Praktická úloha řešená konvoluční neuronovou sítí

Zadání č. 42
Jakub Svoboda

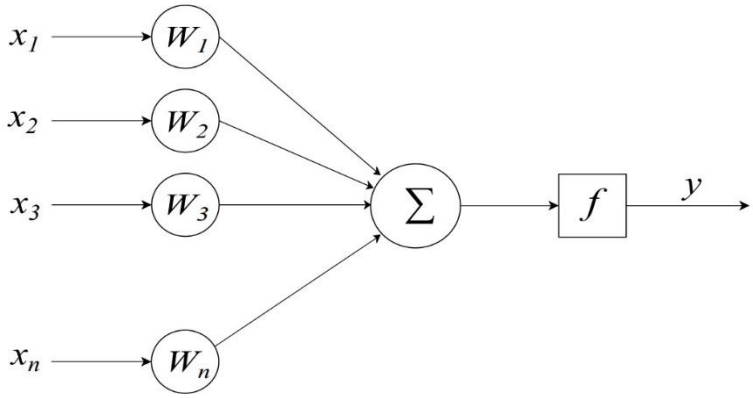
1. Úvod

Tento dokument slouží jako technická zpráva k projektu do předmětu Soft Computing na Fakultě informačních technologií VUT v Brně. V druhé kapitole je popsán teoretický základ fungování neuronových sítí, jsou zde popsány jednotlivé vrstvy a operace neuronových sítí a je zde také popsán proces učení. Ve třetí kapitole je popsána samotná implementace programu pro řešení problému tohoto projektu. Ve čtvrté kapitole je popsán proces trénování sítě pro splnění konkrétní úlohy a jsou zde také výsledky evaluace sítě. V páté kapitole je popsán postup instalace projektu na vlastní stroj, popřípadě popis jeho spuštění na serveru Merlin. V šesté kapitole jsou pak shrnuty dosažené výsledky tohoto projektu.

2. Teoretický úvod do problematiky

Umělé neuronové sítě jsou výpočetní struktury používané v oblasti umělé inteligence. Jejich vzorem jsou sítě mozkových buněk – neuronů v živých organismech. Neuronové sítě jsou schopny se naučit plnit zadaný úkol, aniž by na daný problém byly specificky programovány. Vytrénované neuronové sítě jsou schopny odhalit vzorce z velkého množství dat, mnohdy s větší úspěšností než lidé [1]. Úkony řešené algoritmy strojového učení lze obecně klasifikovat na učení s učitelem (*supervised learning*) a učení bez učitele (*unsupervised learning*). Při učení s učitelem má algoritmus k dispozici zpětnou vazbu, která indikuje, kdy systém pracuje správně a může také poskytovat informace o velikosti chyby [1]. Příkladem takového učení může být rozpoznávání obsahu obrazu, kdy se síť učí na předem označených (*labeled*) datech. Oproti tomu učení bez učitele spoléhá na heuristicky získané informace a systém sám vyhledává a rozpoznává vzorce v datech. Typickým problém řešeným učením bez učitele je shlukování (*clustering*). [11]

Umělý neuron je základním stavebním prvkem neuronových sítí. Umělý neuron (viz obrázek níže) je funkce s vektorem vstupů (ekvivalent dendritů v lidském mozku) a vektorem vah. Při výpočtu jsou váhy a vstupní hodnoty navzájem vynásobeny. K sumě těchto hodnot je pak přičtena skalární hodnota – bias



Obrázek 1: Model umělého neuronu s vektorem vstupů $x = (x_1, x_2, x3, \ldots, x_n)$. Každá vstupní hodnota je vynásobena váhovým koeficientem $W = (W_1, W_2, W_3, \ldots, W_n)$. Suma těchto hodnot je pak předána aktivační funkci f, která vypočítá výstupní hodnotu y.

a výsledek je předán aktivační funkci, po jejímž výpočtu je výsledná hodnota předána z neuronu dále, obvykle do další vrstvy neuronů. Pokud se jedná o poslední vrstvu, je vyhodnocen výsledek. [1]. [11]

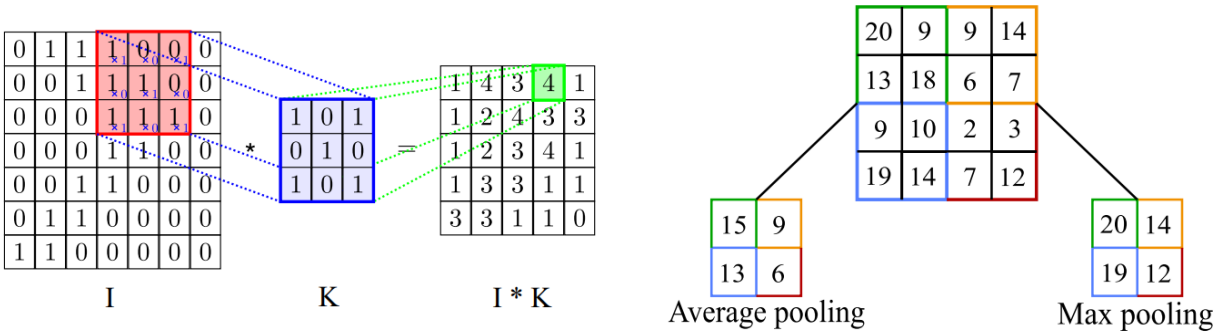
Vícevrstvé neuronové sítě jsou sítě skládající se ze dvou nebo více vrstev neuronů. První vrstva sítě přijímá vektorizovaná data a předává je do hlubších vrstev. Vrstvy mezi první a poslední vrstvou se nazývají skryté. Data, se kterými skryté vrstvy pracují, jsou směrem od vstupu více abstraktní a ke konci sítě lze pomocí příznaků definovat i velmi abstraktní datové struktury. Pokud má neuronová síť více než jednu skrytou vrstvu, hovoříme o ní jako o hluboké neuronové síti (*Deep Neural Network*). [11]

Plně propojená vrstva (*fully connected layer*, někdy také *dense layer*) je vrstva neuronů, kde každý neuron přijímá hodnoty ze všech neuronů předchozí vrstvy, nebo ze vstupního vektoru dat. Sítě skládající se pouze z plně propojených vrstev se nazývají plně propojené neuronové sítě a jsou nejjednodušším příkladem neuronových sítí [1]. Ačkoliv jsou sítě s plně propojenými vrstvami schopny dobře aproximovat libovolnou funkci, vysoký počet spojení zvyšuje nároky na výpočetní kapacity. Proto jsou u detekčních sítí využívány zřídka a spíše se vyskytují v koncových vrstvách sítě, kde mají na starosti klasifikaci objektů. [11]

Konvoluční vrstva je jedním z nejdůležitějších stavebních prvků konvolučních neuronových sítí. Její základní matematická operace je konvoluce. Tu lze popsat jako operaci nad dvěma funkcemi [2], kde první argument konvoluce je obvykle označován jako vstup (*input*) a druhý argument jako konvoluční jádro (*kernel*). Ve strojovém učení jsou vstupem a konvolučním jádrem často vícerozměrná pole, která se označují souhrnným názvem *tenzor*. V neuronových sítích je konvoluce často uplatňována na více než jedné ose. [11]

Parametry konvoluční vrstvy se skládají z množiny konvolučních jader, která obvykle pracují se všemi barevnými kanály. Samotná síť se učí optimalizací parametrů těchto konvolučních jader (konvolučních filtrů). Filtry jsou po vstupu přesouvány po určitém kroku (*stride*) a na příslušných místech je spočítán hadamatův součin parametru a filtru. Je tak vytvořena dvoudimenzionální mapu, která zobrazuje místa, kde došlo k aktivaci [3]. Znázornění operace konvoluce je na obrázku níže. Výpočet konvoluce je problematický v okrajových částech obrazu. V těchto případech nemusí být konvoluce vypočítána tam, kde by konvoluční filtr přesahoval okraj vstupu (tzv. plná konvoluce). Tím je způsobeno zmenšení obrazu dané velikostí filtru. Alternativou je počítat konvoluci s nulovými body za hranicí obrazu (tzv. *zero padding*), popřípadě s jinou, vhodně vybranou hodnotou. [11]

Pooling vrstva agreguje výstupy sousedních neuronů do jednoho, čímž dochází k podvzorkování a zmenšení rozměrů aktivační mapy. Tím lze výpočetně zjednodušit operace, které následují po této vrstvě. V praxi je běžné umístění pooling vrstvy vždy za konvoluční vrstvou [4]. Způsob agregace hodnot je například výpočet průměrné hodnoty, výběr minima nebo nejčastější výběr maxima (*max pooling*). Počet hodnot k agregaci je volitelný, je také možné využít překrývající pooling vrstvu, u které bylo empiricky dokázáno, že snižuje chybovost a náchylnost sítě k přetrénování [4]. [11]



Obrázek 2: Vlevo - Konvoluce vstupu I a jádra K [5]. Vpravo - ukázka operací Average Pooling a Max Pooling.

Aktivační funkce neuronu slouží pro výpočet výstupní hodnoty pro daný vektor vstupních hodnot. Volbou vhodné aktivační funkce lze významně urychlit výpočty při trénování neuronových sítí a pomoci síti ke konvergenci. Hodnota vypočtená na každém neuronu je předána aktivační funkci. Celý proces tak lze vyjádřit vztahem $a_j = \sigma(w_{i,j} * x_i + b_j)$, kde σ značí chybovou funkci, w_i jsou váhové koeficienty, x_i jsou vstupní hodnoty do neuronu a b_j značí bias. [11]

Sigmoid je aktivační funkce s nebinárním výstupem z intervalu (0, 1). Funkce a její kombinace jsou nelineární. Tyto vlastnosti činí sigmoid vhodnou funkcí pro využití v hlubokých neuronových sítích. Sigmoid lze definovat předpisem $f(x) = 1/(1+e^{-x})$. Pro vstupní hodnoty z intervalu $(-2, 2)$ strmě roste hodnota na ose y, což znamená, že funkce tíhne ke stranám křivky a je proto vhodná pro využití u klasifikace. Na tuto vlastnost se váže i několik problémů. Derivace se v bodech více vzdálených od středu osy x blíží nule a síť se z takto malých hodnot derivace nedokáže v rozumném čase učit. Tento problém je známý jako *vanishing gradient* [6] a je řešitelný buď úpravou výpočtu [7] nebo jednoduše použitím rychlejšího hardware. [11]

Rectified Linear Unit je aktivační funkce, jejíž výstup je roven x pro kladné hodnoty a nule pro záporné. Samotná funkce i její kombinace jsou nelineární. ReLU je navíc oproti sigmoidu nebo hyperbolickému tangentu méně výpočetně náročná, při náhodně inicializovaných vahách by v síti bylo 50 % neuronů, které by se díky záporné hodnotě neaktivovaly, stejně tak samotné matematické operace jsou jednodušší. Ačkoliv je chování biologického neuronu blíže sigmoidu [8], v umělých neuronových sítích je ReLU, díky úspoře času potřebného na výpočet, využívána častěji než sigmoid nebo hyperbolický tangent. Funkce ReLU je definována předpisem $f(x) = max(x, 0)$. [11]

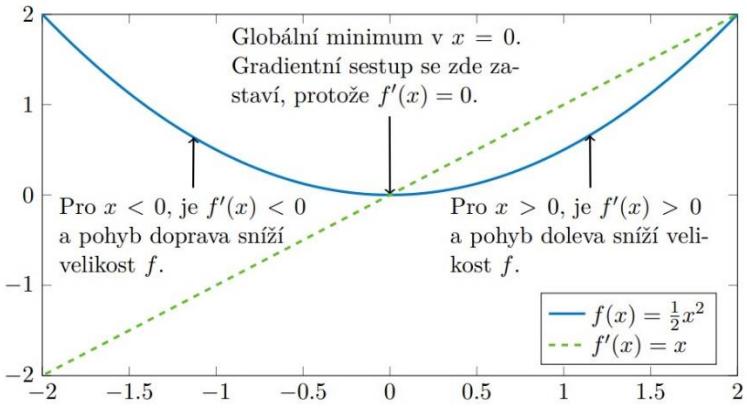
Pro záporné hodnoty x je gradient funkce ReLU nulový, což má za následek, že neuron může při učení přestat reagovat na podněty (tzv. Dying ReLU problem [8]). V praxi se proto často využívá obměna funkce ReLU, pro kterou jsou výstupy pro záporné hodnoty pouze velmi blízké nule (např. $y = 0{,}01x$ pro $x < 0$). Takto modifikovaná funkce se nazývá Leaky ReLU. [11]

Funkce softmax se využívá v posledních vrstvách neuronových sítí, obvykle tam, kde síť řeší klasifikační problém, například přiřazení objektu do předem definovaných tříd, které nejsou vzájemně výlučné. Její použití je také vhodné z důvodu, že součet výstupních hodnot bude roven jedné, což je vhodné v případech, kdy chceme výstupem vyjádřit pravděpodobnost, že vstup spadá do jedné z kategorií a celkový součet pravděpodobností by tedy měl byt roven jedné. [11]

Pro **učení neuronové sítě** je nutná existence algoritmu, který provede změnu vah a biasů tak, aby při vstupu vzorku dat z datasetu do sítě odpovídal výstup očekávanému výsledku. Ke zjištění toho, do jaké míry se výstup sítě liší od ideálního výsledku očekávaného u trénovacích dat, se využívá chybových funkcí. Z jejich výstupu (*loss*) je při trénování určeno, jakým směrem a do jaké míry je nutné změnit váhové hodnoty neuronů. Správný průběh trénování neuronové sítě lze tak pozorovat z hodnoty chybové funkce, která by se měla po každé iteraci snížit a síť by tak měla stále lépe aproximovat funkci specifickou pro daný úkol. [11]

Gradient descent je iterativní, optimalizační algoritmus pro hledání minima funkce. Chyba funkce se zvětšuje ve směru nejvyššího gradientu (∇G), při hledání minima tedy algoritmus postupuje po krocích proti směru největšího gradientu. Velikost kroku se zmenšuje proporcionálně s velikostí gradientu. Pro změnu vah v neuronech se používá předpis $w_{i+1} = w_i - \eta \nabla G(w_i)$, kde w značí váhu, η je velikost učícího kroku (*learning rate*), udávající rychlost změny vah. ∇G je gradient pro aktuální iteraci gradiálního sestupu. Algoritmus je ukončen v momentě, kdy je změna vah menší než požadovaná hodnota. Stochaický gradientní sestup využívá pro změnu vah pouze jeden nebo několik vzorků dat (*mini-batch*), což má za

následek snížení výpočetní náročnosti. Stochaický gradient je spolu se zpětným šířením chyby nejpoužívanějším algoritmem pro trénování neuronových sítí [1]. [11]



Obrázek 3: Ukázka algoritmu Gradient Descent při hledání minima funkce [9].

Backpropagation (zpětné šíření chyby) je adaptační algoritmus, pomocí kterého lze vypočítat podíl jednotlivých neuronů na celkové chybě a upravit váhy jednotlivých neuronů tak, aby byla minimalizována chyba. Jedná se o nejrozšířenější adaptační algoritmus, který je používaný v přibližně 80 % všech aplikací neuronových sítí [10]. Samotný algoritmus se skládá ze tří, neustále se opakujících fází. Nejprve je vstupní signál šířen sítí dopředu (*feedforward*), následně je chyba šířena v opačném směru, a nakonec jsou aktualizovány jednotlivé váhy včetně biasů na spojeních neuronů [10]. Propagací chyby zpět do sítě dochází ke změně vah neuronů proti směru gradientu tak, aby se chybová funkce blížila minimu. Parametry sítě se upraví v závislosti na učícím kroku (*learning rate*) a velikosti gradientu, čímž se síť trénuje. Velikost učícího kroku se může v průběhu procesu učení měnit, například skokovým zmenšením po dosažení určitého počtu iterací. [11]

3. Implementace

Jako zadaný problém pro neuronovou síť byla zvolena klasifikace obrazových dat. Konkrétně byla síť natrénována a testována pro klasifikaci číslic z datasetu MNIST, nicméně síť je schopná pracovat s libovolným datasetem. Implementace byla realizována bez použití frameworků pro pracování s neuronovými sítěmi a z tohoto důvodu je doporučeno pracovat se vstupními daty menších rozměrů. U větších vstupních dat se výpočty sítě značně prodlužují z důvodu nízké optimalizace a absence akcelerace výpočtů na GPU.

Implementace samotné sítě se nachází v souboru *network.py*. Tento soubor obsahuje funkce pro načtení a uložení natrénovaného modelu na disk. Z důvodu delšího trénovacího času je vhodné využít předtrénovaný model uložený na disku. V souboru se dále nachází samotná třída zahrnující síť (*class Network()*), obsahující především odkaz na jednotlivé vrstvy a metody pro učení sítě.

Konvoluční vrstva je implementována s kernelem s rozměry 3x3, pro krok konvoluce byla zvolena hodnota jedna. Počet filtrů vrstvy lze nastavit parametrem v konstruktoru vrstvy. Konvoluční vrstva v této implementaci nevyužívá padding, data jsou tak při dopředném průchodu zmenšena. Pro dataset MNIST je rozměr vstupního vektoru 28x28. Implementaci vrstvy lze nalézt ve třídě *Conv3x3* v souboru *network.py*.

Max Pooling vrstva je naimplementovaná s rozměry vstupních polí 2x2, velikost dat je tedy zmenšena na čtvrtinu (na polovinu ve dvou rozměrech). Tato vrstva samotná neobsahuje žádné váhy ani biasy, a není z tohoto důvodu trénovatelná. Po pooling vrstvě následuje ReLU aktivační funkce. Implementace této

vrstvy je ve třídě *MaxPool2x2* v souboru *network.py*, funkce ReLU je implementována v souboru *activations.py*.

Softmax vrstva je souhrnný název pro plně propojenou vrstvu následovanou aktivační funkcí softmax. Při dopředném průchodu jsou vynásobeny vstupní hodnoty s váhami a suma těchto hodno je předána aktivaci. Aktivace softmax zaručí, že výstupní hodnoty odpovídají pravděpodobnosti příslušnosti do dané třídy. Tato vrstva má deset neuronů (pro dataset MNIST), kde každý neuron odpovídá jedné třídě.

Obrázek 4: Architektura sítě.

4. Trénování a výsledky

Jako úloha demonstrující činnost sítě byla zvolena klasifikace číslic z datasetu MNIST. Tento dataset obsahuje 60000 obrazců číslic pro učení a 10000 pro testování. Testovací dataset byl dále rozdělen v poměru 9:1 na testovací a validační část. Při učení byla hodnota učícího kroku zvolena jako 0,005. Trénování proběhlo přes celý dataset s jednou epochou. Před trénováním byly hodnoty datasetu převedeny z rozsahu 0 až 255 na interval (-0,5; 0,5). Jelikož obsahuje dataset je jeden kanál, je vstup do sítě dvoudimenzionální.

Po trénování nad celým datasetem proběhla evaluace sítě (úspěšnost nad 10000 obrázky). Síť predikovala správnou číslici v 96,6 % případů. Tento model je uložen v souboru *model_proper.pckl* a tento soubor je také použit při spuštění programu bez parametru --train.

5. Instalace

Běh programu byl testován na serveru Merlin. Po rozbalení archívu je nutné stáhnout dataset MNIST (z důvodu velikosti není k projektu přiložen) a rozbalit jej do složky */dataset.* Spuštění programu potom z hlavní složky projektu provedete příkazem:

```
python3 main.py
```

což spustí program bez trénování. Dojde pouze k načtení modelu z disku a proběhne jeho evaluace. Pro trénování modelu od nuly spusťte program s parametrem --train a specifikujte počet epoch:

```
python3 main.py --train 10
```

Tento příkaz natrénuje síť během desíti epoch a výsledný model uloží na disk. Mějte na paměti, že při učení sítě na serveru Merlin delší trénování překročí maximální povolenou kvótu času na procesoru a server proces ukončí předčasně.

Pro rychlé testování a demonstrace je možné spustit i jednoduché demo s GUI. Jelikož Merlin nemá podporu pro GUI frameworky, doporučuji nainstalovat si dané knihovny na stůj PC:

```
python3 -m pip install PyQt5
python3 demo.py
```

Celý proces instalace a návod na spuštění je shrnut v souboru *readme.txt* přiloženém v archívu projektu.

6. Závěr

V tomto projektu byla vytvořena implementace konvoluční sítě bez použití frameworků. Výsledná síť byla natrénována na klasifikaci číslic z datasetu MNIST. Při evaluaci dosáhla úspěšnosti v 96,6 procentech případů.

7. Reference

[1] Mehrotra, K.; Mohan, C. K.; Ranka, S.: Elements of Artificial Neural Networks. Cambridge, MA, USA: MIT Press, 1997, ISBN 0-262-13328-8.

[2] LeCun, Y.; Bengio, Y.; Hinton, G.: Deep learning. Nature, ročník 521, 2015, [online]. URL http://graveleylab.cam.uchc.edu/WebData/mduff/MEDS_6498_SPRING_2016/deep_learning_nature_2015.pdf

[3] Karpathy, A.: Convolutinal Neural Networks for Visual Recognition. 2016, [online]. URL http://cs231n.github.io/convolutional-networks/

[4] Krizhevsky, A.; Sutskever, I.; Hinton, G. E.: ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems 25, editace F. Pereira; C. J. C. Burges; L. Bottou; K. Q. Weinberger, Curran Associates, Inc., 2012, s. 1097–1105, [online]. URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deepconvolutional-neural-networks.pdf

[5] Velikovič, P.: 2D Convolution. [online], URL https://github.com/PetarV-/TikZ/tree/master/2D%20Convolution

[6] Hochreiter, S.: The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions. Int. J. Uncertain. Fuzziness Knowl.-Based Syst., ročník 6, č. 2, Duben 1998: s. 107–116, ISSN 0218-4885, doi:10.1142/S0218488598000094, [online]. URL http://dx.doi.org/10.1142/S0218488598000094

[7] Pascanu, R.; Mikolov, T.; Bengio, Y.: Understanding the exploding gradient problem. CoRR, ročník abs/1211.5063, 2012, [online], 1211.5063. URL http://arxiv.org/abs/1211.5063

[8] Xu, B.; Wang, N.; Chen, T.; aj.: Empirical Evaluation of Rectified Activations in Convolutional Network. CoRR, ročník abs/1505.00853, 2015, [online], 1505.00853. URL http://arxiv.org/abs/1505.00853

[9] Ian Goodfellow, Y. B.; Courville, A.: Deep Learning, 2016, book in preparation for MIT Press, [online]. URL http://www.deeplearningbook.org

[10] Volná, E.: Neuronové sítě 1. 2008, [online]. URL http://www1.osu.cz/~volna/Neuronove_site_skripta.pdf

[11] SVOBODA, Jakub. Detektor hlavy v obraze. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Goldmann