Fakulta elektrotechnická
Katedra aplikované elektroniky a telekomunikací

Bakalářská práce

Realizace neuronové sítě s využitím grafických procesorů

Autor práce: Martin Farkaš
Vedoucí práce: Ing. Jan Broulím

Plzeň 2019

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta elektrotechnická
Akademický rok: **2018/2019**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Martin FARKAŠ**
Osobní číslo:	**E16B0071P**
Studijní program:	**B2612 Elektrotechnika a informatika**
Studijní obor:	**Elektronika a telekomunikace**
Název tématu:	**Realizace neuronové sítě s využitím grafických procesorů**
Zadávající katedra:	**Katedra aplikované elektroniky a telekomunikací**

Zásady pro vypracování:

1. Sestavte přehled v současnosti využívaných neuronových sítí.
2. Vybranou neuronovou síť realizujte na CPU a GPU.
3. Zvolte vhodnou trénovací množinu.
4. Porovnejte rychlosti aplikací.

Rozsah grafických prací: **podle doporučení vedoucího**

Rozsah kvalifikační práce: **30 - 40 stran**

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

1. **Principe J., C.: "Artificial Neural Networks" The Electrical Engineering Handbook. CRC Press LLC, 2000**
2. **Sanders J.; Kandrot, E.: "CUDA by Example" NVIDIA Corporation, 2011**

Vedoucí bakalářské práce: **Ing. Jan Broulím**
Regionální inovační centrum elektrotechniky

Datum zadání bakalářské práce: **5. října 2018**

Termín odevzdání bakalářské práce: **13. června 2019**

Prof. Ing. Zdeněk Peroutka, Ph.D.
děkan

L.S.

Doc. Dr. Ing. Vjačeslav Georgiev
vedoucí katedry

V Plzni dne 5. října 2018

Abstrakt

Tato práce se věnuje umělým neuronovým sítím a rychlosti jejich trénování. Teoretická část bakalářské práce popisuje historii umělých neuronových sítí, výhody grafických procesorů pro neuronové sítě, vybrané optimalizační algoritmy a strukturu konvolučních neuronových sítí. Sestavení neuronové sítě je řešeno použitím programovacího jazyku Python 3.7 s užitím knihoven Tensorflow a Keras. Porovnání rychlostí trénování je provedeno na trénovacích množinách MNIST a CIFAR-10. Získané výsledky ukazují výrazné zrychlení trénování pomocí GPU, i na jednoduchých problémech.

Klíčová slova

umělé neuronové sítě, konvoluční neuronové sítě, hluboké učení, grafický procesor

Abstract

Farkaš, Martin. *Implementation of neural network using graphics processing units* [*Realizace neuronové sítě s využitím grafických procesorů*]. Pilsen, 2019. Bachelor thesis (in Czech). University of West Bohemia. Faculty of Electrical Engineering. Department of Applied Electronics and Telecommunications. Supervisor: Jan Broulím

---

This thesis deals with artificial neural networks and the speed of their training. The theoretical part of the thesis describes the history of artificial neural networks, the advantages of graphical processors for neural networks, selected optimization algorithms and the structure of convolutional neural networks. The neural network is constructed using the Python 3.7 programming language, using the Tensorflow and Keras libraries. Training speed comparisons are made on MNIST and CIFAR-10 datasets. The obtained results show a significant acceleration of neural network training using GPU, even on simple problems.

Keywords

artificial neural networks, convolutional neural network, deep learning, graphic processing unit

Prohlášení

Předkládám tímto k posouzení a obhajobě bakalářskou práci, zpracovanou na závěr studia na Fakultě elektrotechnické Západočeské univerzity v Plzni.

Prohlašuji, že jsem svou závěrečnou práci vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 270 trestního zákona č. 40/2009 Sb.

Také prohlašuji, že veškerý software, použitý při řešení této bakalářské práce, je legální.

V Plzni dne 13. června 2019

Martin Farkaš

.....

Podpis

Obsah

Seznam obrázků

Seznam tabulek

Seznam symbolů a zkratek

ANN	Artifical Neural Network. Umělá neuronová síť.
MLP	Multi-Layer Perceptron. Vícevrstvý Perceptron.
CNN	Convolutional Neural Network. Konvoluční neuronová síť.
GPU	Graphics processing unit. Grafický procesor.
CPU	Central processing unit. Centrální procesorová jednotka
MSE	Mean Squared Error. Střední kvadratická chyba

1

Úvod

Neuronové sítě se v posledních letech staly velmi populární, a to díky přístupu k výkonnější výpočetní technice, rozsáhlejším datasetům a lepším technikám pro učení hlubokých sítí. Jedná se o lukrativní a stále se vyvíjející oblast, která se postupem času užívá ve větším počtu odvětví. Neuronové sítě jsou v dnešní době využívány mnoha předními technologickými společnostmi, jako je Google, Microsoft, Apple, NVIDIA, Facebook. Používají se k rozpoznávání řeči, rozpoznávání obličejů, doporučování videí, k analýze zvuku a jsou základním stavebním prvkem pro autonomní řízení aut.

Pro řešení složitých problémů musí mít neuronová síť velké množství parametrů. Počet parametrů se v dnešní době pohybuje v řádu 10^7. Pro efektivní trénování takto hlubokých neuronových sítí, musíme použít výkonný hardware a optimalizovaný software. V dnešní době se trénování těchto sítí provádí výhradně na GPU nebo na více GPU paralelně.

Důvodem řešení bakalářské práce je rozvíjení se potenciálních aplikací pro neuronové sítě a testování Govorun clusteru. Cílem této bakalářské práce je implementace neuronové sítě na CPU a GPU, a porovnat rychlost trénování. Realizace je provedena v programovacím jazyku Python 3.7 s využitím knihoven TensorFlow a Keras. Porovnání rychlosti bylo provedeno na MNIST a CIFAR-10 datasetech. Jako svůj cíl jsem si stanovil napsání knihovny pro C#, která by podporovala trénování neuronových sítí na GPU.

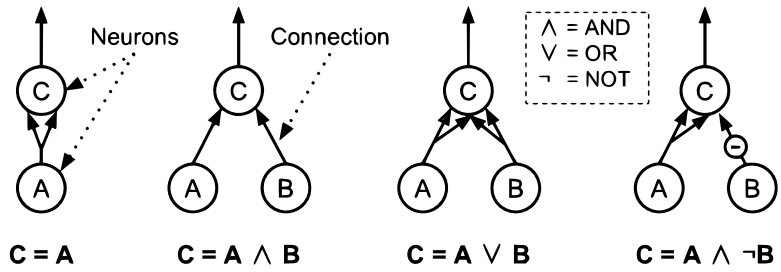
2

Umělé neuronové sítě

Myšlenku umělých neuronových sítí (angl. artificial neural networks) poprvé popsaly v roce 1943 neurofyziolog Warren McCulloch a matematik Walter Pitts v publikaci [9]. McCulloch a Pitts v ní představili zjednodušený model biologického neuronu, který se později začal nazývat umělý neuron. Umělý neuron má jeden nebo více binárních vstupů n a jeden binární výstup y. Výstup se aktivuje pokud součet vstupů z je větší, než daný práh (angl. threshold) T.

$$y = \begin{cases} 1 & z \geq T \\ 0 & z < T \end{cases} \tag{2.1}$$

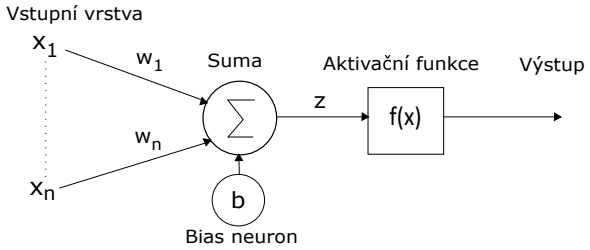
McCulloch a Pitts ukázali, že s tímto jednoduchým modelem je možné sestrojit logické funkce OR, AND, NOT (obr. 2.1)



Obr. 2.1: Jednoduché logické funkce pro práh aktivace T=2 |Převzato z [3]|

2.1 Perceptron

Perceptron je jedna z nejjednodušších umělých neuronových sítí. Poprvé ji představil Frank Rosenblatt v roce 1957 [10]. Perceptron je složen ze vstupní a výstupní vrstvy. Vstupní vrstva se skládá z n vstupů, ke kterým je přiřazena váha spojení. Jeden vstup je vždy bias b, který slouží pro posun aktivační funkce po ose x. Výstupní vrstva obsahuje i výstupů. Každý výstup obsahuje váhový součet a aktivační funkci (angl. activation function). Aktivační funkce se aplikuje na váhový součet vstupů z (2.2). Jako aktivační



Obr. 2.2: Schéma Perceptronu s jedním výstupem

funkce se ve většině případů používá Heaviside funkce H (Jednotkový skok), nebo funkce signum. Výstup Perceptronu oproti McCulloch-Pitts neuronu nemusí být binární, ale závisí na použité aktivační funkci.

$$z = b + \sum_{1}^{n} x_n \cdot w_n \tag{2.2}$$

$$H(z) = \begin{cases} 1 & f(z) \geq 0 \\ 0 & f(z) < 0 \end{cases} \tag{2.3}$$

2.1.1 Trénování Perceptronu

Rosenblattovův algoritmus učení Perceptronu [10], byl inspirován Hebbovským učením. Ve své knize [6] Donald Hebb uvedl , že pokud biologický neuron aktivuje jiný neuron, tak spojení mezi nimi zesílí. To znamená, že váha mezi dvěma neurony se zvětší pokaždé, když mají stejný výstup. Pro trénování Perceptronů se bere v potaz výstupní chyba Perceptronu. Do Perceptronu pouštíme jednotlivá trénovací data (známe správný výsledek) a získáváme výstupy. Pokaždé když bude výstup špatně, tak zesílíme váhy vstupů, které by vedly ke správnému řešení. Rosenblatt dokázal, že pokud je problém lineárně separabilní, tak tento algoritmus bude konvergovat k řešení. Tento teorém se nazývá *Perceptron convergence theorem.*

$$w_{n,i}^{(next)} = w_{n,i} + \eta(y_i - \hat{y}_i)x_n \tag{2.4}$$

- $w_{n,i}^{(next)}$ je váha spojení mezi *n-tým* vstupním neuronem a *i-tým* výstupním neuronem, která se použije v dalším kroku.
- $w_{n,i}$ je váha spojení mezi *n-tým* vstupním neuronem a *i-tým* výstupním neuronem.
- x_n je *n-tá* vstupní hodnota trénovacích dat.
- $\hat{y}_i$ je výstup *i-tého* výstupního neuronu.
- y_i je správná hodnota výstupu *i-tého* výstupního neuronu pro dané trénovací data.
- η je hyperparametr *learning rate*- ovlivňuje velikost změny.

2.1.1.1 Problémy Perceptronu

V roce 1969 Marvin Minsky a Seymour Papert ve své knize *Perceptrons* ukázali několik vážných nedostatků Perceptronu [7]. Nejdůležitější poznatek byl, že Perceptrony nejsou schopny řešit triviální nelineární problém typu XOR. Tento poznatek vedl k velkému poklesu zájmu o umělé neuronové sítě většiny vědců.

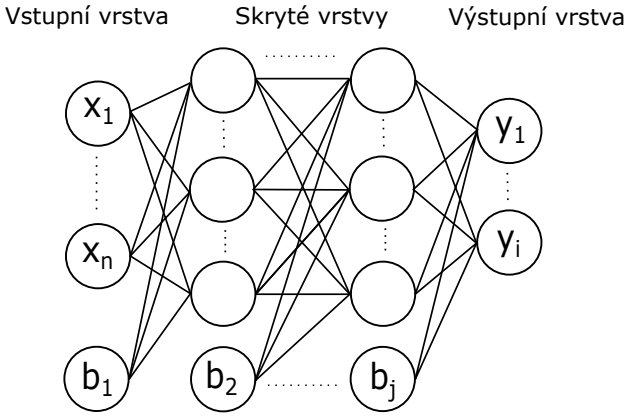
2.2 Vícevrství Perceptron

Ukázalo se, že některá omezení Perceptronu se dají odstranit, pokud se použije více Perceptronů. Výsledná umělá neuronová síť se nazývá vícevrstvý Perceptron (angl. Multi-Layer Perceptron). Vícevrstvý Perceptron (dále jen MLP) obsahuje vstupní vrstvu, jednu nebo více skrytých vrstev a výstupní vrstvu. Počet vstupních a výstupních hodnot záleží na daném problému, který chceme řešit, počet skrytých vrstev a počet neuronů v nich záleží na volbě programátora. Pokud má neuronová síť víc jak jednu skrytou vrstvu, tak se nazývá hluboká neuronová síť (angl. deep neural network). Neuronová síť s jednou skrytou vrstvou se nazývá mělká neuronová síť (angl. shallow neural network). Tyto modely se nazývají dopředné (angl. feedforward), protože informace jde ze vstupu $\mathbf{X}$ do skrytých vrstev a na výstup $\mathbf{Y}$. Výstup ze skryté vrstvy $\mathbf{H}$ a výstupní vrstvy $\mathbf{Y}$ získáme podle rovnice (2.5). Pokud má neuronová síť více skrytých vrstev, tak se jako vstupní hodnoty použijí výstupní hodnoty z předchozí vrstvy.

$$\begin{aligned} H_i &= \sigma(w_{ni} \cdot I_n + b_i) \\ \mathbf{Y} &= \sigma(w_{ij} \cdot H + b_{out}) \end{aligned} \tag{2.5}$$

- w_{ni} je váha spojení mezi *n-tým* vstupním neuronem a *i-tým* výstupním neuronem.
- b je bias daného skrytého nebo výstupního neuronu
- σ je aktivační funkce
- $\mathbf{I}$ je vstupní vektor neuronové sítě.
- $\mathbf{Y}$ je výstupní vektor neuronové sítě.
- $\mathbf{H}$ je vektor hodnot skryté vrstvy neuronové sítě.

Jako aktivační funkce se většinou v dnešní době používá ReLU ve skrytých vrstvách a Softmax ve výstupní vrstvě (slouží k určení pravděpodobnosti výstupu).



Obr. 2.3: Schéma vícevrstvého Perceptronu

2.2.1 Zpětné šíření chyby - Backpropagation

Algoritmus pro trénování Perceptronu se na MLP nedá použít, protože nedokážeme ovlivnit změnu vah skrytých vrstev. V roce 1986 D.E. Rumelhart a kolektiv vydali článek ([11])[1], ve kterém ukázali využití algoritmu zpětného šíření chyby (angl. backpropagation) pro MLP. Než začneme s trénováním MLP musíme inicializovat počáteční parametry (např. náhodně). Poté získáme výstup z neuronové sítě pomocí dopředného šíření a vypočteme chybu (chybovou funkci) s jakou byl výsledek určen. Backpropagation je algoritmus, který slouží k vypočítání gradientu chybové funkce. Chybová funkce je závislá na parametrech (váhy, bias), protože parametry určují výsledný výstup z neuronové sítě. Záporný gradient chybové funkce nám říká, jak je třeba upravit parametry, aby se zmenšila hodnota chybové funkce.

$$\mathbf{W} = \mathbf{W} + (-\nabla C(\mathbf{W}) \tag{2.6}$$

- $\mathbf{W}$ - Vektor všech vah v neuronové síti
- $-\nabla C(\mathbf{W})$ - záporný gradient chybové funkce

Jak moc se změní hodnota chybové funkce pro daný trénovací obrázek v závislosti na vahách ($\frac{\partial C_0}{\partial w_{jk}}$) můžeme zjistit pomocí řetízkového pravidla (angl. chain rule).

$$\begin{aligned} z_j^{(L)} &= \sum_{k=0}^{n_L-1} (w_{jk}^{(L)} a_k^{(L-1)}) + b_j^{(L)} \\ a_j^{(L)} &= \sigma(z_j^{(L)}) \\ \frac{\partial C_0}{\partial w_{jk}^{(L)}} &= \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}} \\ \frac{\partial C_0}{\partial a_k^{(L-1)}} &= \sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}} \end{aligned} \tag{2.7}$$

[1]Samotný algoritmus vymyslel v roce 1974 P. Werbos.

- C_0 hodnota chybové funkce pro daný trénovací obrázek
- a - výstupní hodnota z neuronu
- L - vrstva
- b - bias daného skrytého nebo výstupního neuronu
- σ - aktivační funkce
- z - označení pro přehlednější rovnice.
- $w_{jk}^{(L)}$ - váha spojení mezi neuronem $a_k^{(L-1)}$ a neuronem $a_j^{(L)}$
- n_n - počet neuronů v dané vrstvě

Celková hodnota gradientu se získá průměrem ze součtu příspěvků od každého trénovacího obrázku. Celkový záporný gradient přičteme k vektoru vah a opakujeme stejný proces pro další epochu. V praxi se trénovací data rozdělí do malých částí (mini-batch) a gradient se vypočítá z těchto částí místo z celé trénovací sady. Dojde tak k výraznému zrychlení. Více je popsáno v kapitole s názvem Optimalizace.

3

Konvoluční neuronové sítě

Konvoluční neuronové sítě (angl. convolutional neural networks) vznikly studiem zrakové oblasti (angl. visual cortex). Důležitým milníkem byl rok 1998 kdy Yann LeCun a kol. ve své publikaci ([2]) představili LeNet-5 architekturu s konvoluční a pooling vrstvou. V posledních letech, díky velkému množství dat a výpočetní síle, dokážou konvoluční sítě rozpoznat složité vizuální úkoly s přesností rovnou lidskému rozeznávání. Konvoluční neuronová síť, dále CNN, se dá využít nejen pro rozpoznávání obrázků, ale i pro rozpoznání přirozeného jazyka (angl. natural language processing). Výhodou CNN je, že se učí přímo z dat obrázku a nepotřebují předzpracování dat.

3.1 Vrstvy

CNN je tvořena z jednotlivých vrstev, které se skládají za sebou. Takto tvořené neuronové sítě mají spojení jen mezi sousedními vrstvami a nazývají se sekvenční modely. Opakem jsou grafové modely, které mají více spojení a nemusí být nutně mezi sousedními vrstvami. V práci se zabývám pouze sekvenčním modelem neuronové sítě.

3.1.1 Vstupní vrstva

Jak jsem uvedl na začátku, kapitoly na vstup CNN přivádíme rovnou obrázek, a není tak třeba data předzpracovat. Obrázek na vstupu má tři vlastnosti: šířka, výška a hloubka. Hloubka určuje počet barevných kanálů v obrázku (např. černobílý obrázek má hloubku 1 a barevný (RGB) má hloubku 3)

3.1.2 Konvoluční vrstva

Nejzákladnějším prvkem CNN je konvoluční vrstva, která není plně propojena s každým pixelem v předchozí vrstvě, ale jen s malou částí. Toto se nazývá lokální konektivita a slouží ke snížení počtu parametrů (vah). Propojení je provedeno přes malé konvoluční filtry, které se postupně posouvají po celém obrázku a vytvoří tak příznakové mapy (angl.

feature maps), které se potom předají do další vrstvy. Důležité parametry konvoluční vrstvy jsou:

- *Počet konvolučních filtrů* - určuje kolik konvolučních filtrů použijeme a tím i kolik bude výstupních příznakových map.
- *Velikost konvolučního filtru* - určuje rozměry konvolučního filtru (výška, šířka). Použité rozměry záleží na rozměrech vstupních dat, ale většinou se užívají malé rozměry (3x3, 5x5, 7x7). Často je chyba volit velké rozměry, protože stejného výsledku můžeme dosáhnout užitím více menších filtrů, a to s méně parametry a menším počtem výpočtů.
- *Krok* - (angl. Stride) udává, jak se konvoluční filtr bude posouvat po vstupním obrázku.
- *Zarovnání nulami* - (angl. Zero padding) se používá pokud chceme, aby rozměry vstupního obrázku zůstaly zachovány. Zarovnání se provede přidáním nulových řádků a sloupců okolo vstupní matice. Počet řádků a sloupců pro doplnění lze spočíst ze vzorce $P = \frac{1}{2}(F-1)$ kde F je rozměr filtru. Zarovnání nelze provést na filtry o sudých rozměrech.

$$\mathbf{M}(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n) \tag{3.1}$$

- **M** - Příznaková mapa o rozměru i,j.
- **I** - Vstupní matice.
- **K** - Matice konvolučního filtru o rozměru m,n.

3.1.3 Aktivační vrstva

V aktivační vrstvě se aplikuje aktivační funkce σ na vstupní data a zavede se tak nelinearita do neuronové sítě.

- Tanh - Hyperbolický tangens se dříve často používal, ale bohužel u něj dochází k problému mizejícího gradientu (angl. Vanishing gradient). Při zpětném šíření v neuronové síti se počítá gradient chybové funkce v závislosti na vahách a tento gradient se v každé vrstvě zmenšuje. To znamená, že u vstupní vrstvy může být gradient velice malý a nebude docházet k znatelné změně vah při trénování. V dnešní době je nahrazen funkcí ReLU.

$$\sigma(x_i) = \tanh(x_i) \tag{3.2}$$

- ReLU - (Rectified linear unit) je v dnešní době jedna z nejpoužívanějších aktivačních funkcí v CNN. Pro kladné hodnoty se chová jako lineární funkce a pro záporné je výstup rovný nule (y = max(0,x)). ReLU není složitá matematická funkce a díky tomu se snadno a rychle vypočte. Další výhodou je, že není problém s mizejícím gradientem. Nevýhodou je vynulování všech záporných hodnot, kdy dochází k jevu "dying ReLU". Znamená to, že některé neurony v síti jsou permanentně záporné a nikdy se neaktivují. Takovéto neurony jsou zbytečné a můžeme skončit s velkou částí sítě, která nic nedělá. Problém nastává, pokud je hyperparametr learning rate η nastaven na moc velkou hodnotu nebo pokud je v síti velký záporný bias. Řešením je snížení η a nebo užitím funkce Leaky ReLU.

$$\sigma(x_i) = \max(0, x_i) \tag{3.3}$$

- Leaky ReLU - zabraňuje "umírání" neuronů, přidáním sklonu pro záporné hodnoty (např. 0,01x). Další verzí je parametrický ReLU (PReLU), kde není sklon pevně daný, ale mění si ho neuronová síť. Pro kladné hodnoty se obě funkce chovají stejně jako ReLU.

$$\sigma(x_i) = \max(ax_i, x_i) \tag{3.4}$$

- Softmax - užívá se jako klasifikátor v poslední vrstvě neuronových sítí, protože ze vstupních hodnot udělá pravděpodobnosti. Součet výstupních pravděpodobností je rovný 1.

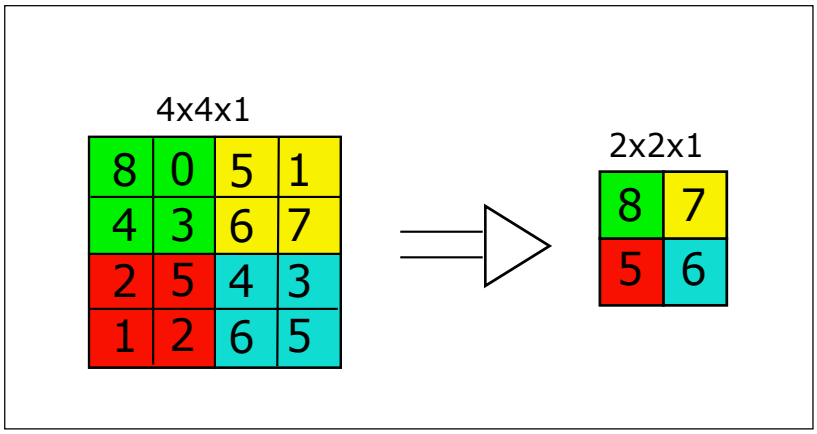
$$\sigma(x_i) = \frac{\mathrm{e}^{x_i}}{\sum_{i=0}^{n} \mathrm{e}^{x_i}} \tag{3.5}$$

3.1.4 Pooling vrstva

Pooling vrstva má za úkol redukovat vstupní obrázek (příznakovou mapu), aby se omezil počet parametrů a výpočetní náročnost. Stejně jako v konvoluční vrstvě je každý neuron v pooling vrstvě spojen s omezeným počtem neuronů z předchozí vrstvy. Tato lokální konektivita je provedena přes pooling kernely s parametry:

- *Velikost pooling kernelu* - určuje rozměry pooling kernelu (výška, šířka).
- *Krok* - (angl. Stride) udává, jak se pooling kernel bude posouvat po vstupním obrázku.
- *Zarovnání nulami* - Je stejné jako u konvoluční vrstvy, ale málokdy se užívá, jelikož pooling vrstvu užíváme pro zmenšení objemu dat.

Pooling vrstva nemá žádné váhové parametry, jen na vstupní data užije danou matematickou funkci. V dnešní době je nejpoužívanější max pooling, kde filtr vyhodnotí nejvyšší hodnotu z matice v dané oblasti. Dříve se používal average pooling nebo L2-pooling, ale ukázalo se, že max pooling má nejlepší výsledky. Pooling vrstva nemění hloubku dat.



Obr. 3.1: Ukázka Max poolingu s krokem 2 a velikostí filtru 2x2, při velikosti vstupu 4x4

V poslední době se začali objevovat architektury, které pooling vrstvu nevyužívají a místo toho používají konvoluční vrstvu s krokem 2. Tyto neuronové sítě (all-convolutional nets) se používají stále častěji, ale ještě úplně nenahradily pooling.

3.1.5 Plně propojená vrstva

Plně propojená vrstva (angl. fully connected layer), je dopředná neuronová síť z předchozí kapitoly. Jako aktivační funkce se používá ReLU a Softmax (na konci neuronové sítě). Na vstupu CNN je jeden obrázek, který postupováním v síti zmenšuje svůj rozměr a zvyšuje svoji hloubku. Předtím než tyto data pošleme na vstup plně propojené vrstvy je třeba je upravit do podoby sloupcového vektoru.

3.2 Zpětné šíření chyby v CNN

- Plně propojená vrstva - zpětné šíření chyby probíhá stejně, jako u MLP.
- Max pooling vrstva - při dopředném šíření je důležité si zapamatovat body, ve kterých bylo určeno maximum, protože chybu šíříme jen přes tyto body a ostatní body matice jsou rovny 0. Vrstva obsahuje jen fixní funkci nemění se její parametry.
- Konvoluční vrstva - zpětné šíření (dat i vah) je také konvoluční operace, jen s prostorově převrácenými filtry (rotace o 180°) a vynásobené derivací aktivační funkce.

3.3 Architektury

Konvoluční neuronové sítě se typicky skládají z několika konvolučních vrstev, které následuje aktivační vrstva a pooling vrstva, kterou mohou znova tyto vrstvy následovat. Síť je zakončena jednou nebo více plně propojených vrstev, kdy poslední má vždy jako akti-

vační funkci Softmax nebo jiný klasifikátor. Níže popíšu nejznámější architektury, které se používají.

3.3.1 LeNet-5

LeNet-5 je pravděpodobně nejznámější architektura CNN. Jak bylo zmíněno v úvodu, vytvořil ji v roce 1998 Yann LeCun. Architektura obsahuje 8 vrstev a byla vytvořena pro rozpoznání ručně psaných číslic (MNIST dataset). Yann LeCun s touto architekturou dosáhl chybovosti pod 1%

Tab. 3.1: LeNet-5 Architektura

Vrstva	Typ	Map	Rozměr	Rozměr kernelu	Krok	Aktivační funkce
1	Vstup	1	32x32	-	-	-
2	Konvoluce	6	28x28	5x5	1	tanh
3	Avg Pooling	6	14x14	2x2	2	tanh
4	Konvoluce	16	10x10	5x5	1	tanh
5	Avg Pooling	16	5x5	2x2	2	tanh
6	Konvoluce	120	1x1	5x5	1	tanh
7	FC	-	84	-	-	tanh
8	FC	-	10	-	-	Gausovo propojení

3.3.2 AlexNet

AlexNet CNN architektura vyhrála v soutěži ILSVRC 2012 challenge s dosaženou chybovostí 17%. Architektura na druhém místě dosáhla chybovosti 26%. Architekturu vytvořil Alex Krizhevsky (proto AlexNet), I. Stuskever a G. Hinton. Je podobná LeNet-5 architektuře, jen je mnohem vetší a hlubší. Jedná se o první architekturu, která použila více konvolučních vrstev za sebou. Pro redukování přetrénování byl použit dropout (dropout rate $p = 50\%$) ve vrstvách 9 a 10. Byla provedena i augmentace dat náhodným posuvem, horizontálním převracením a změnou světelných podmínek.

3.3.3 GoogLeNet

GoogLeNet architekturu vytvořil Christian Szegedy et al. a vyhrál s ní ILSVRC 2014 challenge s chybovostí pod 7% [8]. Této přesnosti bylo možné dosáhnout, jelikož GoogLeNet je mnohem hlubší než předchozí CNN. V architektuře jsou použity podsítě (angl. sub-networks) s názvem inception module, tyto moduly umožňují efektivní využití parametrů a paralelní využití pooling a konvolučních vrstev. GoogLeNet má přibližně 10x méně parametrů, než AlexNet (přibližně 6 milionů oproti 60 milionům).

Tab. 3.2: AlexNet Architektura

Vrstva	Typ	Map	Rozměr	Rozměr kernelu	Krok	Aktivační funkce
1	Vstup	3(RGB)	224x224	-	-	-
2	Konvoluce	96	55x55	11x11	4	ReLU
3	Max Pooling	96	27x27	3x3	2	-
4	Konvoluce	256	27x27	5x5	1	ReLU
5	Max Pooling	256	13x13	3x3	2	-
6	Konvoluce	384	13x13	3x3	1	ReLU
7	Konvoluce	384	13x13	3x3	1	ReLU
8	Konvoluce	256	13x13	3x3	1	ReLU
9	FC	-	4096	-	-	ReLU
10	FC	-	4096	-	-	ReLU
11	FC	-	1000	-	-	Softmax

3.3.4 ResNet

ResNet (Residual Network) architektura vyhrála v ILSVRC 2015 challenge s chybovostí pod 3,6%. ResNet je velice hluboká CNN, složená ze 152 vrstev. ResNet architektura zavedla myšlenku propojení vrstev o více než jednu vrstvu (např. spojení druhé vrstvy s pátou). Toto spojení omezilo problém mizejícího gradientu a umožnilo tak trénování takto hluboké sítě (residual learning).

4

Optimalizace

Neuronové sítě obsahují velké množství parametrů, které chceme upravovat pro optimální výsledky sítě. Optimalizační algoritmy mají za úkol najít globální minimum chybové funkce (nebo chybu alespoň co nejvíce minimalizovat), a toto minimum najít v co nejkratší době. Další důležitá optimalizace je zabránění přetrénování neuronové sítě.

4.1 Chybová funkce

Aby se neuronová síť dala optimalizovat, tak nejdříve musíme být schopni ji ohodnotit. K ohodnocení se využívá chybová funkce (angl. Cost function nebo Error function), která určí velikost chyby sítě. Jedna z možných chybových funkcí je střední kvadratická chyba (angl. Mean Squared Error) dále MSE. Celková chybová funkce je průměr součtu chyb všech trénovacích obrázků.

$$C = \frac{1}{2n} \sum_{i=0}^{n} (y_i - \hat{y}_i)^2 \tag{4.1}$$

- n - počet trénovacích obrázků
- y_i - požadovaný výstup z neuronové sítě
- $\hat{y}_i$ - skutečný výstup z neuronové sítě

4.1.1 Kategoriální cross-entropie

Kategoriální cross-entropie (angl. categorical cross-entropy) je oblíbená chybová funkce, která se využívá hlavně z důvodu své rychlosti konvergence, hlavně v sítích s aktivační funkcí softmax. Očekávaný vstup je vektor o stejném počtu prvků jako je kategorií daného problému s hodnotou 1 na pozici zrovna rozpoznávané kategorie a s hodnotou 0 na ostatních (one-hot encoding).

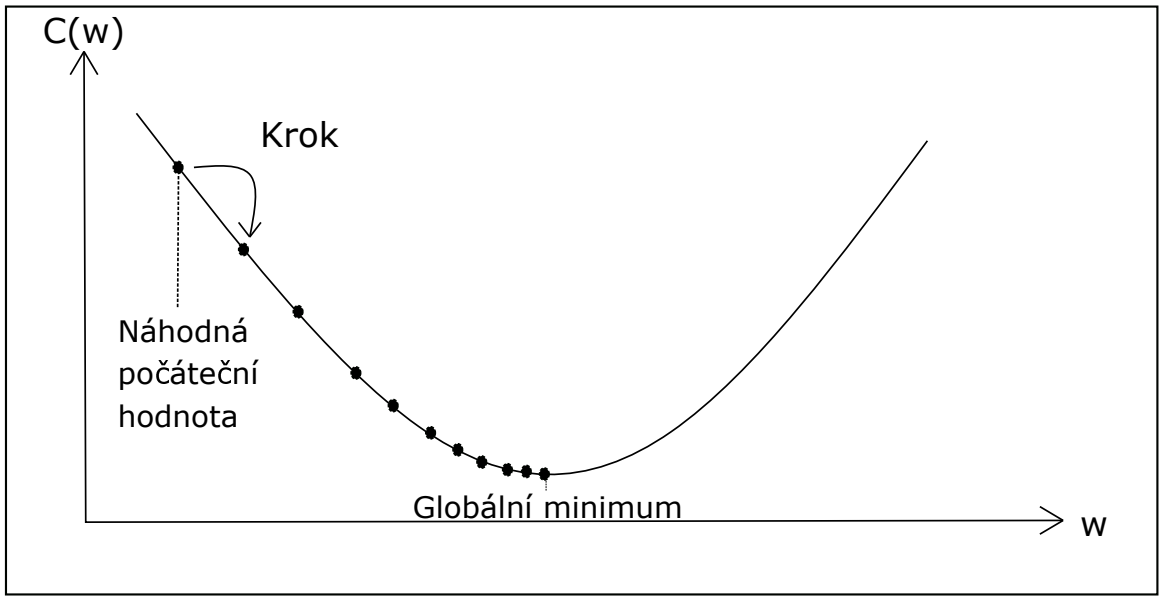
$$C = -\sum_{j=0}^{n} \sum_{i=0}^{m} \frac{(y_i \log(\hat{y}_i)}{n} \tag{4.2}$$

- n - počet trénovacích obrázků
- m - počet kategorií
- y_i - požadovaný výstup z neuronové sítě
- $\hat{y}_i$ - skutečný výstup z neuronové sítě

Modifikace kategoriální cross-entropie, která nevyužívá one-hot encoding se nazývá řídká kategoriální cross-entropie (angl. categorical cross-entropy). Předpis zůstává stejný.

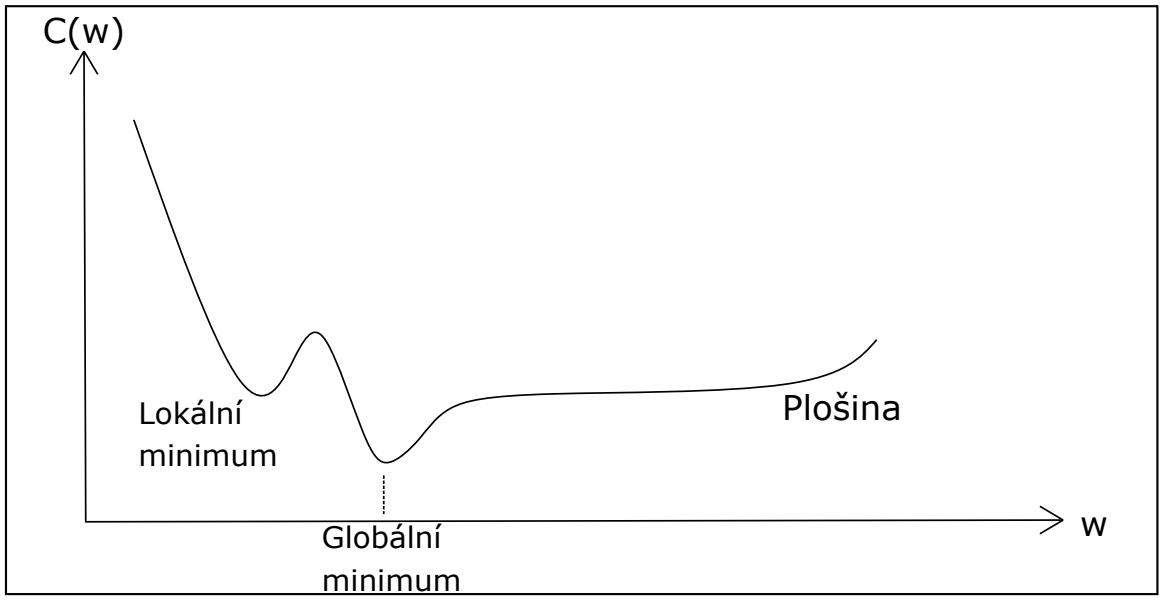
4.2 Optimalizační algoritmy

Základním optimalizačním algoritmem je gradientní sestup (angl. gradient descent), který je schopný najít optimální řešení k velkému množství problémů. Základem je postupná úprava parametrů pro minimalizaci chybové funkce. Algoritmus měří lokální gradient chybové funkce s ohledem na parametry vektoru **W** a jde ve směru sestupného gradientu. Minimum se nachází v místě, kde je gradient rovný nule. Nejdříve se musí náhodně inicializovat hodnoty ve vektoru **W** a z této pozice algoritmus začne krok po kroku snižovat chybovou funkci, dokud algoritmus nedosáhne minima. Důležitým parametrem je veli-



Obr. 4.1: Gradientní sestup

kost kroků, kterou určuje hyperparametr η (koeficient učení). Pokud bude koeficient příliš malý, tak algoritmus bude potřebovat mnoho iterací, aby dosáhl minima. Pokud naopak bude koeficient příliš velký, tak může přeskočit minimum a skončit výše, než byla začáteční inicializace. Toto vede k tomu, že algoritmus bude divergovat. Chybová funkce většinou nemá tvar misky, ale má lokální minima, plošiny a jiné nepravidelné terény, což dělá nacházení globálního minima složitějším. V případě chybové funkce ve tvaru jako



Obr. 4.2: Chybová funkce

je na obrázku 4.2, pokud jsou zvolený začáteční hodnoty nalevo, tak algoritmus bude konvergovat k lokálnímu minimu. Při začátečních hodnotách zvolených napravo, bude algoritmu trvat dlouhou dobu, než se dostane přes plošinu a pokud ukončíme trénování brzo, tak minima nikdy nedosáhne. V případě, že použijeme chybovou funkci MSE tak se o to nemusíme starat, protože MSE nemá lokální minimum, ale jen globální. Tento algoritmus je sice přesný, ale pomalý.

4.2.1 Mini-batch gradientní sestup

Mini-batch gradientní sestup je algoritmus, který místo výpočtu gradientu z celé trénovací sady, počítá gradient, jen z malé skupiny položek z trénovací sady. Menší sadu (mini-batch) získáme tak, že se trénovací sada náhodně rozdělí na několik menších sad (každá tato sada by měla obsahovat stejný poměr prvků z daných kategorií). Výhoda tohoto algoritmu je hlavně rychlost.

4.2.2 Momentová optimalizace

Myšlenka momentové optimalizace vychází z koule, která se valí z kopce dolů. Nejdříve se valí pomalu a potom nabírá na rychlosti. Oproti obyčejnému gradientnímu sestupu, jelikož nebere v potaz předchozí hodnoty gradientu, tak dělá malé kroky pokud je lokální gradient malý. Momentová optimalizace předchozí hodnoty gradientu bere v potaz. V každé iteraci odečte lokální gradient od momentového vektoru m, který je násobený koeficientem učení a nastaví váhy jednoduše přidáním tohoto momentového vektoru. Pro zabránění nekontrolovaného zvětšování momentového vektoru je v algoritmu hyperparametr β nazývaný *momentum*. Momentum si lze představit jako tření a nabývá hodnot od

0 (velké tření) do 1 (žádné tření).

$$\begin{aligned} 1. \quad & \mathbf{m} = \beta\mathbf{m} - \eta\nabla C \\ 2. \quad & \mathbf{W} = \mathbf{W} + \mathbf{m} \end{aligned} \tag{4.3}$$

4.2.3 RMSProp optimalizace

RMSProp je algoritmus s adaptivním koeficientem učení. Představil ho Tieleman T. a Geoffrey H. v roce 2012. Algoritmus akumuluje jenom gradienty z nedávných iterací (místo všech gradientů od začátku trénování) do vektoru $\mathbf{s}$. Docílí se toho užitím míry úpadku (angl. decay rate) β. I když je β nový hyperparametr, tak se vetšinou nemusí ladit, protože hodnota 0.9 funguje velice dobře.

$$\begin{aligned} 1. \quad & \mathbf{s} = \beta\mathbf{s} + (1-\beta)\nabla C \otimes \nabla C \\ 2. \quad & \mathbf{W} = \eta\nabla C \oslash \sqrt{\mathbf{s} + \epsilon} \end{aligned} \tag{4.4}$$

- ϵ - prvek, který zajišťuje, aby nedošlo k dělení nulou. Typická hodnota 10^{-10}
- $\oslash$ - dělení prvek po prvku
- $\otimes$ - součin prvek po prvku

4.2.4 Adam optimalizace

Adam [12] je adaptivní odhad momentu (angl. adaptive moment estimation), který kombinuje momentovou optimalizaci a RMSProp optimalizaci. Adam stejně jako momentová optimalizace bere v potaz gradienty z předchozí iterace (krok 1) a kvadráty gradientu z předchozí iterace jako RMSProp (krok 2.). Krok 3 a 4 pomáhají ke zvýšení $\mathbf{m}$ a $\mathbf{s}$, protože jsou na začátku trénování inicializovány na 0. Hyperparametr $beta_1$ se typicky nastaví na 0.9 a hyperparametr $beta_2$ se typicky nastavuje na hodnotu 0.999. Adam stejně jako RMSProp patří mezi algoritmy s adaptivním koeficientem učení a potřebuje méně ladění tohoto hyperparametru (většinou $\eta = 0.001$)

$$\begin{aligned} 1. \quad & \mathbf{m} = \beta_1\mathbf{m} - (1-\beta_1)\nabla C \\ 2. \quad & \mathbf{s} = \beta_2\mathbf{s} + (1-\beta_2)\nabla C \otimes \nabla C \\ 3. \quad & \mathbf{m} = \frac{\mathbf{m}}{1-\beta_1^t} \\ 4. \quad & \mathbf{m} = \frac{\mathbf{m}}{1-\beta_1^t} \\ 5. \quad & \mathbf{W} = \mathbf{W} + \eta\mathbf{m} \oslash \sqrt{\mathbf{s} + \epsilon} \end{aligned} \tag{4.5}$$

- t - iterační číslo (začíná na 1)

4.3 Přetrénování neuronové sítě

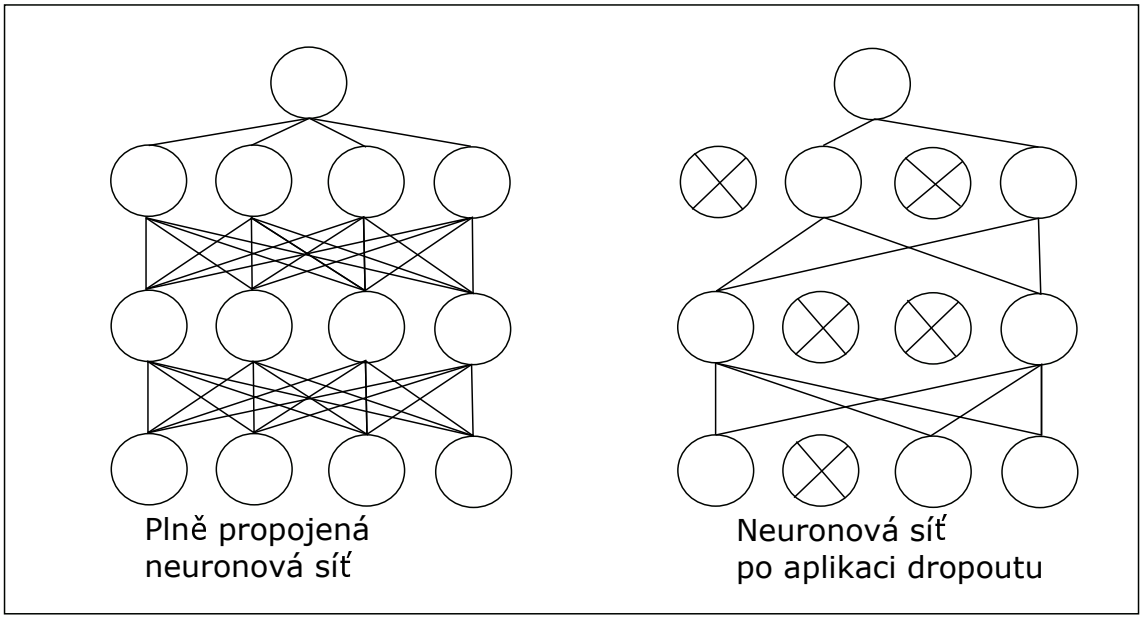
Přetrénování (angl. overfitting) znamená, že model je velice dobrý v rozpoznávání trénovacích dat, ale špatně rozeznává validační data. Lze si to vysvětlit tak, že model není dostatečně obecný, ale zapamatoval si přesně trénovací data a nedokáže dobře rozeznat data, které se lehce liší od trénovacích. Validační data se využívají pro ohodnocení modelu během trénování a pro zisk informací pro úpravu hyperparametrů. Validační data jsou část dat oddělených od trénovacích. Nejjednodušší způsob odstranění přetrénování je přidání více dat, čímž zvýšíme různorodost dat a model se tak stane více obecným. Ne vždy ale máme přístup k více datům, a tak musíme využít regularizaci.

4.4 Regularizace

Abychom zamezily přetrénování, využíváme regularizační algoritmy. Algoritmus předčasného zastavení (angl. early stopping) zabraňuje přetrénování tak, že ukončí trénování hned, jak se začne přesnost na validačních datech zhoršovat. Předčasné zastavení funguje velmi dobře, ale lepších výsledků se dosáhne, když se zkombinuje s další regularizační technikou.

4.4.1 Dropout

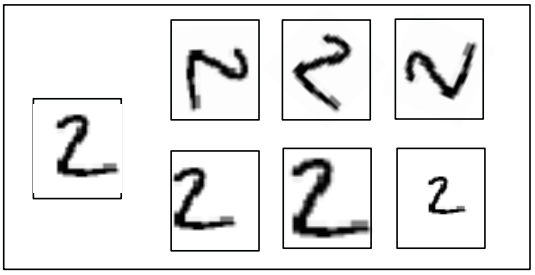
Dropout je populární regularizační technika popsaná N. Srivastavem. Dropout byl prokázaný jako velice úspěšný a dosahuje zlepšení 1-2% i na špičkových (state-of-the-art) neuronových sítích. Je to velice jednoduchý algoritmus. V každém trénovacím kroku, má každý neuron (kromě výstupních) pravděpodobnost p na to, že bude deaktivován a bude ignorován během trénovacího kroku. Neuron, který byl deaktivován v jednom kroku může být aktivní v dalším kroku. Hyperparametr p se nazývá dropout rate a typicky je nastavený na 50% pro skryté vrstvy a 20% pro vstupní vrstvu. Pro p nastavené na 50%, bude při testování každý neuron připojený k dvojnásobku vstupních neuronů, než při trénování. Po skončení trénování, tento fakt kompenzujeme vynásobením vstupních vah neuronů hodnotou 0.5. Když to neuděláme, tak každý neuron bude mít přibližně dvojnásobně velké vstupní hodnoty, a to povede k nepřesným výsledkům. Při zvolení obecné hodnoty p je třeba vstupy vynásobit hodnotou (1-p).



Obr. 4.3: Dropout model: **Vlevo**: Neuronová síť o 2 skrytých vrstvách. **Vpravo**: Neuronová síť po aplikaci dropoutu (neurony s křížkem jsou neaktivní).

4.4.2 Augmentace dat

Augmentace dat (angl. Data Augmentation) je regularizační technika, která upravuje trénovací data a vytváří z nich nové, čímž uměle zvětší velikost trénovacích dat a omezuje tak přetrénování sítě. Možné úpravy jsou rotace, translace, změna osvětlení, přidání gaussovského šumu a přiblížení části obrázku. Můžeme využít všechny úpravy najednou, ale jestli jsou vždy vhodné záleží na daném datasetu. Snaha je přidat užitečná data ne irelevantní. Například u rozpoznávání aut nemá význam rotace, kde je auto vzhůru nohama (pokud nechceme rozpoznávat např. nabouraná auta).



Obr. 4.4: Ukázka augmentace.

5

Grafické procesory pro hluboké učení

Hluboké učení (angl. deep learning) je založeno na spojení mezi neurony. Samotný umělý neuron nedokáže řešit složité problémy a schopnosti řešit složité problémy lze dosáhnout, až po spojení velkého množství neuronů. V dnešní době mají moderní neuronové sítě $10^6 - 10^7$ neuronů. Tradičně se neuronové sítě trénovaly s užitím jednoho CPU. Tento postup je dnes považován za nedostatečný. K trénování takto hlubokých neuronových sítí, se užívá více GPU najednou. Trénovací data rozdělí na části (angl. batch), a každá část se paralelně počítá na jednom GPU.

5.1 Funkce grafických procesorů

Grafické procesory (angl. graphic processing unit), dále GPU, byly původně navrženy pro grafické aplikace. Hlavním podnětem pro rozvoj grafických procesorů, byl trh s herními systémy. Výkonnostní požadavky na dobrý herní systém se ukázaly, jako velice prospěšné pro neuronové sítě. Renderování videoher potřebuje provádět mnoho operací paralelně v co nejkratším čase. Modely objektů a prostředí jsou udávány v 3-D souřadnicích vrcholů. Pro převedení těchto objektů na 2-D souřadnice, musí grafická karta provést paralelně maticové dělení a násobení souřadnic. Poté musí grafická karta provést mnoho paralelních výpočtů na každém pixelu, aby určila barvu každého pixelu. V obou případech se jedná o jednoduché výpočty, které neobsahují mnoho větvení, v porovnání s běžnou výpočetní úlohou pro CPU. Výpočty jsou na sobě nezávislé, a proto je lehká jejich paralelizace. Grafické karty jsou navrženy pro vysoký stupeň paralelismu a s velkou šířkou pásma paměti (angl. memory bandwidth), za cenu nižší rychlosti hodin a omezenější schopnosti na větvení (než u CPU).

5.2 Využití GPU v neuronových sítích

Většina moderních neuronových sítí se implementuje na GPU, protože algoritmy neuronových sítí mají stejné požadavky na výkon, jako real-time grafické algoritmy popsané v 5.1. Neuronové sítě většinou obsahují velké množství parametrů, aktivačních hodnot a hodnot gradientů. Každá z těchto hodnot se musí upravit v každém kroku trénování. Buffer těchto hodnot je velice rozsáhlý a limitujícím faktorem často bývá šířka pásma paměti systému. Zde má GPU výhodu díky velké šířce pásma paměti. Trénovací algoritmy neuronových sítí typicky neobsahují mnoho větvení, a proto jsou vhodné pro GPU. Neuronová síť se dá rozdělit, na jednotlivé neurony, které se zpracují nezávisle na ostatních neuronech v dané vrstvě. A proto neuronové sítě dobře využívají paralelismu výpočtů v GPU. Grafické karty byly původně tak specializovány, že se daly použít pouze na grafické úlohy. Postupem času se grafické karty staly více flexibilní. Tyto grafické karty se daly využít pro vědecké výpočty a výsledek výpočtu se zapisoval do bufferu hodnot pixelů. Steinkraut et al.(2005) realizoval dvouvrstvou plně propojenou neuronovou síť na GPU a dosáhl trojnásobného zrychlení oproti CPU.

5.3 Psaní kódu pro GPU

Popularita grafických karet pro neuronové sítě prudce narostla po příchodu všeobecných (angl. general purpose) grafických procesorů. Na GP-GPU je možné provést libovolný kód, nejen podprogram renderování. Programovací jazyk CUDA byl představen společností NVIDIA v roce 2006 a umožňoval psát kód pro GPU s podobnou syntaxí jako programovací jazyk C. CUDA je podporována pouze grafickými kartami od společnosti NVIDIA, jako alternativu lze použít OpenCL (např. pro GPU od společnosti AMD). GP-GPUs jsou nyní ideální platformou pro programování neuronových sítí. Techniky pro psaní efektivního kódu na GPU jsou výrazně odlišné od technik pro CPU. Dobře napsaný kód pro CPU je navrhnut tak, aby četl informace co nejvíce z cache paměti. Na GPU může být rychlejší, spočítat stejnou hodnotu dvakrát, než ji vypočítat jednou a číst jí poté z paměti. Operace s pamětí je rychlejší pokud jsou tyto operace sloučené (angl. coalesced). Sloučené čtení a zápis nastává, pokud několik vláken může zapisovat, nebo číst hodnoty, které potřebují současně. Dále je důležité, aby každé vlákno v bloku současně provádělo stejnou instrukci.

6

Implementace

Cílem mé práce bylo realizovat libovolnou strukturu neuronové sítě na GPU a CPU. Otestovat ji na trénovací množině a porovnat rychlost trénování modelu na GPU a CPU. Zajímala mě oblast počítačového vidění, a proto jsem se rozhodl realizovat konvoluční neuronovou síť. Jako vlastní cíl práce jsem si stanovil, že danou neuronovou síť realizuji v jazyce C# jen s užitím knihovny pro práci s maticemi. Realizované CNN byly trénovány na:

- GPU - NVIDIA GeForce GTX 1050 Ti
- CPU - Intel Core i5-8300H

6.1 Programovací jazyk

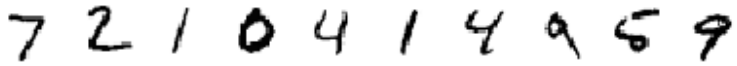
Jako programovací jazyk jsem zvolil Python 3.7.2 a využil jsem rozhraní Keras, které obaluje framework TensorFlow. Keras umožňuje snadné a intuitivní sestavení neuronových sítí. Další výhodou je snadná aplikace na GPU a podrobná dokumentace.
(dostupná z: https://keras.io)

6.2 Trénovací množiny

Pro porovnání rychlosti jsem zvolil dataset MNIST a CIFAR-10. Tyto datasety jsem si zvolil kvůli malým rozměrům a jednoduchosti. Čím složitější problém, tím musí být vetší počet parametrů a prodlužuje se doba trénování.

6.2.1 MNIST

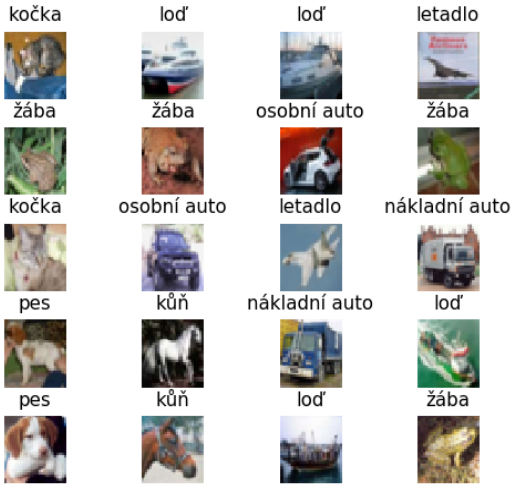
MNIST je dataset ručně psaných číslic, obsahujících 70000 vzorků. Vzorky jsou rozdělené na 60000 trénovacích a 10000 testovacích vzorků. Jedná se o upravený subset databáze NIST. Číslice jsou normalizované a vycentrované na fixní velikost 28x28 pixelů. Jsou to černobílé obrázky, takže každý pixel má hodnoty od 0 (bílý) po 255 (černý).



Obr. 6.1: Ukázka číslic z MNIST datasetu

6.2.2 CIFAR-10

CIFAR-10 dataset obsahuje 60000 barevných obrázků (50000 trénovacích a 10000 testovacích) s rozlišením 32x32. Obrázky patří do 10 kategorií (letadlo, osobní auto, pták, kočka, jelen, pes, žába, kůň, loď, nákladní auto), kde každá kategorie obsahuje 6000 obrázků.



Obr. 6.2: Ukázka několika obrázků z CIFAR-10 datasetu

6.3 Řešení pro MNIST

MNIST je jednoduchý problém, proto stačí jednoduchá CNN architektura o 28458 parametrech:

1. Konvoluční vrstva - 28 filtrů o rozměru 3x3, krok = 1 a bez zarovnání nulami
2. Aktivační vrstva - ReLU funkce
3. Max-pooling vrstva - kernel o rozměru 2x2 s krokem 2
4. Dropout vrstva - dropout rate $p = 0.2$
5. Konvoluční vrstva - 56 filtrů o rozměru 3x3, krok = 1 a bez zarovnání nulami
6. Aktivační vrstva - ReLU funkce
7. Max-pooling vrstva - kernel o rozměru 2x2 s krokem 2

8. Dropout vrstva - dropout rate $p = 0.5$
9. Flatten vrstva
10. Plně propojená vrstva - 10 neuronů (počet kategorií)
11. Aktivační vrstva - Softmax funkce

Jako optimalizační algoritmus je použit Adam s koeficientem učení η=0.001 a hyperparametry β_1= 0.9, $\beta_2 = 0.999$. Chybovou funkci jsem použil řídkou kategoriální cross-entropii. Kód pro tuto CNN architekturu je přiložen v příloze A.

6.3.1 Dosažené výsledky

Na této architektuře jsem dosáhl přesnosti 99.32% po 32 epochách. Doba trénování jedné epochy byla 7 sekund na GPU a 25 sekund na CPU. Ve druhé architektuře jsem odstranil poslední konvoluční vrstvu a pooling vrstvu. Místo nich jsem použil plně propojenou vrstvu se 128 neurony. Tato architektura dosáhla horší přesnosti 99.05% a trénování trvalo 39 epoch. Doba trénování jedné epochy byla 8 sekund na GPU a 28 sekund na CPU.

Tab. 6.1: Porovnání architektur CNN pro MNIST dataset

Architektura	Čas trénovaní CPU	Čas trénování GPU	Přesnost	Počet epoch
1	25sec	7sec	0.9932	32
2	28sec	8sec	0.9905	39

Architekturu (1) jsem trénoval i na clusteru Govorun s těmito výsledky:

Tab. 6.2: Výsledky trénování CNN na clusteru Govorun pro MNIST dataset

Architektura	Čas trénovaní CPU	Čas trénování GPU	Přesnost	Počet epoch
1	24sec	9sec	0.9884	17

Výsledky na clusteru jsou srovnatelné s výsledky na mém CPU a GPU. Důvod pro to může být, že kód dobře nevyužívá více CPU a GPU. Dalším důvodem může být dlouhé načítání kernelu a zrychlení by se tedy ukázalo až na výpočetně náročnějších problémech. Bohužel z důvodu chyby jsem neodzkoušel architekturu pro CIFAR-10.

6.4 Řešení pro CIFAR-10

Pro CIFAR-10 jsem zvolil složitější architekturu CNN, která obsahuje 292138 parametrů.

1. Konvoluční vrstva - 32 filtrů o rozměru 3x3, krok = 1 a se zarovnáním nulami
2. Aktivační vrstva - ReLU funkce
3. Konvoluční vrstva - 32 filtrů o rozměru 3x3, krok = 1 a bez zarovnání nulami
4. Aktivační vrstva - ReLU funkce
5. Max-pooling vrstva - kernel o rozměru 2x2 s krokem 2
6. Dropout vrstva - dropout rate $p = 0.2$
7. Konvoluční vrstva - 64 filtrů o rozměru 3x3, krok = 1 a se zarovnáním nulami
8. Aktivační vrstva - ReLU funkce
9. Konvoluční vrstva - 64 filtrů o rozměru 3x3, krok = 1 a bez zarovnání nulami
10. Aktivační vrstva - ReLU funkce
11. Max-pooling vrstva - kernel o rozměru 2x2 s krokem 2
12. Dropout vrstva - dropout rate $p = 0.5$
13. Konvoluční vrstva - 128 filtrů o rozměru 3x3, krok = 1 a se zarovnáním nulami
14. Aktivační vrstva - ReLU funkce
15. Konvoluční vrstva - 128 filtrů o rozměru 3x3, krok = 1 a bez zarovnání nulami
16. Aktivační vrstva - ReLU funkce
17. Max-pooling vrstva - kernel o rozměru 2x2 s krokem 2
18. Dropout vrstva - dropout rate $p = 0.5$
19. Flatten vrstva
20. Plně propojená vrstva - 10 neuronů (počet kategorií)
21. Aktivační vrstva - Softmax funkce

Jako optimalizační algoritmus je použit Adam s koeficientem učení η=0.001 a hyperparametry β_1= 0.9, β_2 = 0.999. Chybovou funkci jsem použil kategoriální cross-entropii. Kód pro tuto CNN architekturu je přiložen v příloze B.

6.4.1 Dosažené výsledky

Na této architektuře jsem dosáhl přesnosti 83.08% po 104 epochách. Doba trénování jedné epochy byla 10 sekund na GPU a 131 sekund na CPU.

Tab. 6.3: Výsledky CNN architektury pro CIFAR-10

Architektura	Čas trénovaní CPU	Čas trénování GPU	Přesnost	Počet epoch
1	131sec	10sec	0.8308	104

7

Závěr

V úvodu práce byl proveden rozbor umělých neuronových sítí a algoritmu zpětného šíření chyby. Z dnes užívaných typů neuronových sítí byly popsány konvoluční neuronové sítě a jejich populární architektury. Rekurentní neuronové sítě, které se užívají pro zvuková a textová data, nejsou v této práci popsány. Dále byl proveden rozbor užívaných optimalizačních a regularizačních algoritmů. V kapitole 5 byla popsána funkce grafických procesorů a jejich využití při trénování neuronových sítí.
Pro porovnání rychlosti byly sestaveny dvě konvoluční neuronové sítě, které byly realizovány pomocí knihovny TensorFlow a Keras. Experimenty byly provedeny na jednoduchých problémech rozpoznávání ručně psaných číslic (MNIST) a rozpoznávání barevných objektů (CIFAR-10). Z výsledků provedených měřením vyplývá, že doba trénování na CPU prudce narůstá s počtem parametrů sítě. Dále vyplývá, že GPU programování výrazně urychluje trénování hlubokých neuronových sítí. K výraznému zlepšení dochází i u jednoduchých CNN, s malým počtu parametrů, proto je nezbytné, provádět trénování moderních hlubokých sítí na GPU.

Výsledky řešení bakalářské práce lze stručně shrnout do následujících bodů:

- byl proveden rozbor používaných neuronových sítí, včetně rozboru optimalizačních metod
- byla zvolena konvoluční neuronová síť, která byla podrobně popsána v kapitole 3
- CNN byla prakticky zrealizována na CPU a GPU pomocí knihovny Keras
- byla zvolena trénovací množina MNIST a CIFAR-10 pro porovnání rychlosti
- byla porovnána rychlost trénování na dvou architekturách CNN

Další postup bakalářské práce:

- Dokončení realizace knihovny v jazyce C# - jsou implementovány jednotlivé vrstvy, ale nejsou plně konfigurovatelné a nepodporují GPU. Z časových důvodů jsem tento cíl nedodělal

- Govorun cluster - Napsat kód pro lepší využití clusteru a porovnat rychlosti na rozsáhlejších neuronových sítích

Literatura

[1] Srivastava, N., et al. *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*. 2014 [cit. 4.6.2019], Dostupné z: http://goo.gl/DNKZo1

[2] LeCun, Y., et al. *Gradient-Based Learning Applied to Document Recognition*. 1998 [cit. 5.6.2019], Dostupné z: http://goo.gl/A347S4

[3] GÉRON, A. *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems.* Boston: O'Reilly Media, 2017. ISBN 978-1-491-96229-9.

[4] SANDERS, Jason a Edward KANDROT. *CUDA by example: an introduction to general-purpose GPU programming*. NVIDIA Corporation, 2011. ISBN 9780131387683.

[5] GOODFELLOW, I., BENGIO Y., COURVILLE, A. *Deep Learning*. Cambridge, Massachusetts: The MIT Press, 2016. ISBN 0-26-203561-8.

[6] HEBB, D. O. *he organization of behavior: a neuropsychological theory*. JOHN WILEY AND SONS. INC., 1949 [cit. 30.5.2019], Dostupné z: https://pure.mpg.de/rest/items/item˙2346268/component/file˙2346267/content

[7] Marvin, L., Papert, M., Papert, S. *Perceptrons*. 1988 [cit. 6.6.2019], Dostupné z: https://leon.bottou.org/publications/pdf/perceptrons-2017.pdf

[8] Szegedy, C., et al. *Going Deeper with Convolutions*. 2015 [cit. 8.6.2019], Dostupné z: https://goo.gl/tCFzVs

[9] McCulloch, W., S., Pitts, W. *A Logical Calculus of Ideas Immanent in Nervous Activity*. 1943 [cit. 1.6.2019], Dostupné z: https://www.cs.cmu.edu/∼./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf

[10] Rosenblatt, F. *THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN*. 1957 [cit. 30.5.2019], Dostupné z: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.335.3398&rep=rep1&type=pdf

[11] Rumelhart, D., E., Hinton, G., E., Williams, R., J. *Learning Internal Representations by Error Propagation*. 1986 [cit. 2.6.2019], Dostupné z: https://web.stanford.edu/class/psych209a/ReadingsByDate/02˙06/PDPVolIChapter8.pdf

[12] Kingma, D., Ba, J. *Adam: A Method for Stochastic Optimization*. 2015 [cit. 4.6.2019], Dostupné z: https://goo.gl/Un8Axa

Příloha A

Kód pro MNIST

A.1 Načtení a úprava dat

```
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Conv2D, Dropout, Flatten, MaxPooling2D,
    Activation
from keras.datasets import mnist
import keras

epochy = 50

#načtení datasetu
(x_train,y_train),(x_test,y_test)=mnist.load_data()

# převedení na tvar (50000,28,28,1) a (10000,28,28,1)
x_train = x_train.reshape(x_train.shape[0], 28, 28, 1)
x_test = x_test.reshape(x_test.shape[0], 28, 28, 1)
#deklarování hodnot jako float
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')

#normalizace vstupních hodnot (0,255) => (0,1)
x_train /= 255
x_test /= 255
```

A.2 Sestavení modelu a trénování

```
# Vyvoření modelu
model = Sequential()
model.add(Conv2D(28, kernel_size=(3,3), input_shape=(28,28,1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.2))
model.add(Conv2D(56, kernel_size=(3,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(10))
model.add(Activation('softmax'))

#optimalizace + compilace modelu
opt = keras.optimizers.Adam(lr=0.001, beta_1=0.9,beta_2=0.999)
model.compile(optimizer=opt,
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
#trénování modelu
model.fit(x=x_train,y=y_train,validation_split=1/6 ,epochs=epochy,
    callbacks=[csv_log,timecall])
test_loss, test_acc = model.evaluate(x_test, y_test)
```

Příloha B

Kód pro CIFAR-10

B.1 Načtení a úprava dat

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten, Conv2D,
    MaxPooling2D
from keras.callbacks import CSVLogger

batch = 64
pocet_kategorii = 10
epochy = 150

# data rozdělená na testovací a trénovací:
(x_train, y_train), (x_test, y_test) = keras.datasets.cifar10.load_data()

#normalizace (0,255) => (0,1)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255

#převedení na one_hot matici
y_train = keras.utils.to_categorical(y_train, pocet_kategorii)
y_test = keras.utils.to_categorical(y_test, pocet_kategorii)
```

B.2 Sestavení modelu a trénování

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3), padding='same',input_shape
    =(32,32,3)))
model.add(Activation('relu'))
model.add(Conv2D(32, kernel_size=(3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.20))
model.add(Conv2D(64, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, kernel_size=(3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.50))
model.add(Conv2D(128, kernel_size=(3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(128, kernel_size=(3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.50))
model.add(Flatten())
model.add(Dense(pocet_kategorii))
model.add(Activation('softmax'))

#Optimalizace a compilace modelu
opt = keras.optimizers.Adam(lr=0.0001, beta_1=0.9,beta_2=0.999)
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])
#trénování
model.fit(x_train, y_train,
            batch_size=batch,
            epochs=epochy,
            callbacks=[csv_log,timecall],
            validation_split=1/5,
            shuffle=True)
```