**Matematika a implementace neuronových sítí – synopse**

**Petr Jeřábek, Gymnázium Jana Nerudy**

**1. Rosenblattův perceptron**

**1.1 Inspirace a historický kontext**

Umělé neuronové sítě jsou inspirovány skutečným uspořádáním nervové soustavy. Koncept fungování biologického neuronu představili v roce 1943 Warren McCulloch a Walter Pitts s cílem navrhnout fungující umělou inteligenci. Tento model nese název *MCP* *neuron* (*McCulloch-Pitts neuron*). O pár let později publikoval Frank Rosenblatt na základě MCP neuronu první koncept nejjednoduššího modelu jednovrstvé neuronové sítě, takzvaný perceptron.

**1.2 Formální definice**

Perceptronový algoritmus funguje dobře zejména při řešení problému binární klasifikace. Pro každý příklad z datového souboru vypočítá perceptron lineární kombinaci (*lineární vstup* ) vektoru *příznaků* a vektoru *váhových koeficientů* . Predikovaná třída je poté určena skokovou *aktivační funkcí* na základě stanoveného *prahu* :

Negativní hodnotu prahu nazýváme *bias*. Rovnici pro výpočet lineárního vstupu můžeme zapsat v kompaktní vektorové formě:

**; ;**

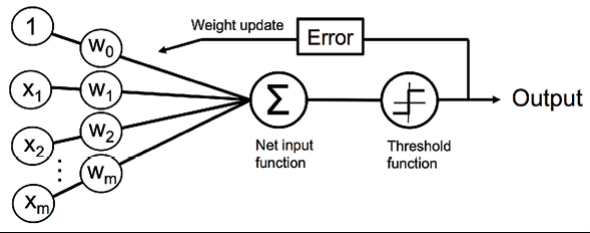
**1.3 Optimalizace váhových koeficientů**

Cílem všech neuronových sítí je optimalizovat váhové koeficienty pro co nejlepší výsledky. Vektor těchto koeficientů je na začátku trénování generován zcela náhodně v rámci určeného rozmezí (například 0 až 1). Pro výpočet aktualizované hodnoty váhy používá perceptron následující rovnici:

kde je stanovená *rychlost učení*, je *cílová třída*aznačí *predikovanou třídu*. Váhy jsou postupně aktualizovány během trénovacích iterací, takzvaných *epoch*. Formálně můžeme aktualizaci každého váhového koeficientu zapsat jako:

Pořadí trénovacích příkladůje typicky určeno náhodně pro každou epochu, aby nedocházelo k *přetrénování* (*overfitting*) sítě.

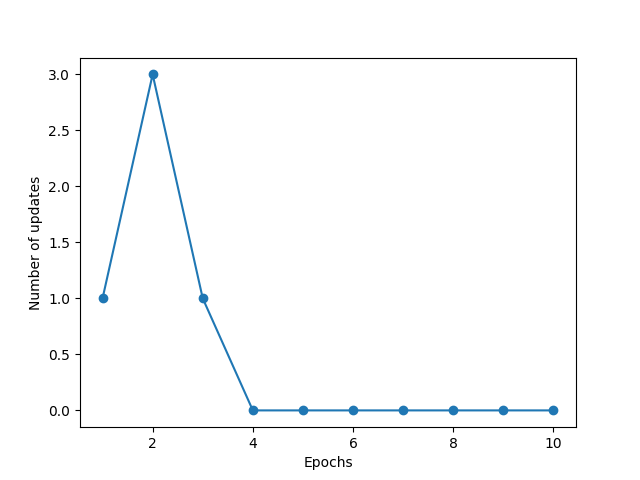
**1.4 Souhrn**

Cílem perceptronu je aplikovat jisté učební pravidlo a s jeho pomocí optimalizovat váhové koeficienty tak, aby dosahoval co nejlepších výsledků při klasifikaci vstupních dat. Tento proces se nazývá *učení* (odsud pojem *strojové/hluboké učení*) nebo *trénování*. Model by měl po natrénování (dovršení počtu epoch nebo dosažení určené přesnosti) naučený řešit problém binární klasifikace i na příkladech, které nebyly obsaženy v trénovací množině dat (k validaci slouží testovací datový soubor). Poskytnutá data musí být lineárně separovatelná úplné oddělení dat v různých výstupních třídách lineárním útvarem, například přímkou ve 2D případě), aby byl perceptron schopen příklady klasifikovat. Schéma tohoto algoritmu znázorňuje obrázek 1.1.

Obr. 1.1 Schéma perceptronového algoritmu

**1.5 Implementace**

Implementace perceptronu v Pythonu je velmi jednoduchá, jelikož se jedná pouze o jeden neuron. Kompletní kód s komentáři je k dispozici zde: <https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/Perceptron/Perceptron.py> (1). K otestování algoritmu byl použit datový soubor Iris, který obsahuje míry okvětních lístků tří druhů kosatce (jelikož se však jedná o binární klasifikaci, relevantní jsou pouze dva druhy – každý je zastoupen 50 příklady). Jak ukazuje graf na obrázku 1.2, perceptron dosáhl cíle a přestal optimalizovat váhové koeficienty ve čtvrté epoše. Natrénovaný model poté správně klasifikoval všech 10 příkladů z testovacího souboru. Algoritmus byl otestován tímto scriptem: <https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/Perceptron/perceptron_iris_binary.py> (2).



Obr. 1.2 Graf znázorňující počet aktualizací vah v každé epoše

**2. ADALINE**

**2.1 Inspirace a historický kontext**

Adaptivní lineární neuron (ADALINE) je podstatně vylepšená verze perceptronu. Stále se jedná o jednovrstvou síť, avšak proces učení je daleko efektivnější a přesnější. Jako podklad pro ADALINE sloužil právě model McCulloch-Pitts neuronu, na základě kterého navrhl Rosenblatt svůj perceptron.

**2.2 Lineární aktivační funkce**

Zatímco u perceptronu jsou váhové koeficienty aktualizovány podle rozdílu mezi cílovou a predikovanou třídou, ADALINE nejdříve zmenší jistou ztrátovou funkci úpravou vah pomocí vhodného algoritmu, a teprve poté vytváří predikci. Na lineární vstup navíc aplikuje lineární aktivační funkci, která je pouze identickou funkcí lineárního vstupu:

Využití lineární aktivační funkce namísto skokové je důležité hlavně proto, že je prostá, což je jedna z klíčových podmínek při aplikaci optimalizace založené na gradientu.

**2.3 Ztrátová funkce**

*Ztrátová* nebo *účelová funkce* (také *loss* nebo *cost*) je jednou z hlavních komponent všech neuronových sítí. Zjednodušeně se jedná o funkci určující ztrátové skóre*,* které představuje momentální míru chybovosti sítě na základě hodnot váhových koeficientů. Během učení se snažíme ztrátové skóre minimalizovat a docílit tak co nejlepších predikcí. V případě ADALINE definujeme ztrátovou funkci jako *sumu kvadratických odchylek* (*sum of squared errors*) SSE, kde je cílová třída příkladu a aplikace lineární aktivační funkce na lineární vstup :

Tato je funkce je diferencovatelná (má v každém bodě derivaci)a konvexní (má globální minimum).

**2.4 Gradientní sestup**

Chart, surface chart

Description automatically generatedAlgoritmus *gradientního sestupu* (*gradient descent*) GDje velmi efektivní a populární způsob optimalizace váhových koeficientů. Jeho cílem je minimalizovat ztrátové skóre a nalézt tak co nejvýhodnější hodnoty vah. Abychom mohli na funkci aplikovat gradientní sestup, musí být diferencovatelná a konvexní (výše definovaná SSE obě podmínky splňuje). Optimalizační algoritmuspoté upravuje váhové koeficienty v opačném směru gradientu ztrátové funkce (přibližuje se tak globálnímu minimu aby nalezl nejoptimálnější hodnoty vah – ztrátové skóre bude v tomto bodě nejnižší). Obr. 2.1 a 2.2 vizuálně znázorňují tento princip.

Obr. 2.1 Model znázorňující gradientní sestup ve 3D

Obr. 2.2 Znázornění gradientního sestupu ve 2D

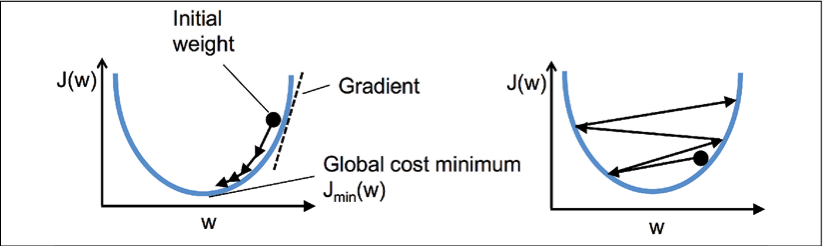
**2.5 Optimalizace váhových koeficientů**

Gradient ztrátové funkce získáme její derivací. Jelikož pracujeme s více proměnnými, výsledkem bude vektor parciálních derivací podle každého váhového koeficientu :

Úpravou tohoto výrazu pro pohyb v opačném směru oproti gradientu získáme následující rovnici:

Z uvedené rovnice zároveň vyplývá, že u gradientního způsobu optimalizace, který ADALINE využívá, se všechny váhové koeficienty aktualizují najednou:

**W**

Diagram

Description automatically generatedChart, surface chart

Description automatically generatedRychlost učení se v tomto případě určuje experimentálně, protože nesmí být příliš nízká (mohlo by dojít k ke zpomalení nebo zaseknutí algoritmu v lokálním minimu – viz obr. 2.3 a 2.4) ani vysoká (mohlo by dojít k přestřelení– viz obr. 2.5).

Obr. 2.5 Porovnání gradientního sestupu při optimální a příliš veliké rychlosti učení (přestřelení)

Obr. 2.4 Krajina obsahující globální a lokální minimum

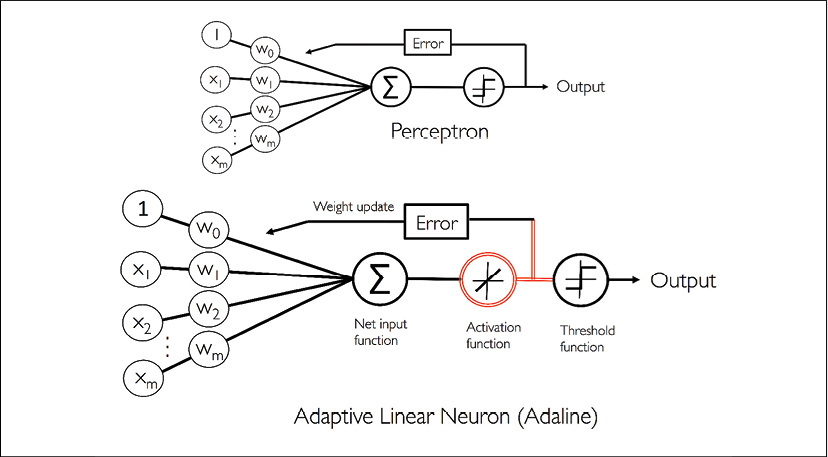
Obr. 2.3 Znázornění záseku gradientního sestupu v lokálním minimu ztrátové funkce

**2.6 Stochastický gradientní sestup**

*Stochastický gradientní sestup* (*stochastic gradient descent*) je populární (dost možná i častěji používanější) alternativou ke klasickému gradientnímu sestupu. Stochastický gradientní sestup redukuje problém s konvergenční rychlostí klasického gradientního sestupu (což je jeden z hlavních problémů tohoto algoritmu) tak, že namísto aktualizace na základě součtu všech chyb v celém trénovacím souboru aktualizuje váhové koeficienty inkrementálně zvlášť pro každý trénovací příklad souboru podle následující rovnice:

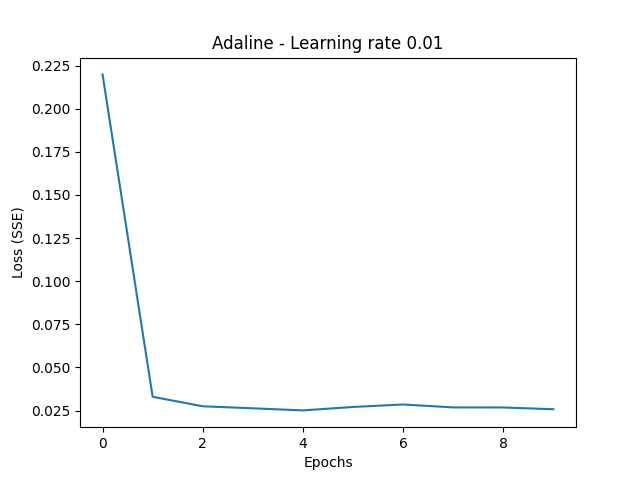
I když se může zdát, že SGD je pouze aproximací gradientního sestupu, avšak díky častějším aktualizacím váhových koeficientů nachází obvykle minimum funkce mnohem rychleji. Pořadí trénovacích příkladů v každé epoše musí být kvůli riziku přetrénování opět náhodné.

**2.7 Souhrn**

Adaptivní lineární neuron je jednovrstvá neuronová síť, která vychází z Rosenblattova perceptronu. Na lineární vstup aplikuje lineární aktivační funkci a pomocí definované ztrátové funkce (v tomto případě SSE) počítá ztrátové skóre, které udává míru chybovosti sítě při použití aktuálních váhových koeficientů . Pro aktualizaci vah aplikuje efektivní algoritmus gradientního sestupu. Jelikož pracujeme s více než jednou proměnnou, výsledkem této operace je vektor parciálních derivací podle každé váhy . Populární alternativou ke gradientnímu sestupu je stochastický gradientní sestup. Schéma ADALINE v porovnání s perceptronem je vidět na obrázku 2.6.

Obr. 2.6 Porovnání schémat perceptronu a ADALINE

**2.8 Implementace**

Implementace ADALINE v Pythonu je podobná jako u perceptronu. Kompletní kód s komentáři je k dispozici zde: <https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/ADALINE/ADALINE.py> (3). Graf na obrázku 2.7 ukazuje, jakým způsobem se během trénování snižovalo průměrné ztrátové skóre funkce SSE během deseti epoch při rychlosti učení 0,01:

Obr. 2.7 Graf zobrazující průměrné hodnoty ztrátové funkce SSE během 10 epoch

Obr. 2.7 Graf zobrazující průměrné hodnoty ztrátové funkce SSE během 10 epoch

Opět jsem pracoval se dvěma příznaky z datového souboru Iris a rozdělil příklady na trénovací a testovací množinu. Stejně jako u perceptronu jsem i zde bral v úvahu pouze dva druhy, jelikož se stále jedná o binární klasifikaci. Natrénovaný model opět správně klasifikoval všech 10 příkladů z testovací množiny dat. Jak je vidět z grafu na obrázku 2.7, stochastický gradientní sestup fungoval v tomto případě skvěle, jelikož ztrátová funkce byla minimalizována již během první epochy (samozřejmě i díky tomu, že se jedná o síť s velmi málo parametry). Algoritmus byl otestován tímto scriptem: <https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/Perceptron/perceptron_iris_binary.py> (4).

**3. Vícevrstvý perceptron**

**3.1 Inspirace a historický kontext**

*Vícevrstvý perceptron* (MLP – *multilayer perceptron*) je typ hluboké neuronové sítě, která se skládá z neuronů uspořádaných ve více vrstvách. V roce 1986 D.E.Rumelhart, G.E. Hinton a R.J. Wiliams ve své práci zpopularizovali algoritmus *zpětného šíření* (*backpropagation*), který je pro hluboké neuronové sítě klíčový, a obnovili tak ztracený zájem nejen většiny vědecké komunity, ale později i mnoha amatérů.

**3.2 Architektura MLP**

Diagram

Description automatically generatedMLP je typově *hluboká neuronová síť* (DNN – *Deep Neural Network*), jelikož se skládá z více vrstev. Zároveň se jedná o *hustě propojenou* (*dense*) síť, takže každý neuron jedné vrstvy bude propojen se všemi neurony následné vrstvy (viz obr. 3.1). Takové hluboké sítě, které nejsou cyklické, označujeme jako *dopředné* (*feedforward*).

Obr. 3.1 Příkladné schéma architektury vícevrstvého perceptronu (MLP)

**3.2.1 Vrstvy**

Vrstvy neuronových sítí slouží k předávání a transformaci vstupních dat pomocí aktivačních funkcí. Vrstvou se rozumí soubor neuronů se stejnými aktivačními funkcemi, které jsou zároveň propojeny s neurony dalších vrstev (v případě husté sítě plně). MLP obsahuje (ostatně jako všechny hluboké sítě) *vstupní vrstvu* (*input layer –* vkládáme do ní vlastní zadání příkladu), *výstupní vrstvu* (*output layer –* je v ní výsledek výpočtu sítě) a jednu nebo více *skrytých vrstev* (*hidden layers*). Vstupní vrstva slouží k přebrání vstupních dat, tedy příznaků, zatímco výstupní vrstva naopak transformuje vstupní data na požadované hodnoty (například můžeme chtít jako výstup pravděpodobnosti cílových tříd). Skryté vrstvy jsou přidávány kvůli nelinearitě, což je důležité při řešení složitějších problémů.

**3.2.2 Neurony a synapse**

Neurony v jednotlivých vrstvách MLP fungují naprosto stejně jako perceptron (odsud název vícevrstvý perceptron). Vstupní data jsou sítí předávána jako příznaky daného příkladu, takže počet neuronů ve vstupní vrstvě odpovídá počtu příznaků. Počet neuronů ve skrytých vrstvách je volitelný, počet neuronů ve výstupní vrstvě je obvykle určen počtem tříd (pokud pracujeme s více než dvěma třídami, jedná se o multinomiální klasifikaci a predikovaná třída je určena neuronem s nejvyšší hodnotou). Synapse, kterými jsou jednotlivé neurony propojeny, představují váhy.

**3.3 Dopředné šíření**

Proces trénování sítě ve smyslu předávání vstupních dat je obvykle označován jako *dopředné šíření* (*forward propagation*). Jedná se postupné transformování dat pomocí jednotlivých vrstev.

**3.3.1 Aktivace skryté vrstvy**

Hodnoty neuronů **A(in)** ve vstupní vrstvě jsou identické s hodnotami příznaků **X(i)**:

Jedinou výjimkou je první neuron , který obvykle představuje bias. Počet neuronů ve vstupní vrstvě tak odpovídá počtu příznaků **.** Prvním krokem při dopředném šíření je výpočet hodnot neuronů ve skryté vrstvě. Nejdříve musíme vypočítat lineární vstup daného neuronu (včetně biasu ), a poté na něj aplikovat aktivační funkci.

V dalším kroku aplikujeme na lineární vstup aktivační funkci . Hodnotu libovolného neuronu skryté vrstvy můžeme tedy zapsat jako:

**3.3.2 Aktivace výstupní vrstvy**

Výstupní vrstva na rozdíl od vstupní a skryté neobsahuje bias, ale pouze výstupní neurony **,** jinak je postup naprosto stejný. Hodnota libovolného výstupního neuronu je tedy opět:

Je důležité zvolit správnou aktivační funkci . Její výběr je určen tvarem výstupu, který od sítě požadujeme. Často používané aktivační funkce v případě MLP jsou například softmax nebo sigmoida.

**3.4 Ztrátová funkce – křížová entropie**

Při MLP můžeme použít pro výpočet ztrátového skóre použít například ztrátovou funkci (v teorii informace je tato funkce označována je křížová entropie **–** „cross entropy“**):**

Výraz představuje logistickou aktivaci *i*-tého příkladu. Tuto funkci ještě musíme upravit tak, aby brala v úvahu všechny výstupní aktivační jednotky **t** naší sítě:

3.4.1 Regularizace ztrátové funkce

Experimentálně bylo zjištěno, že přetrénovaná síť obsahuje část vazebných koeficientů ***w***, které jsou v absolutní hodnotě výrazně větší než ty ostatní. Intuitivně se dá říci, že právě to je ten způsob, jak si síť některé motivy pamatuje přehnaně více na úkor zevšeobecnění příkladů (přetrénování). Způsob, jak toto potlačit, je přičíst ke ztrátové funkci člen úměrně závisející na velikosti všech synapsí (vah). Tento postup se nazývá regularizace. Její nejčastější forma je tzv. *L2 regularizace*, což znamená přičtení součtu kvadrátů všech vazeb ***w*** do ztrátové funkce:

kde je vhodná regularizační konstanta (obvykle o velikosti 0 – 0,1) a součet probíhá přes všechny vazby v síti (kromě těch navázaných na biasy). Ztrátová funkce s L2 regularizací bude tedy vypadat takto:

**+**

Použití L2 regularizace není zásadní pro všechny typy sítí, avšak výkon modelu se často může výrazně zlepšit (sníží se šance na přetrénování).

3.5 Zpětné šíření chyby

*Algoritmus zpětného šíření chyby* (*backpropagation algorithm*) je velmi mocný nástroj pro optimalizaci váhových koeficientů většiny typů neuronových sítí, založený na gradientním sestupu. Zjednodušeně se jedná o efektivní způsob propagace chyby v opačném směru a následné optimalizace vah na základě „chybovosti“ každé vrstvy. **C**hyba poslední skryté vrstvy ***h*** je dána výrazem:

***δ(h) = δ(out)(W(out))T ʘ***

kde ***δ(h)*** je vektor chyb celé skryté vrstvy a symbol ***ʘ*** označuje tzv. Hadamardův součin (*Hadamard product*), neboli součin vektorů po jednotlivých složkách. ***δ(out)*** je vektor výstupních chyb sítě (prostý rozdíl výstupu sítě ***a(out)*** a správného cílového výsledku ***y*** daného příkladu):

***δ(out) = a(out) – y***

Získané chyby pro každý neuron v síti potom tvoří základ pro dopočet skutečných gradientů, tedy parciálních derivací ztrátové funkce podle vah a biasů těchto vnitřních neuronů:

***J(W) =***

***J(W) =***

kde ajsou aktivace j-tého neuronu skryté, respektive vstupní vrstvy. Schéma algoritmu zpětného šíření je vidět na obrázku 3.2.



Obr. 3.2 Popsané schéma algoritmu zpětného šíření chyby

3.6 Souhrn

MLP (vícevrtsvý perceptron) je umělá neuronová síť schopná provádět multinomiální klasifikaci. Skládá se ze vstupní, výstupní a skrytých vrstev, které jsou tvořeny jednotlivými neurony (a biasy), které jsou vzájemně propojeny synapsemi (váhami). Při trénování MLP jsou nejdříve náhodně určeny hodnoty všech vah. Poté je provedeno dopředné šíření (jedná se dopřednou síť), při kterém se vypočítají hodnot všech neuronů a aplikují se na ně aktivační funkce. Následně je určena hodnota ztrátové funkce a provádí se zpětná propagace chyby, při které jsou aktualizovány váhové koeficienty na základě gradientu ztrátové funkce.

3.7 Implementace

3.7.1 MNIST

K otestování této sítě byla použita databáze MNIST obsahující 60 000 trénovacích a 10 000 testovacích černobílých obrázků rukou psaných číslic s rozměry 28x28 pixelů (<http://yann.lecun.com/exdb/mnist/>). O zpracování stažených souborů tak, aby se poté daly jednoduše načítat z jediného souboru se stará tento script: <https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/MLP/dataset/load_mnist.py> (5).

3.7.2 MLP v prostém Pythonu

Implementace MLP v prostém Pythonu je složitá, výpočetně náročná a neefektivní. Komentovaný příklad takové implementace je k dispozici zde: <https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/MLP/MLP.py> (6).

Graf na obrázku 3.3 ukazuje, jakým způsobem se během trénování snižovalo průměrné ztrátové skóre funkce křížové entropie během 300 epoch při rychlosti učení 0,0005 s regularizací L2 o hodnotě 0,01, velikostí dávky 100 příkladů (s povoleným mícháním trénovacích příkladů) a 150 neuronech ve skryté vrstvě. Přesnost natrénovaného MLP byla 98,06 %. K otestování modelu byl použit tento script: <https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/MLP/mlp_mnist.py> (7).

Chart

Description automatically generated

Obr. 3.3 Graf zobrazující průměrné hodnoty křížové entropie během 300 epoch

3.7.3 MLP v TensforFlow Keras

Pro demonstraci síly a efektivitiy frameworku Keras v rámci moderní knihovny TensorFlow pro budování hlubokých neuronových sítí lze nahlédnout do tohoto kódu: <https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/MLP/Keras_MLP.py> (8). Toto je kompletní kód potřebný k implementaci MLP se stejnými parametry a konfigurací (kromě aktivační funkce výstupní vrstvy – zde jsem použil softmax namísto sigmoidy) jako při implementaci v prostém Pythonu. Díky knihovně TensorFlow a frameworku Keras je nepochybně kratší (41 řádek kódu oproti 210), jednodušší na čtení, výpočetně efektivnější a snadno upravovatelný. Přesnost tohoto MLP byla 95,5 % a ztrátové skóre na konci trénování 0,15.

**Zdroje obrazových příloh**

Obr. 1.1

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch02/images/02_04.png>

Obr. 2.1

<https://www.researchgate.net/figure/Non-convex-optimization-We-utilize-stochastic-gradient-descent-to-find-a-local-optimum_fig1_325142728>

Obr. 2.2

<https://morioh.com/p/b0fefc78f330>

Obr. 2.3

<https://www.researchgate.net/figure/Gradient-Descent-Stuck-at-Local-Minima-18_fig4_338621083>

Obr. 2.4

<https://medium.com/analytics-vidhya/journey-of-gradient-descent-from-local-to-global-c851eba3d367>

Obr. 2.5

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch02/images/02_12.png>

Obr. 2.6

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch02/images/02_09.png>

Obr. 3.1

<https://www.researchgate.net/figure/The-Multi-Layer-Perceptron-ANN-scheme-1_fig3_262252654>

Obr. 3.2

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch12/images/12_12.png>

**Použitý kód**

1. *Perceptron.py*

[*https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/Perceptron/Perceptron.py*](https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/Perceptron/Perceptron.py)

1. *perceptron\_Iris\_binary.py*

[*https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/Perceptron/perceptron\_iris\_binary.py*](https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/Perceptron/perceptron_iris_binary.py)

1. *ADALINE.py*

[*https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/ADALINE/ADALINE.py*](https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/ADALINE/ADALINE.py)

1. adaline\_iris\_binary.py

[*https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/ADALINE/adaline\_iris\_binary.py*](https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/ADALINE/adaline_iris_binary.py)

1. load\_mnist.py

[*https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/MLP/dataset/load\_mnist.py*](https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/MLP/dataset/load_mnist.py)

1. MLP.py

<https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/MLP/MLP.py>

1. mlp\_mnist.py

[*https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/MLP/mlp\_mnist.py*](https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/MLP/mlp_mnist.py)

1. Keras\_MLP.py

<https://github.com/Hengrs99/Neural-Networks/blob/main/Networks/MLP/Keras_MLP.py>