Gymnázium Arabská, Praha 6, Arabská 14
Obor programování

Neuronová síť: Digitor

Ondřej Salát

Duben, 2024

Prohlašuji, že jsem jediným autorem tohoto projektu, všechny citace jsou řádně označené a všechna použitá literatura a další zdroje jsou v práci uvedené. Tímto dle zákona 121/2000 Sb. (tzv. Autorský zákon) ve znění pozdějších předpisů uděluji bezúplatně škole Gymnázium, Praha 6, Arabská 14 oprávnění k výkonu práva na rozmnožování díla (§ 13) a práva na sdělování díla veřejnosti (§ 18) na dobu časově neomezenou a bez omezení územního rozsahu.

V ..... dne ..... Ondřej Salát .....

Abstrakt

Práce se zabývá procesem tvorby programu na vytváření a trénování neuronových sítí. Popisuje princip strojového učení za použití algoritmu backpropagation. Dále se práce věnuje pokusu o hlubší pochopení neuronových sítí. Konkrétně se věnuje různým architekturám neuronové sítě pro rozpoznávání rukou napsaných cifer.

Obsah

1 Úvod

Cílem maturitní práce je vytvořit neuronovou síť a naprogramovat k ní základní učící algoritmus bez použití knihoven pro strojové učení (jako TensorFlow, nebo PyTorch). Záměrem je hlubší pochopení a prozkoumání různých implementací neuronových sítí. Neuronová síť Digitor bude natrénována pro rozpoznávání rukou napsaných číslic [0-9]. Jako učící data jsem použil databázi MNIST[1]. Síť dostane na vstup čtvercový obrázek a výstupem bude číslice obsažena v obrázku.

1.1 Výběr tématu

Toto téma jsem si vybral z důvodu, že si myslím, že neuronové sítě jsou velice zajímavé a dosud ne úplně prozkoumané téma. Věřil jsem, že tvorbou tohoto projektu se naučím jak neuronové sítě fungují. Zároveň bych na toto téma rád navázal při studiu na vyskoké škole.

1.2 Použité technologie

Hlavní část tohoto projektu jsem naprogramoval v jazyce C++. Jazyk jsem vybral z důvodu, že je velice rychlý a to za cenu, že uživatelský komfort je horší než například při použití Pythonu. Jak už jsem zmiňoval, pro tento projekt byla nejdůležitější rychlost, a proto jsem vybíral mezi jazykem C a C++. Nakonec jsem zvolil C++ z důvodu, že i přes svoji rychlost a relativně velkou kontrolou je objektově orientován a obecně uživatelsky přívětivější. Konkrétně jsem použil verzi C++23.

Dále jsem použil nástroj CMake, který slouží k buildování C++ kódu. CMake umožňuje zjednodušení celého buildovacího procesu a dokáže zajistit přenositelnost.

Pro práci s formátem JSON jsem použil C++ knihovnu ‘nlohmann/json‘[2]. Knihovna zjednodušuje práci s formátem JSON, který je v tomto projektu využit pro ukládání natrénovaných modelů.

Při práci s obrázky jsem použil jazyk Python. V tomto jazyce jsem napsal různé scripty, které dokáží načíst obrázky ze souboru a převést je do formátu, se kterým dokáže neuronová síť dále pracovat.

2 Neuronová síť

Neuronová síť je výpočetní model, který je inspirován fungováním lidského mozku. Neuronová sít se skládá z tzv. umělých neuronů a vah. Neurony jsou propojeny váhami mezi vrstvami.

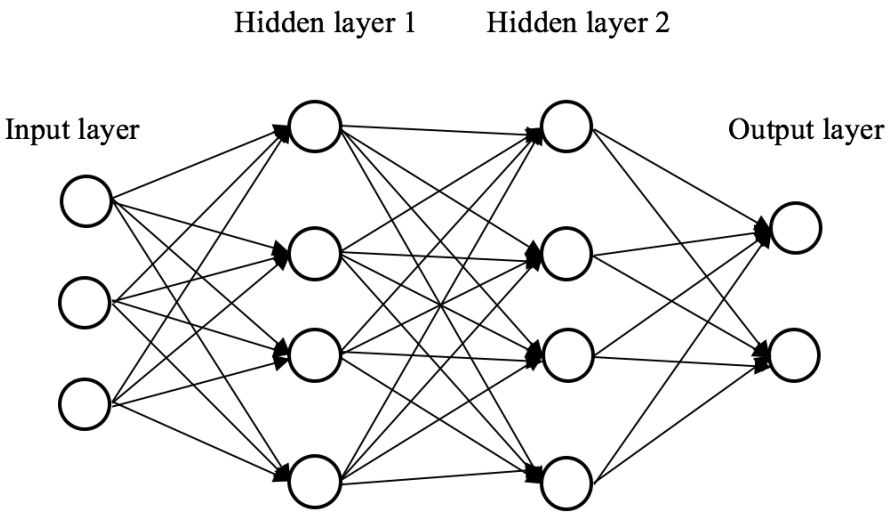
Struktura neuronové sítě:

- vstupní vrstva - zde se zadají vstupní data
- skryté vrstvy - tyto vrstvy transofrmují data a napomáhají řešit komplexní úlohy
- výstupní vrstva - vrací výsledky zpracování dat

Počty neuronů v každé vrstvě závisí na konkrétním využití a na problému, který má neuronová síť řešit.

Každý neuron představuje jednoduchý výpočetní prvek, který zpracovává vstupy a generuje výstup. Vstupy a výstupy jsou spojeny váhami, které ovlivňují hodnoty neuronů, ze kterých váha vede. Dále má každý neuron svůj bias, který ovlivňuje jeho hodnotu.

Existují různé typy neuronových sítí, které se liší tím, jak jsou neurony mezi vrstvami propojeny. Různé typy propojení jsou vhodné pro různé typy úloh. Mezi nejběžnější typy patří neuronové sítě typu perceptrony, vícevrstvé perceptory, konvulační neuronové sítě a rekurentní neuronové sítě.



Obrázek 1: Neuronová síť
[3]

Mezi hlavní výhody neuronových sítí patří schopnost učit se z dat a adaptovat se novým informacím. Neuronové sítě dokáží řešit komplexní úlohy, které jsou extrémně náročné pro běžné algoritmy. Tyto sítě jsou také vhodné pro práci s velkými objemy dat.

2.1 Fungování neuronových sítí

Neuronová síť načte vstup a postupně se počítají hodnoty neuronů po vrstvách. Po průchodu hodnot celou sítí bude ve výstupní vrstvě výsledek.

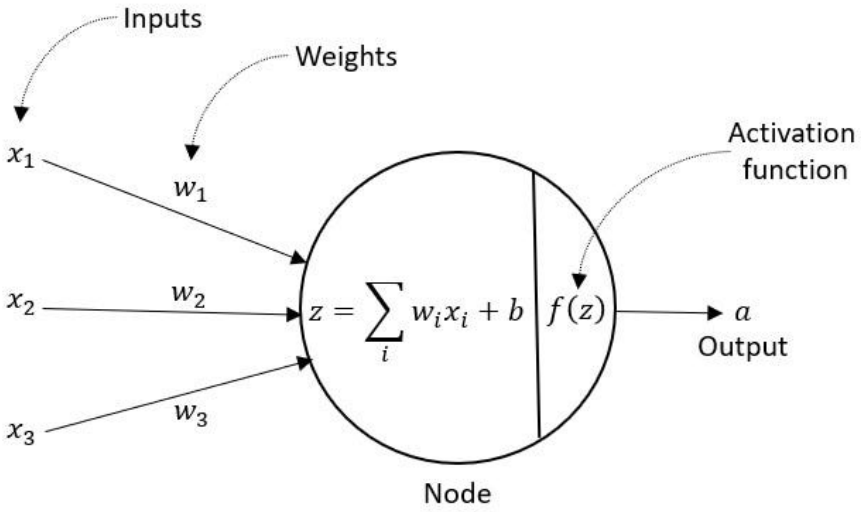
Hodnota konkrétního neuronu se spočítá z jeho vstupů. Každý neuron, až na vstupní vrstvu, má vstupní neurony. Jeho hodnota se vypočítá tak, že se sečte hodnota všech jeho vstupních neuronů vynásobena spojujícími vahami každého neuronu. Dále se přičte bias daného neuronu, který slouží jako práh aktivace. Nakonec se použije aktivační funkce. Aktivačních funkcí je řada. Mezi nejběžnější patří:

$$Sigmoid \quad \sigma(x) = \frac{1}{1 + e^{-x}}$$

$$ReLU \quad ReLU(x) = max(0, x)$$

$$Tanh \quad tanh(x) = \frac{e^{2x} - 1}{e^{2x} + 1}$$

Výběr ideální aktivační funkce opět zavisí na problému, který má síť řešit. Tyto funkce v jednoduchosti transformují vstupy neuronu na jeho výstupy. Aktivační funkce dodávají sítím jistou nelinearitu, která umožňuje sítím řešit i problémy, které nejsou linearní.



Obrázek 2: Umělý neuron
[4]

3 Strojové učení

Strojové učení je nedílnou součástí celé kapitoly neuronových sítí. Strojové učení umožňuje sítím se učit ze známých dat. Díky strojovému učení je možné řešit problémy, které byly do objevení těchto učících technik neřešitelné. Existuje několik způsobů učení, ale v této práci bylo použito takzvané učení s učitelem. To znamená, že všechna data, ze kterých se síť učí, jsou předkládána síti se správným výsledkem. Existují však metody bez učitele, které dokáží síť učit i z nepopsaných dat.

Princip storjového učení je, že síť dostane data se správnými výsledky a na základě těchto dat si poupraví svoje hodnoty vah a biasů tak, aby při načtení těchto dat odpověděla příště správně.

3.1 Zpětné počítání chyby

Zpětné počítání chyby[5], neboli backpropagation, je algoritmus pro hluboké učení neuronových sítí. Tento algoritmus funguje tak, že se šíří chyba výstupu zpátky do neuronové sítě a podle chyby upravuje postupně váhy a biasy.

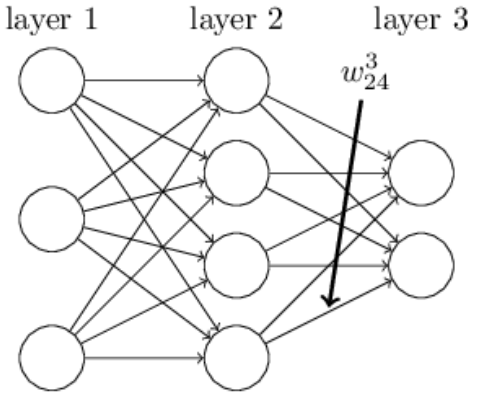
Tento algoritmus se skládá ze 4 kroků. Nejprve se musí vyhodnotit chyba. To znamená, že se konkrétní data nechají vyhodnotit sítí a spočítá se chyba. Chyba je v tomto případě rozdíl od očekávaného výsledku. Dále se chyba šíří zpět do sítě a počítá se derivace chyby vzhledem ke konkrétním vahám a biasům. Na základě této derivace se váhy a biasy aktualizují. Tato aktualizace pomáhá zmírňovat celkovou chybu výsledku. Nakonec se tento proces pro všechny trénovací data opakuje.

3.2 Počítání chyby a aktualizace vah a biasů

Celková chyba sítě se označuje velkým písmenem C. Při zpětné úpravě vah a biasů nás zajímá jaká je derivace chyby vůči každé váze $\frac{\delta C}{\delta w}$ a biasu $\frac{\delta C}{\delta b}$. V moment, kdy budeme znát tento vztah, tak víme jakým směrem upravit váhu nebo bias, abychom snížili celkovou chybu sítě. Celková chyba sítě pro jeden konkrétní trénovací příklad se spočítá jako součet očekávaný výsledek všech neuronů ve výstupní vrstvě mínus jejich realný výsledek $C_0 = \sum(target_i - output_i)^2$. Tím pádem celková chyba pro všechny trénovací data je $C = \frac{1}{n}\sum(target_i - output_i)^2$.

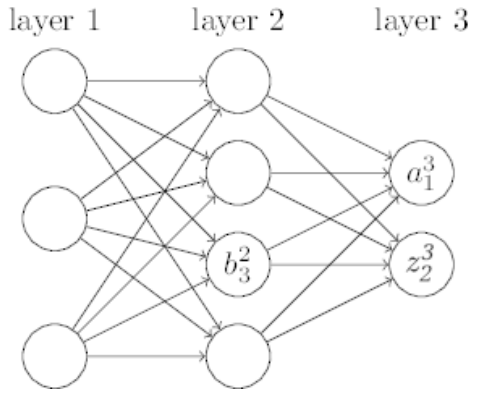
Důležitá část pro pochopení vzorce algoritmu zpětného počítání chyby je notace zápisu neuronové sítě. Proto je potřeba nejprve definovat zápis a až dále se budu věnovat samotným vzorcům a vztahům při zpětném počítání chyby.

Pro popsání váhy budem zapisovat jako w^l_{jk}, kde l je číslo vrstvy, j je označení, do kterého neuronu v l-té vrstvě váha směřuje a číslo k označuje z kolikátého neuronu z vrstvy $(l-1)$ váha vychází.



Obrázek 3: Příklad zápisu váhy
[6]

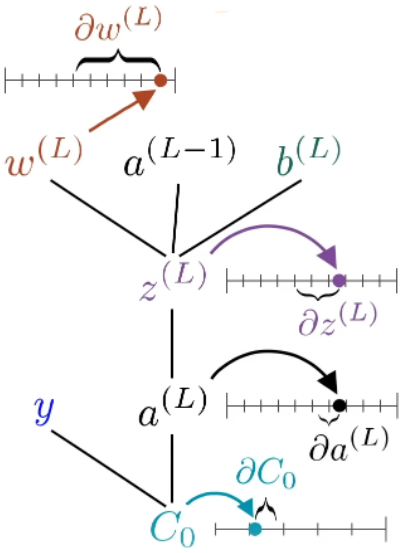
Podobná notace se použije také při popisu biasu, neaktivované a aktivované hodnoty neuronu. Bias neuronu se zapíše jako b^l_j, kde l je vrstva neuronu a j je j-tý neuron v l-té vrstvě. Hodnota neuronu před aktivací je označí písmenem $z^l_j = \left(\sum(w^l_{jk} \cdot a^{l-1}_k) + b^l_j\right)$. Hodnota z^l_j je tedy součet součinů neuronů z předešlé vrsty a příslušných vah. Hodnota aktivovaného neuronu ze zapíše jako $a^l_j = f(z^l_j)$, kde funkce $f(x)$ je nějaké aktivační funkce, jako například funkce sigmoid $\sigma(x)$.



Obrázek 4: Příklad zápisu biasu, aktivace a neaktivace neuronu
[7]

Pokud máme upřesněnu notaci, můžeme se posunout k samotnému algoritmu počítání chyby. Základem, jak už bylo zmíněno, je poměr $\frac{\delta C}{\delta w}$. Tento vztah je pro výpočet klíčový. Tento vztah nám v podstatě říká, jak se změní celková chyba při změně w.

Nejprve si ukážeme, pro jednoduchost, jak funguje učení neuronové sítě pouze s jedním neuronem na vrstvu. Jak je z obrázku vidět, když změníme hodnotu $w^{(L)}$, tak to ovlivní hodnotu $z^{(L)}$) a to zas nějak ovlivní $a^{(L)}$ a to pak přímo ovlivní C_0. Existuje tedy pravidlo, řetězové pravidlo[9], které říká, že $\frac{\delta C_0}{\delta w^{(L)}} = \frac{\delta z^{(L)})}{\delta w^{(L)}} \cdot \frac{\delta a^{(L)}}{\delta z^{(L)}} \cdot \frac{\delta C_0}{\delta a^{(L)}}$. Toto pravidlo tedy říká, že pokud nějaký



Obrázek 5: Změny
[8]

prvek nepřímo ovlivňuje poslední prvek, tak se ta změna rovná součinu změně přímo po sobě jdoucích členů.

Jakmile máme tento vztah, je potřeba spočítat derivaci čitatele vůči jmenovateli.

Protože $C_0 = (a^{(L)} - y)^2$, tak potom bude derivace $\frac{\delta C_0}{\delta a^{(L)})} = 2(a^{(L)} - y)$. Parcialní derivace $\frac{\delta a^{(L)}}{\delta z^{(L)}} = activation'(z^{(L)})$. Nakonec parcialní derivace $\frac{\delta z^{(L)}}{\delta w^{(L)}} = a^{(L-1)}$. Z toho vyplývá, že $\frac{\delta C_0}{\delta w^{(L)}} = a^{(L-1)} \cdot activation'(z^{(L)}) \cdot 2(a^{(L)} - y)$

Už zbývá spočítat chybu vůči vahám, aby byl algoritmus zpětného počítání chyby hotový. Naštěstí se počítání moc neliší od počítání chyby vůči vahám. Vzorec pro spočítání zůstane skoro stejný, kde $\frac{\delta C_0}{b_j^{(L)}} = \frac{\delta z^{(L)})}{\delta b^{(L)}} \cdot \frac{\delta a^{(L)}}{\delta z^{(L)}} \cdot \frac{\delta C_0}{\delta a^{(L)}}$. Derivace $\frac{\delta z^{(L)}}{\delta b^{(L)}} = 1$ a zbytek derivací zůstane stejný. Tedy $\frac{\delta C_0}{b_j^{(L)}} = activation'(z^{(L)}) \cdot 2(a^{(L)} - y)$.

Jakmile máme odvozený vztah pro $\frac{\delta C_0}{\delta w^{(L)}}$, můžeme odvodit pravidlo rozšířit, aby fungovalo obevně pro všechny neuronové sítě. Tím pádem nás zajímá $\frac{\delta C_0}{\delta w_{jk}^{(L)}}$. Prvním rozdílem je, že $C_0 = \sum_{j=1}^{n_L}(a_j^{(L)} - y_j)^2$. Druhým rozdílem je, že $\frac{\delta C_0}{\delta a_j^{(L-1)}}$ se bude počítat trochu složitěji. Protože neuron $n_j^{(L-1)}$ ovlivňuje chybu skrz několik cest. Tím pádem $\frac{\delta C_0}{\delta a_j^{(L-1)}} = \sum_{j=1}^{n_L} \frac{\delta z_j^{(L)}}{\delta a_k^{(L-1)}} \cdot \frac{\delta a_j^{(L)}}{\delta z_j^{(L)}} \cdot \frac{\delta C_0}{\delta a_j^(L)}$. Z toho vyplývá, že $\frac{\delta C_0}{\delta a_j^{(L-2)}} = \sum_{j=1}^{n_L} \frac{\delta C_0}{\delta a_j^{(L-1)}}$ a $\frac{\delta C_0}{\delta a_j^{(L)}} = \sum_{j=1}^{n_L} 2(a_j^{(L)-y_j})$.

To samé platí i pro výpočet chyby vůči biasu. Tím pádem vzorec zůstane stejný, jen s tou změnou, že $\frac{\delta C_0}{\delta a_j^{(L-2)}} = \sum_{j=1}^{n_L} \frac{\delta C_0}{\delta a_j^{(L-1)}}$. V tuto chvíli už dokážeme spočítat chybu vůči všem vahám i biasům v síti, a tudíž upravovat jejich hodnoty k lepšímu.

4 Implementace v programu

Program slouží k vytváření a trénování neuronových sítí podle vstupních parametrů. Samotný program se dá spustit s přepínači, které určují chování programu. Program je schopen na požadavek buď vytvořit novou síť podle vstupních parametrů, nebo načíst existující síť ze souboru JSON. S takto vytvořenou neuronovou sítí je program dále schopen pracovat dvěma způsoby. Neuronovou síť může trénovat nebo jí jen načte a čeká na vstupní data, která neuronová síť zpracuje a vrátí výsledek.

4.1 Vytváření a načítání neuronové sítě

Neuronová síť je v programu reprezentována vícerozměrnými poli. První dvojrozměrné pole reprezentuje neurony.

Pole *neuron* má dva rozměry tzn. pole v poli. První (nulté) pole v *neuron* reprezentuje první vrstvu neuronů (vstupní) vrstvu. Naopak poslední prvek pole *neuron* reprezentuje poslední (výstupní) vrstvu. Počet skrytých vrstev v neuronové síti se tedy rovná $neuron.size() - 2$. Každé toto pole uložené v *neuron* obsahuje n hodnot, které reprezentují hodnoty konkrétních neuronů. Hodnoty jsou typu long double.

Dále je potřeba pole $rawNeuron$, které je úplně stejné jako pole *neuron*, ale jsou do něj ukládány neaktivované hodnoty neuronů.

Další pole *bias* má také dva rozměry a jeho velikost je identická jako velikost pole *neuron*. Hodnoty v jednotlivých polích reprezentují hodnotu biasu konkrétního neuronu. Hodnoty jsou také typu long double.

Poslední pole reprezentující samotnou neuronovou síť je trojrozměrné pole *weight*. Velikost pole je rovno počtu vrstev mínus jedna. První (nultý) prvek tj. $weight[0]$ reprezentuje dvojrozměrné pole, které reprezentuje váhy spojující vstupní neurony s neurony druhé vrstvy. Pole $weight[0][0]$ je pole hodnot jednotlivých vah, které míří do prvních (nultého) neuronu druhé vrstvy. Váhy jsou také typu long double.

Při vytváření nové neuronové sítě se v konstruktoru vytvoří zmiňovaná pole o velikostech, které odpovídají zadaným parametrům. Hodnoty vah a biasů se incializují s náhodnou hodnotou. Dále se celá neuronová síť serializuje do souboru JSON, který se následně uloží. V názvu souboru je obsažen počet vrstev sítě navíc s náhodným číslem, které se pokusí zabránit kolizi na disku. Soubor obsahuje všechna potřebná data pro pozdější načtení neuronové sítě. To znamená velikost, aktivační funkce neuronů, hodnoty vah a hodnoty biasů.

Pokud program pouze načítá neuronovou síť z již existujícího souboru, konstruktor pouze očekává název souboru. Dále konstruktor deserializuje data ze souboru a uloží si je do proměnných.

4.2 Trénování neuronové sítě

Jak už bylo zmíněno v kapitole o strojovém učení, proces učení neuronové sítě se skládá ze čtyř částí. Tedy průchod dat sítí a vyhodnocení chyby, zpětné počítání chyby, aktualizace vah a biasů a nakonec se vše zopakuje. Průchod dat sítí neboli přímý průchod je vyřešen jako tři vnořené for cykly. Hodnota každého neuronu se vypočítá podle vzorce na výpočet hodnoty neuronu, a tudíž $z_j^l = \left(\sum(w_{jk}^l \cdot a_k^{l-1}) + b_j^l\right)$. Z důvodu, že při zpětném počítání chyb je potřeba aktivovaný i neaktivovaný neuron, tak se do paměti uloží obě hodnoty (do pole *neuron* a $rawNeuron$). Pro samotné učení je v programu metoda *train*, která všechny tyto kroky provede.

Při volání metody *train*, funkce očekává dvojrozměrné pole typy $TrainData$, které obsahuje vstupní hodnoty a správný výsledek průchodu, počet trénovacích cyklů (iterací) a jako poslední parametru očekává tzv. rychlost učení (learning rate). Learning rate určuje míru změny při aktualizaci vah a biasů. Hodnota rychlosti učení se běžně pohubuje v rozmezí od 0.1 do 0.0001.

Pole *TrainData* je dvojrozměrné z důvodu, že data jsou rozdělena do tzv. seté, které zajišťují rychlejší a univerzálnější učení. Tím pádem metoda *train* funguje tak, že zavolá metodu *backpropagate* na všechny členy setu. Aktualizované hodnoty vah a bíasů se ukládají do externího pole a skutečné váhy a biasy neuronové sítě se aktualizují až po dokončení. Takto se postupně trénuje síť pomocí všech setů dat. Následně se celý tento proces opakuje podle počtu iterací. Metoda v průběhu počítá průměrnou chybu pro vstupní data, kterou vypisuje na standardní výstup společně s progresem trénování (procento vykonaných iterací).

Metoda *backpropagate* funguje na principu popsaném v kapitole 3 Strojové učení. Funkce je rozdělena do dvou částí. První část spočíta chybu pro poslední vrstvu, která se počítá jednodušeji, protože neurony ovlivňují chybu pouze jednou cestou. Druhá část naváže na první a počítá zpětně chybu pro zbytek vrstev. Pro výpočty jsou použity vzorce z kapitoly 3 o strojovém učení.

4.3 Spouštění programu

Při spouštění si pomocí přepínačů lze nastavit, zda chceme například vytvářet úplně novou neuronovou síť, nebo chceme načíst již exístující síť. Dále je možné specifikovat jestli chceme síť trénovat nebo ji chceme použít k zpracování dat.

Možné použití jsou:

- Načte neuronovou síť ze souboru a čeká na standartním vstupu data na zpracování.

```
$ ./digitor <jmeno_souboru>
```

- Vytvoří neuronovou síť podle zadaných parametrů. Formát pro zadání neuronů je vždy počet neuronů pro každou vrstvu, který je oddělen čárkou. Př. "10,2,2,10", tato síť by měla čtyři vrstvy, z čehož vstupní i výstupní by měla 10 neuronů a obě skryté by měly 2 neurony.

```
$ ./digitor -n <neurony> <aktivace>
```

- Trénování exitující sítě. Program dostane informace o počtu učících dat, počtu iterací a míře učení. Dále program čeká na standartní vstup zmiňované učící data a následovně síť trénuje.

```
$ ./digitor -t <jmeno_soubor> <pocet_iteraci> <rychlost_uceni>
  <pocet_davek> <pocet_dat>
```

- Vytvoří novou neuronovou síť podle požadavků a síť natrénuje stejným způsobem jako v předešlém bodě.

```
$ ./digitor -t -n <neurony> <iterace> <rychlost_uceni>
  <aktivace> <pocet_davek> <pocet_dat>
```

5 Zkoumání architektur

Při vytváření sítě na rozpoznávání rukou napsaných číslic jsem narazil na problém, že nevím, jak velkou síť mám použít. Při hledání na internetu jsem zjistil, že vlastně neexistuje žádná perfektní síť, a že ideální velikost sítě vždy závisí na problému, který má řešit. Z toho důvodu jsem se rozhodl, že vytvořím malý výzkum na to, které sítě mají nejlepší výsledky. Vytvořil jsem proto osm různých neuronových sítí, kterým jsem dal úplně stejný učící set a stejný počet iterací na natrénování. Sítě měly vždy stejný počet vstupních a výstupních neuronů pouze se lišily v počtu skrytých vrstev a v počtu neuronů v těchto vrstvách. Použité architektury byly následovné:

- 2 vrstvy po 64 neuronech
- 3 vrstvy po 16 neuronech
- 3 vrstvy po 24 neuronech
- 3 vrstvy po 32 neuronech
- 3 vrstvy po 40 neuronech
- 3 vrstvy po 48 neuronech
- 3 vrstvy po 56 neuronech
- 3 vrstvy po 64 neuronech

Tyto konkrétní architektury jsem zvolil z důvodu, že se mi ještě před uskutečněním tohoto zkoumání podařilo vytvořit model, který měl 3 skryté vrstvy po 64 neuronech, a který dokázal rozpoznat obrázky s vysokou přesností. Proto mě napadlo zkusit, kolik neuronů je potřeba, aby neuronová síť měla uspokojivé výsledky. Zvolil jsme tedy neuronové sítě, které mají postupně nižší počet neuronů a zkoumal jsem, jaký to má vliv na výsledky.

Vytvořil jsem tedy 8 těchto sítí a spustil jsme trénování. Všechny sítě dostaly identická data a identický počet iterací. Zároveň jsem měřil čas, za jak dlouho se síť s těmito parametry natrénuje. Sítě byly tedy natrénovány pomocí 10000 obrázků (1000 z od každé číslice). Po ukončení trénování jsem sítě otestoval pomocí 27730 vyčleněných obrázků z databáze MNIST[1], které sítě nikdy neviděly.

Architektura	Čas učení	Přesnost
2 vrstvy 64 neuronů	20h 5m 31s	53.0%
3 vrstvy 16 neuronů	5h 46m 27s	24.7%
3 vrstvy 24 neuronů	8h 30m 19s	34.6%
3 vrstvy 32 neuronů	10h 42m 14s	54.5%
3 vrstvy 40 neuronů	13h 49m 14s	50.9%
3 vrstvy 48 neuronů	16h 48m 6s	57.9%
3 vrstvy 56 neuronů	19h 54m 12s	55.2%
3 vrstvy 64 neuronů	23h 25m 54s	58.4%

Z výsledků je zřejmé, že sítě s vyšším počtem neuronů mají v tomto případě lepší výsledky, ale trénování takových sítí trvá déle. V tomto případě jsem došel k závěru, že rozdíl mezi trénováním menších a větších sítí není tak markantní, a tudíž se v tomto konkrétním připadě vyplatí použít sítě s větším počtem vrstev a neuronů. Největší síť se dokázala vytrénovat v pozdějším trénování vytrénovat natolik, že dokázala poznat více než 98% obrázků, které nikdy neviděla.

6 Závěr

Při vytváření tohoto projektu jsem se ponořil do problematiky neuronových sítí a zjistil jsem, na jakém principu funguje strojové učení pomocí zpětného počítaní chyby. Tento projekt mi pomohl vytvořit si představu, jak neuronové sítě fungují a k čemu vůbec mohou sloužit.

6.1 Prostor pro zlepšení

Při trénování modelu by se měla použít metoda špatných dat. To znamená, že se při trénování dávají modelu nejen správná data, která se model má naučit, ale i chybná data, která má síť vyhodnodit jako chybná. Tím se zabrání tomu, že model vyhodnotí úplně špatná data jako správná. To jsem při trénování hlavního modelu na rozpoznávání číslic nepoužil. Zaprvé z důvodu, že jsem neměl přístup k velkému objemu chybných dat, zadruhé proto, že o existenci této metody jsem se dozvěděl až po ukončení trénování modelu. Při trénování dalších modelů bych tuto metodu určitě implementoval.

V budoucnu bych také chtěl v programu implementovat parelelní výpočet pomocí CUDA. Díky tomu by se mohlo trénování sítí řádově urychlit. Dále bych chtěl v budoucnu přidat možnost pro vytváření jiných typů neuronových sítí, než jen plně propojených.

Odkazy

[1] *Databáze MNIST*. URL: https://yann.lecun.com/exdb/mnist.

[2] *Knihovna pro práci s JSON*. URL: https://github.com/nlohmann/json.

[3] *Neuronová síť, obrázek*. URL: https://www.oreilly.com/api/v2/epubs/9781838642709/files/assets/61bc8450-f3ac-4d81-b405-3d748e30d04a.png.

[4] *Umelý neuron, obrázek*. URL: https://miro.medium.com/v2/resize:fit:640/1*sPg-0hha7o3iNPjY4n-vow.jpeg.

[5] *Zpětné počítání chyby*. URL: http://neuralnetworksanddeeplearning.com/chap2.html.

[6] *Váha v síti, obrázek*. URL: http://neuralnetworksanddeeplearning.com/images/tikz16.png.

[7] *Bias a neuron, obrázek*. URL: http://neuralnetworksanddeeplearning.com/images/tikz17.png.

[8] *Řetězové pravidlo, obrázek*. URL: https://youtu.be/tIeHLnjs5U8?si=IN_ja7N9fvgSU_PO&t=232.

[9] *Řetězové pravidlo*. URL: https://en.wikipedia.org/wiki/Chain_rule.

Seznam obrázků