyter-notebook-tutorial/.

[3] Kučerová, Helena. KTD: Česká terminologická databáze knihovnictví a informační vědy (TDKIV) [online]. Praha : Národní knihovna ČR.
https://aleph.nkp.cz/F/TTMNRYSXYGC5H324LIN7NNDIBMD51QJ2XJS3QTJH9J8UFCB8BU-19723?func=find-b.

[4] Co je strojové učení.
https://www.oracle.com/cz/artificial-intelligence/what-is-machine-learning.html.

[5] Helena, Kučerová. KTD: Česká terminologická databáze knihovnictví a informační vědy (TDKIV) [online]. Praha : Národní knihovna ČR.
https://aleph.nkp.cz/F/HX523GR5HUKHM76C5935MEPXJX3B63BJJUJRURR7U5YCU14J1T-05411?func=find-b.

[6] Goodfellow, Ian. *Deep Learning*. MIT Press, 2017. ISBN 9780262035613.
http://www.deeplearningbook.org/.

[7] Nielsen, Michael A. *Neural Networks and Deep Learning*.
http://neuralnetworksanddeeplearning.com/.

[8] LeCun, Yann a Corinna Cortes. MNIST handwritten digit database. 2010 .
http://yann.lecun.com/exdb/mnist/.

[9] Pandey, Pranjal. *Data Preprocessing : Concepts*.
https://towardsdatascience.com/data-preprocessing-concepts-fa946d11c825.

[10] Vydavatelství Nová média, s. r. o. *Matematické symboly*.
https://matematika.cz/symboly.

[11] Goodfellow, Ian, Yoshua Bengio a Aaron Courville. *Deep Learning*. MIT Press, 2016.
http://www.deeplearningbook.org.

[12] Petr, Olšák. *Úvod do algebry, zejména lineární*. České vysoké učení technické, 2013.

[13] Tkadlec, Josef. *Diferenciální a integrální počet funkcí jedné proměnné*. České vysoké učení technické, 2011.

[14] Hamhalter, Jan a Tišer Jaroslav. *Diferenciální počet funkcí více proměnných*. Česká technika - nakladatelství ČVUT, 2005.

[15] LeCun, Yann, Corinna Cortes a Chris Burges. MNIST Handwritten Digit Database.
http://yann.lecun.com/exdb/mnist/.

[16] Sanderson, Grant. *Gradient descent*.
https://youtu.be/IHZwWFHWa-w?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[17] *Backpropagation*.
https://brilliant.org/wiki/backpropagation/.

[18] Sanderson, Grant. *What is backpropagation really doing? — Deep learning, chapter 3*.
https://youtu.be/Ilg3gGewQ5U?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[19] Sanderson, Grant. *Backpropagation calculus — Deep learning, chapter 4*.
https://youtu.be/tIeHLnjs5U8?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

[20] Solomon, Brad. *Python Plotting With Matplotlib (Guide)*.
https://realpython.com/python-matplotlib-guide/.