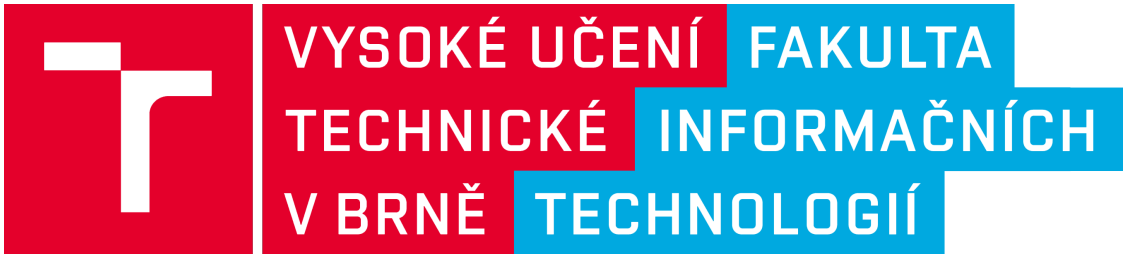


Soft Computing 2023/2024

Demonstrace učení BP - základní algoritmus + vybraný optimalizátor

David Chocholatý (xchoch09)

Brno, 27. listopadu 2023

1 Úvod

Tato dokumentace popisuje projekt s tématem zadání *Demonstrace učení BP - základní algoritmus + vybraný optimalizátor*. Mezi její hlavní stěžejní části patří úvod do problematiky, popis významných implementačních oblastí, návod pro spuštění aplikace a představení práce se samotnou vytvořenou aplikací.

2 Algoritmus zpětného šíření chyby a gradientní sestup

Algoritmus *zpětného šíření chyby* je nejznámějším a nejpoužívanějším algoritmem pro nastavování vah především acyklických dopředných *neuronových sítí*. Cílem tohoto algoritmu je minimalizovat *objektivní funkci*, která je funkcí všech *vah* sítě, a která vyjadřuje odchylky odezvy sítě od požadovaných hodnot.[1] Chyba vyjádřená pomocí objektivní funkce je minimalizována pomocí metody *gradientního sestupu*:

$$\nabla \vec{w} = -\mu \nabla E_p,$$

kde ∇E_p vyjadřuje gradient chyby E_p a μ *koeficient chyby učení*.

Při změně vah je možné namísto klasického přístupu využívat také *optimalizátory*. Ty budou blíže popsány v podsekci 3.1.6.

3 Popis stěžejních částí projektu

Tato sekce popisuje významné implementační části projektu, a to především knihovnu pro práci s neuronovými sítěmi se základní funkcionalitou. Následně je popsána architektura neuronové sítě pro vybraný klasifikační problém.

3.1 Implementace základní knihovny pro práci s neuronovými sítěmi

Následující podsekce popisuje veškeré stěžejní části implementace základní knihovny pro práci s neuronovými sítěmi.

3.1.1 Architektura knihovny

Mezi jednu z nejdůležitějších součástí projektu patří implementace základní knihovny pro práci s neuronovými sítěmi. Tato knihovna implementuje veškeré důležité komponenty neuronové sítě. Architektura knihovny je navržena s využitím *automatické diferenciace*, přičemž tento návrh je známý také jako pod anglickým názvem *autograd engine*[2]. V podobném stylu je například konstruována mnohem rozsáhlejší knihovna, co se týče funkcionality, a to knihovna PyTorch[3], která se v současnosti řadí k nejvyužívanějším knihovnám pro práci s neuronovými sítěmi. Při implementaci knihovny byly použity některé části knihovny Tensorgrad[4], která je založena na knihovně micrograd[5]. Inspirací pro vytvoření dané aplikace byl již publikovaný projekt

[1] https://www.fit.vutbr.cz/study/courses/SFC/private/zboril2020/20sfc_2.pdf
[2] https://pytorch.org/blog/overview-of-pytorch-autograd-engine/
[3] https://pytorch.org/
[4] https://github.com/hkxIron/tensorgrad/tree/master
[5] https://github.com/karpathy/micrograd

na dané téma[6], ovšem při implementaci funkcionality knihovny a modelu byl zvolen odlišný přístup, implementováno více operátoru a zvýšena uživatelská přívětivost aplikace.

3.1.2 Automatická diferenciace

Při automatické diferenciaci se na základě tzv. *výpočtového grafu* počítají gradienty vstupů, přičemž lze provést *dopředný průchod* (anglicky *forward pass*) nebo *zpětný průchod* (anglicky *backward pass*), který se provádí až po dopředném průchodu. Výpočet gradientů začíná od výstupu. Automatická diferenciace se opírá o klasický výpočetní vzorec známý jako *řetězcové pravidlo* (anglicky *chain rule*). To umožňuje počítat složité derivace jejich rozdělením a pozdější rekombinací, formálně zapsáno jako

$$\frac{\partial}{\partial x} f(g(x)) = f'(g(x))g'(x).$$[7]

3.1.3 Implementace neuronové sítě

Mezi hlavní implementované entity knihovny pro práci s neuronovými sítěmi patří *vícevrstvý umělý neuron* (anglicky *multilayer perceptron*) obsahující *vrstvy* (anglicky *layer*) s jednotlivými *umělými neurony* vrstvy (anglicky *perceptron*).

Základní entitou, která reprezentuje a obsahuje data, se kterými se manipuluje, je *tenzor* (anglicky *Tensor*). Ten také obsahuje jednotlivé vztahy pro výpočty při dopředném a zpětném průchodu výpočtovým grafem pro jednotlivé základní operace, jako je součet, součin, ale i *aktivační funkce* jako *ReLu* nebo *Softmax*, které jsou blíže popsány v následující podsekci 3.1.4.

3.1.4 Aktivační funkce

Při vytvoření požadovaného modelu jsou zapotřebí dvě aktivační funkce, a to již zmíněné *ReLu* a *Softmax*. Tyto aktivační funkce jsou počítány dle následujících vztahů:

- ReLu[8]
  - Dopředný průchod:

$$f(x) = \begin{cases} x, & \text{jestliže } x > 0, \\ 0, & \text{jinak.} \end{cases}$$

  - Zpětný průchod:

$$f'(x) = \begin{cases} 1, & \text{jestliže } x > 0, \\ 0, & \text{jestliže } x < 0. \end{cases}$$

- Softmax[9,10]

[6] https://github.com/RichardKlem/SFC/tree/main
[7] https://pytorch.org/blog/overview-of-pytorch-autograd-engine/
[8] https://en.wikipedia.org/wiki/Rectifier_(neural_networks)
[9] https://www.pinecone.io/learn/softmax-activation/
[10] https://www.mldawn.com/the-derivative-of-softmaxz-function-w-r-t-z/

– Dopředný průchod:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}}.$$

– Zpětný průchod:

$$\frac{\partial}{\partial x_j} softmax(x_i) = \begin{cases} softmax(x_i)(1 - softmax(x_i)), & \text{jestliže } i = j, \\ -softmax(x_i)softmax(x_j), & \text{jestliže } i \neq j. \end{cases}$$

3.1.5 Ztrátová funkce

Jako *ztrátová funkce* pro navržený model je použita *vícetřídní křížová entropie* (anglicky *multi-class cross-entropy*). Tato funkce je definována následovně:

$$H(P, Q) = -\sum_{x \in \mathcal{X}} p(x) \log q(x)^{11},$$

kde q značí odhadnuté rozdělení pravděpodobnosti, p skutečné rozdělení pravděpodobnosti, P a Q příslušnou funkci hustoty pravděpodobnosti a $\mathcal{X}$ nosič, přičemž x značí jednotlivá označení tříd. Konkrétně $p(x)$ nabývá hodnoty 0 nebo 1 podle toho, zda označení třídy x je správnou klasifikací.[12]

3.1.6 Optimalizátory

Vytvořená aplikace implementuje několik vzájemně souvisejících optimalizátorů, a to *RMSprop*, *Adam* a *AMSGrad*. Oproti původnímu znění zadání jsou další dva operátory implementovány jako rozšíření. Veškeré vzorce jsou převzaty z oficiálních materiálů předmětu *SFC*[13].Následně jsou uvedeny vztahy pro výpočty jednotlivých optimalizátorů:

RMSprop optimalizátor

$$\nabla \vec{w} = -\mu \nabla E = -\frac{\alpha}{\sqrt{\nu + \varepsilon}} \nabla E,$$
$$\nu = \beta \nu^{old} + (1 - \beta)(\nabla E)^2,$$

kde $\nu^0 = 0$ a $\alpha = 0.001$, $\varepsilon = 10^{-6}, \beta = 0.9$.

Adam optimalizátor

$$\nabla \vec{w} = -\frac{\alpha}{\sqrt{\hat{\nu}} + \varepsilon} \vec{m},$$
$$\vec{m} = \beta_1 \vec{m}^{old} + (1 - \beta_1) \nabla E,$$
$$\hat{\nu} = \frac{\nu}{1 - \beta_2},$$
$$\nu = \beta_2 \nu^{old} + (1 - \beta_2)(\nabla E)^2,$$

kde $\vec{m}^0 = \vec{0}, \nu^0 = 0$ a $\alpha = 0.001, \beta_1 = 0.9$, $\beta_2 = 0.999, \varepsilon = 10^{-7}$.

AMSGrad optimalizátor

$$\nabla \vec{w} = -\frac{\alpha}{\sqrt{\hat{\nu}} + \varepsilon} \vec{m},$$
$$\vec{m} = \beta_1 \vec{m}^{old} + (1 - \beta_1) \nabla E,$$
$$\boldsymbol{\hat{\nu} = max(\hat{\nu}^{old}, \nu)},$$
$$\nu = \beta_2 \nu^{old} + (1 - \beta_2)(\nabla E)^2,$$

kde $\vec{m}^0 = \vec{0}, \nu^0 = 0$ a $\alpha = 0.001, \beta_1 = 0.9$, $\beta_2 = 0.999, \boldsymbol{\varepsilon = 10^{-8}}$.

[11] https://en.wikipedia.org/wiki/Cross-entropy

[12] https://www.oreilly.com/library/view/hands-on-convolutional-neural/9781789130331/7f34b72e-f571-49d2-a37a-4ed6f8011c93.xhtml

[13] https://www.fit.vut.cz/study/course/268371/.en

3.2 Model

Jako demonstrační úloha pro představení vlastností jednotlivých optimalizátorou byla zvolena jedna z nejznámějších úloh z oblasti strojového učení, a to úloha klasifikace ručně psaných číslic do deseti tříd s využitím nejznámějšího datasetu *MNIST*. Tento dataset byl v redukované formě převzat z platformy Kaggle[14]. Uvedený dataset ovšem narozdíl od obrázků obsahuje uložené hodnoty jednotlivých pixelů (soubor ve formátu *.csv*), a tudíž je možné vytvořit jednoduchou architekturu modelu oproti potřebě využití například *konvolučních* neuronových sítí.

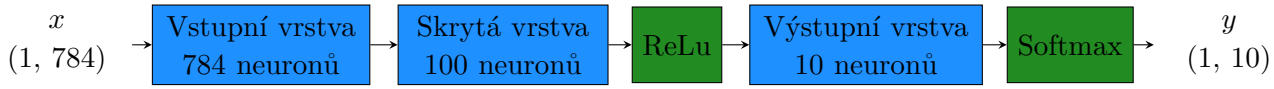
Samotný gradientní sestup je realizován *stochastickým přístupem* (anglicky *stochastic gradient descent — SGD*). Při implementaci je také za účelem předcházení tzv. *přeučení* použita *L2 regularizace*.

Pro všechny vrstvy jsou výstupy neuronů, i před případnou aplikací aktivační funkce, počítány dle vztahu

$$y = \sum_i w_i x_i + b,$$

kde x_i značí i-tý vstup neuronu, w_i váhu příslušející i-tému vstupu x_i a b tzv. *bias*.

Architektura vytvořeného modelu je vyobrazena na obrázku 1.



Obrázek 1: Architektura navrženého modelu pro klasifikaci ručně psaných číslic s využitím datasetu MNIST. Jelikož původní rozměr obrázku je 28 pixelů $*$ 28 pixelů $=$ 784 pixelů, vstupní vrstva obsahuje 784 neuronů, přičemž se jedná o lineární vrstvu. Následující a jediná skrytá vrstva je vrstva se sto neurony využívající aktivační funkci ReLu. Jako poslední výstupní vrstva obsahuje deset neuronů, jelikož je možné klasifikovat deset číslic v rozsahu 0–9, a využívá aktivační funkci Softmax.

4 Návod na spuštění aplikace

Následující sekce obsahuje návod pro spuštění a ovládání vytvořené aplikace. Nejprve jsou uvedeny používané knihovny, dále popsána funkcionalita skriptu pro instalaci knihoven a spuštění aplikace. V poslední řadě je uveden základní popis práce s aplikací.

4.1 Využívané knihovny

Vytvořená aplikace byla implementována v jazyce Python 3 a je otestována s verzí jazyka Python 3.10.12. Dále jsou vyžadovány pro správnou funkcionalitu následující knihovny: NumPy[15], PyQt5[16] a matplotlib[17].

[14] https://www.kaggle.com/competitions/digit-recognizer/data
[15] https://numpy.org/
[16] https://www.riverbankcomputing.com/software/pyqt/
[17] https://matplotlib.org/

4.2 Skript pro instalaci knihoven a spuštění programu

Pro instalaci uvedených knihoven a následné spuštění programu byl vytvořen skript s názvem *run.sh* nacházející se v kořenové složce projektu. Pro validní funkcionalitu skriptu je vyžadována instalace balíčku[18] pro Python virtuální prostředí (pro Python 3.10 balíček *python3.10-venv*). Na operačním systému Ubuntu je instalace je možná pomocí příkazu:

```
$ apt install python3.10-venv
```

Implementovaný skript nejprve vytvoří Python virtuální prostředí (anglicky *virtual enviroment*) a následně jej aktivuje. Poté je provedena instalace požadovaných knihoven a spuštěna samotná aplikace. Pro případné odstranění vytvořeného virtuálního prostředí lze zadat následující příkaz v kořenové složce projektu:

```
$ rm -rf xchoch09\_venv
```

Správná funkcionalita ověřena na serveru merlin[19] a operačním systému *Ubuntu 22.04.3 LTS*.

4.3 Popis ovládání aplikace

Tato podsekce popisuje ovládání jednotlivých prvků aplikace. Následující výčet popisuje práci s každým elementem aplikace:

- Počet epoch (anglicky *number of epochs*) — hodnota udává počet provedených iterací (*epoch*) při výpočtu *ztrát* (anglicky *loss*) pro všechny optimalizátory. Zadaná hodnota může být v rozsahu 1–100.
- Koeficient učení (anglicky *learning rate*) — hodnota reprezentuje koeficient učení při výpočtu gradientního sestupu pro všechny optimalizátory. Zadaná hodnota může být v rozsahu 0.00001–1,0.
- Optimalizátory (anglicky *optimizers*) — tato sekce nabízí výběr vyhodnocovaných optimalizátorů a jejich zobrazovaných ztrát. Možná je libovolná kombinace několika optimalizátorů z následujících čtyř variant: *No optimizer*, *RMSprop optimizer*, *Adam optimizer* a *AMSGrad optimizer*.
- Spuštění měření a resetování parametrů — spuštění měření a vyhodnocování pro zvolenou konfiguraci je možné pomocí tlačítka *Start*. Následně jsou s výpočty jednotlivých ztrát pro vybrané optimalizátory vynášeny do grafu vypočítané hodnoty. Po skončení běhu vyhodnocování je možné aplikaci spustit znovu se stejnými parametry nebo pomocí tlačítka *Reset* vrátit výchozí nastavení parametrů a vyčištit graf.

[18] Na serveru *merlin* je již balíček nainstalován.
[19] merlin.fit.vutbr.cz