ODBORNÁ PRÁCE

2021/2022

**MATEMATIKA A IMPLEMENTACE NEURONOVÝCH SÍTÍ**

Petr Jeřábek

Logo

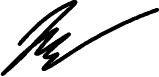
Description automatically generated

4. A

**Prohlášení**

Já, Petr Jeřábek, prohlašuji, že jsem tuto práci vypracoval samostatně a řádně jsem citoval použité zdroje.

V Praze dne 16. 4. 2022



**Anotace**

Tato odborná práce se zabývá matematikou umělých neuronových sítí. Zaměřuje se na fungování různých typů umělých neuronů a jednoduchých sítí o více vrstvách. Algoritmy jsou následně implementovány v programovacím jazyku Python.

**Annotation**

This professional work deals with the math of artificial neural networks. It focuses on the functionality of various types of artificial neurons and a simple network with multiple layers. Algorithms are then implemented in the Python programming language.

**Klíčová slova**

hluboké učení, Python, neuronové sítě, umělý neuron, perceptron, ADALINE

**Key words**

deep learning, Python, neural networks, artificial neuron, perceptron, ADALINE

**Poděkování**

Mnohokrát děkuji mojí profesorce matematiky, Mgr. Kateřině Brochot, za odborné vedení a kontrolu mojí práce. Obrovské díky také patří mému tátovi, který ve mně od dětství probouzel zájem o přírodní vědy a technologie, sehrál zásadní roli v mém studiu neuronových sítí a vždy si rád našel čas na detailní komentáře k této práci. Dále chci vyjádřit své díky celé online komunitě věnující se neuronovým sítím a programování obecně. Jedná se zejména o Dr. Sebastiana Raschku, jehož práce je solidním základem nejen většiny použitého kódu, ale také mých znalostí v tomto oboru. V neposlední řadě chci poděkovat všem open source službám, které jsem při práci využíval (Python, TensorFlow a Keras, GitHub, Visual Studio Code, Jupyter).

Obsah

1. **Úvod…………………………………………………………………………………………………………………….6**
2. **Rosenblattův perceptron………………………………………………………………………………………9**
   1. Inspirace a historický kontext……………………………………………………………………………..9
   2. Formální definice………………………………………………………………………………………………10
   3. Optimalizace váhových koeficientů…………………………………………………………………..12
   4. Souhrn………………………………………………………………………………………………………………14
   5. Implementace…………………………………………………………………………………………………..15
3. **ADALINE..……………………………………………………………………………………………………………21**
   1. Inspirace a historický kontext……………………………………………………………………………21
   2. Lineární aktivační funkce…………………………………………………………………………………..21
   3. Ztrátová funkce…………………………………………………………………………………………………22
   4. Gradientní sestup……………………………………………………………………………………………..23
   5. Optimalizace váhových koeficientů…………………………………………………………………..24
   6. Stochastický gradientní sestup………………………………………………………………………….26
   7. Souhrn………………………………………………………………………………………………………………27
   8. Implementace…………………………………………………………………………………………………..28
4. **Vícevrstvý perceptron…………………………………………………………………………………………32**
   1. Inspirace a historický kontext……………………………………………………………………………32
   2. Architektura MLP………………………………………………………………………………………………32
      1. Vrstvy…………………………………………………………………………………………………...33
      2. Neurony a synapse………………………………………………………………………………..33
   3. Dopředné šíření………………………………………………………………………………………………..34
      1. Aktivace skryté vrstvy……………………………………………………………………………34
      2. Aktivace výstupní vrstvy………………………………………………………………………..36
   4. Ztrátová funkce – křížová entropie……………………………………………………………………38
      1. Regularizace ztrátové funkce…………………………………………………………………39
   5. Zpětné šíření chyby…………………………………………………………………………………………..40
   6. Souhrn……………………………………………………………………………………………………………...42
   7. Implementace…………………………………………………………………………………………………..43
      1. MNIST……………………………………………………………………………………………………43
      2. MLP v prostém Pythonu……………………………………………………………………..…44
      3. MLP v TensorFlow Keras………………………………………………………………………..53
5. **Závěr…………………………………………………………………………………………………………………..56**

1. Úvod

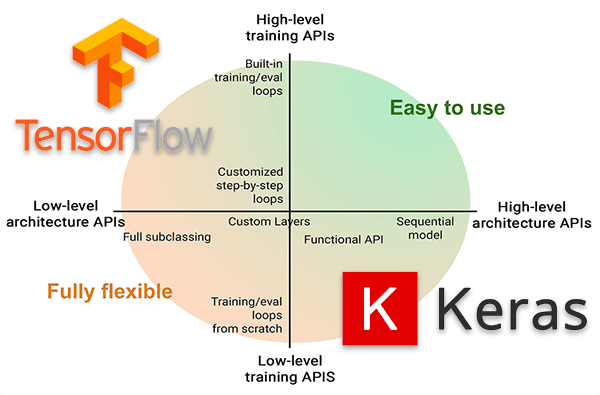
Během posledních zhruba 10 let došlo k obrovskému skoku v interdisciplinárním oboru *umělé inteligence* (AI – *Artificial Inteligence*). Tento nevídaný posun se konkrétně týkal tzv. *umělých neuronových sítí* (*artificial neural networks*). Samotné *umělé neurony* (*artificial neurons*) a neuronové sítě přitom nebyly v matematice či počítačové vědě žádnou novinkou, začátek jejich výzkumu se dá dokonce datovat už ke konci II. světové války (McCulloch-Pitts model neuronu, 1943). Nicméně až teprve vývoj hardwarových a softwarových prostředků s dostatečnou kapacitou, rychlostí a sofistikovaností v poslední dekádě umožnil skutečnou realizaci těchto biologicky inspirovaných výpočetních zařízení. Došlo dokonce i k jisté změně základních termínů, dnešní neuronové sítě jsou většinou nahlíženy jako tzv. hluboké sítě (neuronové sítě s více než jednou skrytou vrstvou, obvykle v počtu vyšších desítek) a proces jejich trénování je nazýván *hluboké učení* (DL – *Deep Learning*), což je samostatný obor v rámci *strojového učení* (ML – *Machine Learning*) – rozdělení a překrývání hlavních oborů AI ukazuje obrázek 1.1.

Obsah obrázku text, elektronika, snímek obrazovky

Popis byl vytvořen automaticky

Obr. 1.1 Názorné rozdělení podoborů AI

Předkládaná práce se pokouší o výklad a prezentaci základních konceptů, metodologií a ideových východisek dnešních sítí tak, jak historicky v minulém století vznikaly, nicméně již z hlediska a s pomocí současných technologických platforem a programovacích prostředků. Projdeme tedy první skutečně sebeučící se algoritmus *perceptronu*, jeho matematicky dokonalejšího následníka *ADALINE*, abychom se dopracovali k první skutečné síti ve smyslu více neuronů – tzv. vícevrstvý perceptron. Tyto algoritmy byly historicky realizovány buď přímo hardwarově nebo s pomocí strojově orientovaných programovacích jazyků. My je ovšem budeme navrhovat, po jejich teoretickém zdůvodnění, v moderním jazyce *Python*, který je dnes v oblasti umělé inteligence a zvlášť v oblasti strojového učení a neuronových sítí velmi oblíben a preferován. Tyto návrhy provedeme dvěma způsoby, jednak čistě pythonickými základními jazykovými prostředky (v technickém žargonu tzv. *from scratch*), a jednak pomocí speciální technologické programovací platformy *Keras* (která je sama nadstavbou platformy *Tensorflow* – viz obr. 1.2). Tímto dvojím provedením budeme

demonstrovat současný vývoj – v prostém Pythonu lze názorně vidět stavbu a vnitřní detaily algoritmu, nicméně v moderní nadstavbové API platformě zredukujeme mnohonásobně množství potřebného kódu díky zabudovaným funkcionalitám.

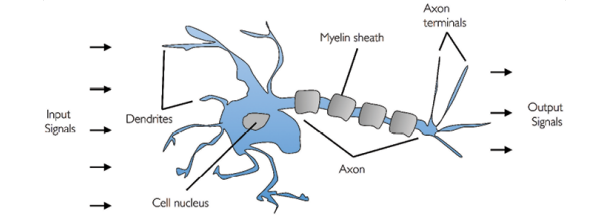
Obr. 1.2 Přehled odlišností mezi Tensorflow a Keras

Tento podobor umělé inteligence – strojové učení pomocí umělých neuronových a hlubokých sítí, hluboké učení, je relativně pořád mladý, v podstatě v začátcích a bouřlivě se rozvíjející. Rovněž jeho terminologie není v češtině plně ustálená, případně je používáno více obdobných výrazů pro jedno a totéž. Autor se tedy omlouvá, použije-li někde jiný výraz, než je v obdobné souvislosti použit v nějakém jiném textu. Zároveň autor vítá jakékoliv připomínky a zpětnou vazbu k této práci, jejíž téma je sice v daném oboru základní, ale odráží jeho vlastní cestu ve studiu těchto fascinujících záležitostí. Celá práce včetně všech příloh je k dispozici na autorově GitHubu na adrese: <https://github.com/Hengrs99/Neural-Networks>.

2. Rosenblattův perceptron

2.1 Inspirace a historický kontext

Umělé neuronové sítě jsou inspirovány skutečným uspořádáním nervové soustavy. Abychom správně pochopili jejich architekturu a fungování, zamysleme se nejdříve nad fungováním lidského mozku. Ten může obsahovat až 100 miliard mozkových buněk (neuronů – viz obr. 2.2), které považujeme za jeho základní stavební jednotky. Neurony jsou mezi sebou propojeny synapsemi a předávají si chemické a elektrické signály přijímané dendrity. Pokud intenzita elektrického potenciálu přesáhne určitý práh, neuron vypálí signál dále po axonu a přenáší tak informaci. V případě opakujících se vzorců chování jsou zodpovědné synapse posíleny.3 Tento koncept v roce 1943 poprvé představili Warren McCulloch a Walter Pitts s cílem navrhnout fungující umělou inteligenci. Mozkovou buňku popsali jako prostý rozhodovací mechanismus s binárním výstupem (jejich úspěšný model nese název *MCP – McCulloch-Pitts neuron* – viz obr. 2.1).4

****

Obr. 2. Model MCP neuronu

Obsah obrázku strom, posazený, exteriér, větev

Popis byl vytvořen automatickyJen o pár let později publikoval Frank Rosenblatt na základě MCP neuronu první koncept nejjednoduššího modelu jednovrstvé neuronové sítě, takzvaný perceptron. Tento algoritmus se řídí velmi jednoduchými pravidly inspirovanými skutečnými mozkovými buňkami. Jeho cílem je optimalizovat váhové koeficienty, kterými jsou násobeny příznaky vstupních dat pro výpočet „intenzity signálu“. Jedná se o první umělý neuron, který poskytuje základ pro všechny složitější architektury umělých neuronových sítí.

Podobnosti mezi umělými a biologickými neurony budou stále viditelnější při detailnějším vysvětlení v dalších kapitolách.

Obr. 2. Mikroskopický snímek neuronu

**2.2** **Formální definice**

Perceptronový algoritmus funguje dobře zejména při řešení problému binární klasifikace. Představme si, že chceme naše data na základě jejich příznaků rozdělit do dvou tříd, a . Perceptron pro každý příklad vypočítá lineární kombinaci vektoru příznaků a vektoru váhových koeficientů .

Tato práce důsledně využívá maticový zápis formulí. Tedy zvýrazněný font znamená, že pracujeme s vektorovou, případně maticovou (zejména matice vah ) veličinou.

Pokud výsledná hodnota přesáhne stanovený práh klasifikujeme příklad jako třídu (neuron pálí), pokud bude nižší, připadne příklad třídě (neuron není aktivován). Výsledné hodnotě lineární kombinace říkáme lineární vstup (z anglického *net input*). O tom, zda lineární vstup přesáhne stanovený práh , rozhoduje *skoková funkce* **,** na kterou zatím nelze použít gradientní přístup optimalizace (kvůli její nespojitosti – viz kapitola 3). Funkcím, které mapují lineární vstup neuronu na jiné hodnoty, se v kontextu umělých neuronových sítí říká *aktivační funkce*.5 Budu používat variantu skokové funkce, která převádí kladné proměnné na hodnotu 1 a záporné na -1 (obvykle jsou záporné proměnné převáděny na hodnotu 0, pro můj příklad se však více hodí tato varianta – viz obr. 2.3):

neboli:

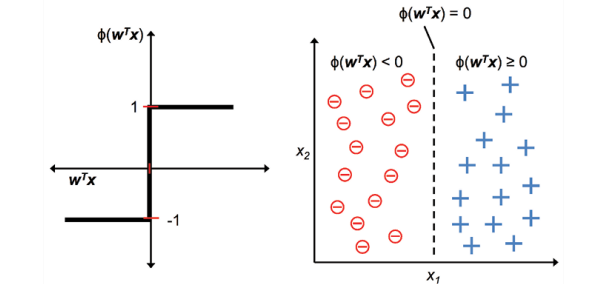
Následně můžeme z rovnice práh úplně odstranit tak, že prvnímu váhovému koeficientu přiřadíme hodnotu a korespondujícímu prvnímu příznaku hodnotu 1:

Negativní práh nazýváme bias. Ten můžeme považovat za základní hodnotu neuronu, kterou budeme vždy přičítat k lineární kombinaci. Zápis rovnice pro výpočet lineárního vstupu můžeme nyní napsat v kompaktní vektorové formě1:

**; ;**

Proces klasifikace vstupních dat podle perceptronu můžeme tedy jednoduše shrnout do těchto kroků:

1. Výpočet lineární kombinace vektoru váhového vektoru a vektoru příznaků příkladu a přičtení jednotky biasu
2. Vyhodnocení klasifikace lineárního vstupu podle skokové aktivační funkce (nebo **)**



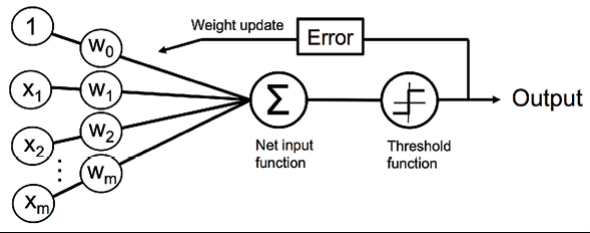
Obr. 2. Skoková aktivační funkce vytvářející predikce

**2.3 Optimalizace váhových koeficientů**

Cílem nejen perceptronu, ale i všech neuronových sítí je optimalizovat váhové koeficienty pro co nejlepší výsledky. Vektor těchto koeficientů je na začátku generován zcela náhodně v rámci určeného rozmezí (například 0 až 1). Při prvních iteracích během učení tedy není příliš veliká pravděpodobnost, že perceptron klasifikuje data správně. Algoritmus zkrátka nejdříve vůbec neví, co dělá, a tak prostě tipuje. Aby dokázal váhové koeficienty optimalizovat, je mu pro porovnání poskytnut vektor cílových tříd výsledků .

Tento přístup nazýváme *učení s učitelem* (z anglického *supervised learning*).2 Formálně můžeme aktualizaci každého váhového koeficientu zapsat jako:

Pro výpočet si nejdříve musíme stanovit *rychlost učení (learning rate)* **η.** Tu obvykle určíme jako konstantu s hodnotou mezi 0 až 1. Aktualizaci každého váhového koeficientu (neboli proces, který nazýváme jako *trénování* nebo *učení*) poté definujeme jako násobek rychlosti učení , závislého příznaku a rozdílu mezi cílovou () a predikovanou ( třídou:

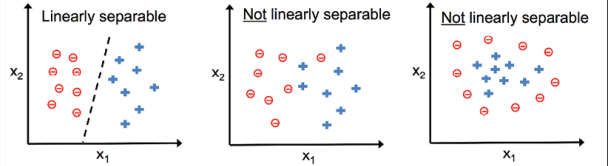
Během klasifikace každého příkladu z trénovacího vzorku (ten obvykle obsahuje zhruba 80 % datového souboru) takto postupně aktualizujeme každý váhový koeficient. Perceptron se obvykle trénuje na celkovém datovém souboru několikrát. Pro počet trénovacích iterací se používá název *epochy.2* Pořadí příkladů ve vzorku je typicky určeno náhodně pro každou epochu, protože algoritmus by si jinak mohl „zapamatovat“ který příklad koresponduje s jakým výstupem a nebyl by schopen pracovat s novými daty – tento problém nazýváme *přetrénování (overfitting)*.1 Schéma percetronu je dobře vidět na obrázku 2.4.

Obr. 2. Schéma perceptronového algoritmu

**2.4 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. To je proces, kterému říkáme *učení* (odsud pojem *strojové/hluboké učení*) nebo *trénování*. Ideálně se snažíme o to, aby model dosahoval dobrých výsledků nejen na trénovací množině, ale aby dokázal pracovat efektivně také s novými daty (k validaci slouží *testovací datový soubor* – viz implementace). Celý proces učení můžeme shrnout těmito kroky:

1. Náhodná inicializace vektoru váhových koeficientů
2. Určení prahu à převod prahu na bias
3. Iterace přes každý příklad trénovací datové množiny a počet epoch
   1. Výpočet lineárního vstupu a výstupu skokové funkce
   2. Aktualizace váhových koeficientů na základě učebního pravidla

Po dovršení počtu epoch nebo dosažení určené přesnosti (pokud je skutečně dosažitelná) je perceptronový model naučený řešit problém binární klasifikace a může být aplikován na nové příklady stejného typu mimo trénovací množinu. Je však důležité, aby byla poskytnutá data lineárně separovatelná (úplné oddělení dat v různých výstupních třídách lineárním útvarem, například přímkou ve 2D případě – viz obr. 2.5), jinak by je perceptron nebyl schopen klasifikovat.

Obr. 2. Příklady lineárně rozdělitelných a nerozdělitelných datových bodů

**2.5 Implementace**

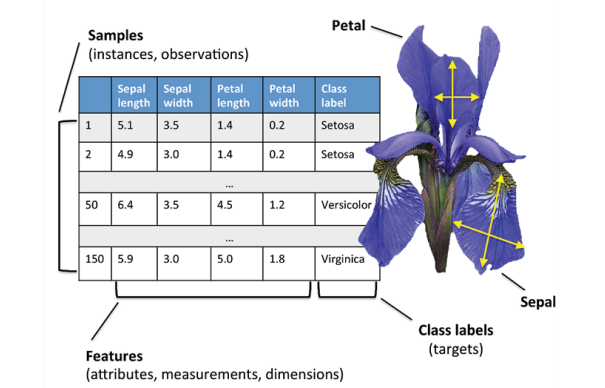
Implementace perceptronu v Pythonu je velmi jednoduchá, jelikož se jedná pouze o jeden neuron. Kompletní kód je k dispozici v příloze, zde představím pouze klíčové metody. Ty jsou součástí třídy *Perceptron()*:



Kód 1 - Perceptron.py

O trénování klasifikátoru se stará metoda *fit(X, y),* která jako parametry vyžaduje příznaky příkladů v trénovacím datovém souboru **X** a jejich cílové třídy **Y** pro porovnání. Na základě výše definovaného učebního pravidla optimalizuje váhové koeficienty.

Na otestování perceptronu využiji datový soubor Iris (viz obr. 2.6), který obsahuje míry okvětních lístků tří druhů kosatce – setosa, versicolor a virginica (každý druh je zastoupen 50 příklady – soubor dohromady obsahuje 150 příkladů).



Obr. 2. Popsaná reprezentace datového souboru Iris

Celý soubor můžeme zapsat jako matici , kde každý řádek *i* reprezentuje jeden příklad a každý sloupec *j* představuje daný příznak :

Cílové třídy příkladů zase zapíšeme jako vektor :

Jelikož perceptron je binární klasifikátor, budu pracovat pouze se druhy setosa a versicolor, takže trénovací datový soubor bude obsahovat pouze 100 příkladů (stejně tak vektor cílových tříd bude mít velikost )1:

Následující kód načte datový soubor Iris a extrahuje z něj prvních 100 příkladů. Vzorkům označeným jako versicolo*r* bude ve vektoru cílových tříd přiřazena hodnota 1, těm označeným jako setosa zase -1:

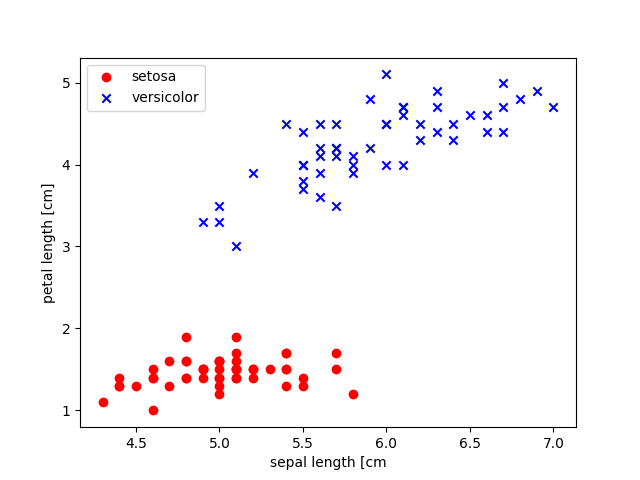
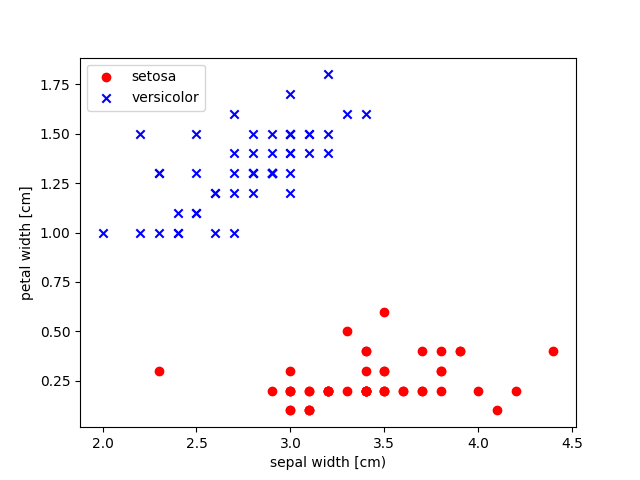


Kód 2 – perceptron\_iris\_binary.py

Datový soubor bude dále rozdělen na 80 trénovacích vzorků (40 od každého druhu) a 20 pro testování natrénovaného modelu (10 od každého druhu):



Kód 2 – perceptron\_iris\_binary.py

Z následujících grafů (viz obr. 2.7 a 2.8) můžeme jednoznačně určit, že data z datového souboru Iris jsou dobře lineárně rozdělitelná (podle délek a šířek lístků korun a kalichů obou druhů), takže perceptron by neměl mít s klasifikací žádné problémy:

Obr. 2.8 Rozdělení jednotlivých druhů podle délky okvětních lístků

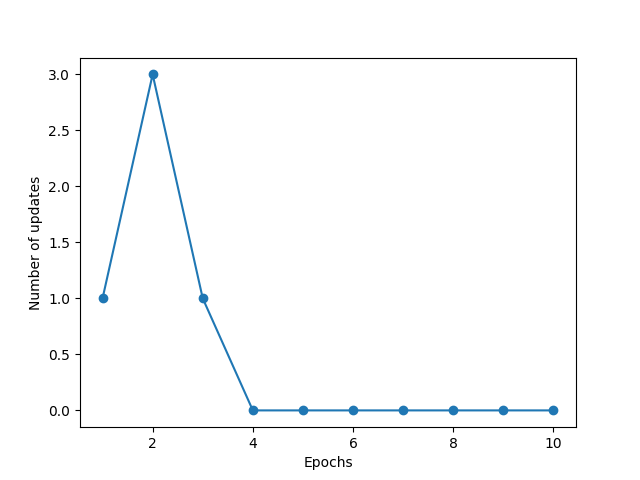
Obr. 2.7 Rozdělení jednotlivých druhů podle šířky okvětních lístků

Nakonec importujeme výše uvedenou třidu *Perceptron()* a předáme trénovací množinu dat metodě *fit(X, y).* Kvůli jednoduššímu 2D zobrazení grafů pracuji pouze se dvěma příznaky, a to sice s délkami okvětních lístků koruny a kalichu.Rychlost učení jsem zvolil o velikosti 0,1 a počet epoch nastavil na 10:



Kód 2 – perceptron\_iris\_binary.py

Jak ukazuje tento graf (viz obr. 2.9), perceptronový algoritmus dosáhl cíle a přestal optimalizovat váhové koeficienty ve čtvrté epoše:



Obr. 2. Graf znázorňující počet aktualizací vah v každé epoše

Natrénovaný model poté správně klasifikoval všech 10 příkladů z testovacího souboru.

Při trénování neuronových sítí obecně je již zcela běžnou praxí rozdělit datový soubor

na trénovací a testovací množinu2 (stejně jako jsem to v tomto příkladu udělal já). Získáme tak mnohem lepší představu o skutečné přesnosti natrénovaného modelu (v tomto případě 100 %), protože při špatně zvolených parametrech může jednoduše dojít k již zmíněnému přetrénovaní.

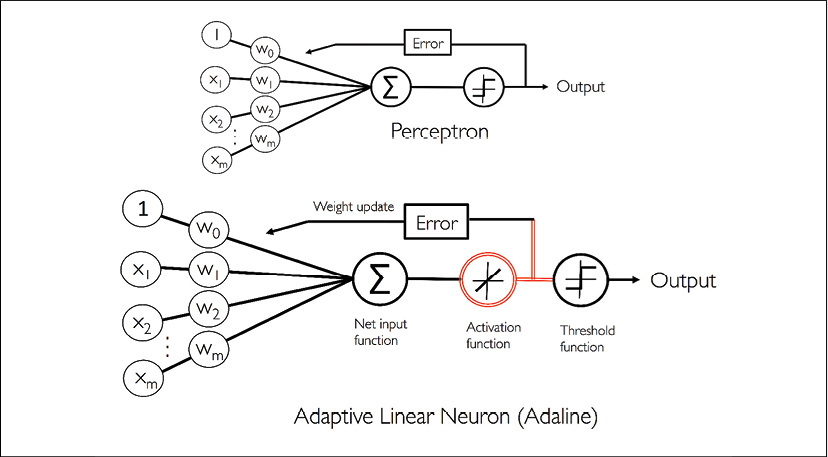
**3. ADALINE**

**3.1 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. ADALINE poprvé představil prof. Bernard Widrow a jeho student Ted Hoff ze Stanfordské univerzity. Jako podklad sloužil právě model McCulloch-Pitts neuronu, na základě kterého navrhl Rosenblatt perceptronový algoritmus.

**3.2 Lineární aktivační funkce**

Hlavním rozdílem mezi ADALINE a perceptronem (viz obr. 3.1) je způsob, jakým probíhá učení (optimalizace vah). 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.6 Kromě skokové funkce kvůli predikci využívá navíc *lineární aktivační funkci*, která je prostou identickou funkcí lineárního vstupu1:



Obr. 3. Porovnání schémat perceptronu a ADALINE

**3.3 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 :

Použití lineární aktivační funkce namísto skokové je výhodné hlavně z toho důvodu, že ztrátová funkce je nyní diferencovatelná (má v každém bodě derivaci2)a konvexní (má globální minimum)1, což je důležité pro použití matematických metod při hledání minim ztrátových funkcí (viz gradientní sestup níže) – rozdíl mezi konvexní a konkávní funkcí je vidět na obrázku 3.2.

Chart, line chart

Description automatically generated

Obr. 3.2 Příklady konvexní a konkávní funkce

**3.4 Gradientní sestup**

Algoritmus *gradientního sestupu* (*gradient descent*) GDje velmi efektivní a populární způsob optimalizace váhových koeficientů. Používá se nejen v oblasti hlubokého učení, ale také v běžných algoritmech strojového učení (například v lineární regresi). 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).7 Optimalizační algoritmuspoté upravuje váhové koeficienty v opačném směru gradientu ztrátové funkce (viz obr. 3.4), protože gradient určuje směr nejrychlejšího růstu ztrátové funkce – viz dále 3.5. Snaží se tak co nejvíce přiblížit jejímu globálnímu minimu (ztrátové skóre zde bude velmi nízké – hodnoty vah budou nejoptimálnější). Často používané vysvětlení principu gradientního sestupu je představa o klesání z kopce do jeho nejnižšího bodu (viz obr. 3.3).

Chart, surface chart

Description automatically generated

Obr. 3.4 Znázornění gradientního sestupu ve 2D

Obr. 3.3 Model znázorňující gradientní sestup ve 3D

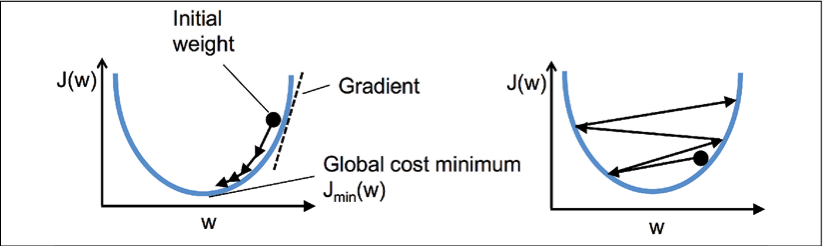
**3.5 Optimalizace váhových koeficientů**

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

Tímto jsme zjistili, jak vypadá výpočet gradientu naší ztrátové funkce:

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

Jak jsem již vysvětloval, pro efektivní optimalizaci vah se musíme pohybovat v opačném směru oproti gradientu v daném bodě, kde značí vektor příznaků daného příkladu:

Záporný gradient je zároveň násoben rychlostí učení . Ta v tomto případě určuje, o jak velký krok se váhové koeficienty posunou ve směru záporného gradientu . Pokud je rychlost příliš veliká, může se stát, že optimalizační algoritmus globální minimum ztrátové funkce doslova *přestřelí* (viz obr. 3.5) a nikdy ho nedosáhne.

Obr. 3.5 Porovnání gradientního sestupu při optimální a příliš veliké rychlosti učení (přestřelení)

Pokud je však rychlost učení příliš malá, algoritmus se může zaseknout v lokálním minimu ztrátové funkce (viz obr. 3.6 a 3.7), ve kterém nemusí být váhové koeficienty optimální.7 Zároveň se snižuje rychlost konvergence (algoritmus je pomalejší).

Chart, surface chart

Description automatically generatedDiagram

Description automatically generated

Obr. 3.7 Krajina obsahující globální a lokální minimum

Obr. 3.6 Znázornění záseku gradientního sestupu v lokálním minimu ztrátové funkce

**3.6 Stochastický gradientní sestup**

*Stochastický gradientní sestup* (*stochastic gradient descent*) je populární (dost možná i častěji používanější) alternativou ke klasickému gradientnímu sestupu. Jedním z hlavních problémů GD je jeho konvergenční rychlost (rychlost posunu směrem k minimu). Očividným řešením tohoto problému je určení větší rychlosti učení , avšak jak jsem již vysvětloval, kvůli přestřelování je při příliš velikých hodnotách téměř nemožné dosáhnout minima. Stochastický gradientní sestup redukuje problém s konvergenční rychlostí tak, že namísto aktualizace na základě součtu všech chyb v celém trénovacím souboru aktualizuje váhové koeficienty inkrementálně zvlášť pro každý trénovací příklad souboru (viz obr. 3.8):

Může se zdát, že SGD je pouze aproximací gradientního sestupu (protože v původním odvození metody GD pracujeme s celým trénovacím souborem), avšak díky častějším aktualizacím váhových koeficientů nachází obvykle minimum funkce mnohem rychleji. Důležité je, aby bylo v každé epoše určeno pořadí trénovacích příkladů náhodně, jinak může jednoduše dojít k přetrénování. 1

Diagram

Description automatically generated

Obr. 3.8 Stochastický gradientní sestup s inkrementální aktualizací

**3.7 Souhrn**

Adaptivní lineární neuron je jednovrstvá neuronová síť, která vychází z Rosenblattova perceptronu. Na lineární vstup aplikuje lineární aktivační funkci (ta je identická, hodnota lineárního vstupu se tedy nemění). Dále 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, který funguje tak, že pomocí derivace ztrátové funkce (ta musí být konvexní a diferencovatelná) zjistí její gradient, a následně učiní krok v opačném směrem podle rychlosti učení . Jelikož pracujeme s více než jednou proměnnou, výsledkem je vektor parciálních derivací podle každé váhy . Pro správné fungování gradientního sestupu musí být pořadí dat v každé epoše zamícháno a rychlost učení nesmí být příliš velká ani malá (parametr se většinou určuje experimentálně), aby nedocházelo k přestřelování a rychlost konvergence byla optimální. Populární alternativou ke gradientnímu sestupu je stochastický gradientní sestup, který pracuje s jednotlivými příklady namísto celého datového souboru, většinou tedy konverguje o dost rychleji.

**3.8 Implementace**

Implementace ADALINE v Pythonu je podobná jako u perceptronu. Celý algoritmus je opět zabalen ve třídě *ADALINE():*



Kód 3 – ADALINE.py

Parametry třídy *ADALINE()* jsou naprosto shodné s nastavením perceptronu (viz kapitola 2). Jediným rozdílem je atribut *cost\_,* který uchovává průměrnou hodnotu SSE v každé epoše. O trénování modelu se opět stará metoda *fit(X, y)*:



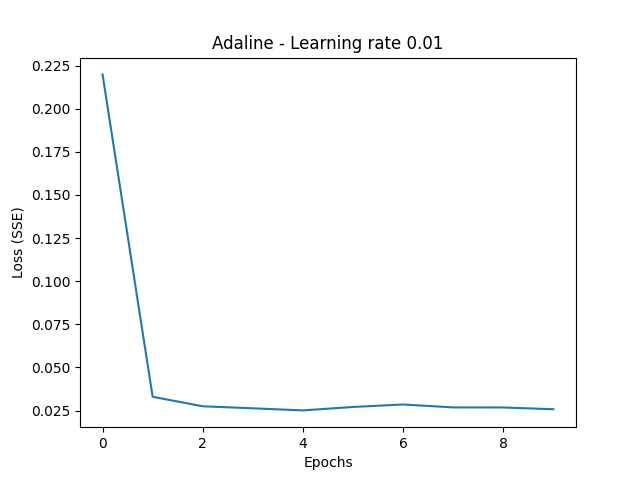
Kód 3 – ADALINE.py

Jak můžeme vidět, aktualizace vah není prováděna přímo metodou *fit(X, y)*, nýbrž pomocnou metodou *\_update\_weights(self, xi, gt)*. Kvůli zjednodušení jsem jako optimalizační algoritmus zvolil stochastický gradientní sestup, takže tato metoda vyžaduje pouze jeden vektor příznaků (jeden trénovací příklad) **xi** a hodnotu skutečné třídy **gt**:



Kód 3 – ADALINE.py

Následující graf ukazuje, jakým způsobem se během trénování snižovalo průměrné ztrátové skóre funkce SSE během deseti epoch při rychlosti učení 0,01:



Obr. 3.9 Graf zobrazující průměrné hodnoty ztrátové funkce SSE během 10 epoch

Opět jsem pracoval se dvěma příznaky z datového souboru Iris (viz kapitola 2) a rozdělil příklady na trénovací a testovací množinu. Jelikož ADALINE je stejně jako perceptron algoritmus binární klasifikace, bral jsem v úvahu pouze druhy setosa a versicolor:



Kód 4 – adaline\_iris\_binary.py

Natrénovaný model opět správně klasifikoval všech 10 příkladů z testovací množiny dat. Jak je vidět z grafu na obrázku 3.9, stochastický gradientní sestup fungoval v tomto případě skvěle, jelikož ztrátová funkce byla minimalizována již během první epochy. Je však třeba mít na paměti, že v porovnání s reálnými situacemi se jedná o síť s velmi málo parametry a data v Iris jsou jednoduše lineárně separovatelná, takže podobný výsledek byl předvídatelný. I tak je však dobře vidět síla a efektivita gradientního přístupu k optimalizaci.

**4. Vícevrstvý perceptron**

**4.1 Inspirace a historický kontext**

*Vícevrstvý perceptron* (MLP – *multilayer perceptron*) je typ hluboké neuronové sítě, která se skládá z neuronů uspořádaných ve více vrstvách. Po představení MCP modelu neuronu a následné první implementaci Rosenblattova perceptronu v padesátých letech začal rychle upadat zájem o neuronové sítě. Nikdo totiž nedokázal vymyslet efektivní způsob, jakým trénovat více perceptronů ve více vrstvách. V roce 1986 však D.E.Rumelhart, G.E. Hinton a R.J. Wiliams ve své práci zpopularizovali algoritmus *zpětného šíření* (*backpropagation*)1 a obnovili tak ztracený zájem nejen většiny vědecké komunity, ale později i mnoha amatérů. V posledních několika letech došlo k obrovským pokrokům ve výzkumu hlubokých neuronových sítí obecně, hlavně díky posunu ve výrobě výkonných grafických karet, které jsou klíčové pro náročné výpočty. Se složitějšími architekturami tak dnes mohou pracovat i naprostí amatéři pouze za použití svého osobního počítače.

**4.2 Architektura MLP**

Diagram

Description automatically generated

Obr. 4.1 Příkladné schéma architektury vícevrstvého perceptronu (MLP)

První dvě kapitoly této práce se týkaly jednovrstvých neuronových sítí (perceptron a ADALINE). MLP je na rozdíl od těchto algoritmů 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. Takové hluboké sítě, které nejsou cyklické, označujeme jako *dopředné* (*feedforward*).

**4.2.1 Vrstvy**

Vrstvy neuronových sítí slouží k předávání a transformaci vstupních dat pomocí aktivačních funkcí. Existuje mnoho způsobů, jakými se dají tyto vrstvy nastavit podle zamýšleného využití (například rozpoznávání vizuálních nebo zvukových dat, klasifikace, či generování nových dat). 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*). Neuronová síť je označována za hlubokou právě tehdy, když má alespoň dvě skryté vrstvy. 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 právě kvůli nelinearitě, což je důležité při řešení složitějších problémů. Na obrázku 4.1 je vidět schéma MLP se dvěma skrytými vrstvami. Opět zdůrazňuji, že se jedná o síť hustou, takže každý neuron jedné vrstvy je propojen se všemi neurony druhé vrstvy.

**4.2.2 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. Na obrázku 4.2 je vidět hustá síť s jednou skrytou vrstvou (vstupní vrstva je plně propojena se skrytou vrstvou, ta je zase plně propojena s vrstvou výstupní). 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, které se budeme snažit optimalizovat během učení sítě (stejně jako u ADALINE a perceptronu). Obrázek 4.2 též ukazuje, jakým způsobem se váhové koeficienty indexují – horní index značí vrstvu, dolní indexy představují pořadí propojených neuronů v daných Diagram

Description automatically generatedvrstvách.

Obr. 4.2 Hustě propojená síť s jednou skrytou vrstvou

**4.3 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. Pro zjednodušení budu celý tento proces vysvětlovat na MLP s jednou skrytou vrstvou, avšak postup je v tomto případě obecný a lze ho aplikovat i na hluboké sítě tohoto typu.

**4.3.1 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. Stejně jako u perceptronu musíme nejdříve vypočítat lineární vstup daného neuronu, a poté na něj aplikovat aktivační funkci. Stejně jako vstupní vrstva, skrytá vrstva též obsahuje jednotku biasu . Jelikož se jedná o síť hustou, k výpočtu lineárního vstupu libovolného neuronu ve skryté vrstvě sečteme násobky všech neuronů vstupní vrstvy (včetně biasu s násobkem 1) a jejich příslušných vah (příslušnou váhou se myslí konkrétní synapse mezi oběma neurony):

nebo:

V dalším kroku aplikujeme na lineární vstup aktivační funkci . Často používanou aktivační funkcí v případě MLP je například logistická aktivační funkce (sigmoida), která transformuje vstup na hodnotu od 0 do 1:

**Diagram

Description automatically generated with medium confidence**

Obr. 4.3 Graf logistické aktivační funkce (sigmoidy)

Hodnotu libovolného neuronu skryté vrstvy můžeme tedy zapsat jako:

**4.3.2 Aktivace výstupní vrstvy**

Výpočet hodnot neuronů výstupní vrstvy je naprosto stejný jako u skryté vrstvy. Výstupní vrstva samozřejmě na rozdíl od vstupní a skryté neobsahuje bias, ale pouze výstupní neurony Opět připomínám, že MLP je hustá síť, takže všechny neurony výstupní vrstvy jsou propojeny se všemi neurony předchozí skryté vrstvy. Hodnota libovolného výstupního neuronu je tedy opět:

Je důležité zvolit správnou aktivační funkci . Její výběr je určen tvarem výstupu, který od sítě požadujeme. Můžeme například opět použít logistickou aktivační funkci – výstupní neuron s nejvyšší hodnotou bude poté představovat predikovanou třídu. Pokud bychom však chtěli vidět procentní pravděpodobnosti jednotlivých neuronů **A(out)** výstupní vrstvy (tento přístup je obvykle označován jako *pravděpodobnostní hluboké učení* – *probabilistic deep learning*), tedy pravděpodobnost, s kterou daný příklad spadá do konkrétní kategorie, bylo by výhodnější použít funkci *softmax*. Ta sice stejně jako sigmoida transformuje vstup na hodnotu od 0 do 1, ale zároveň bere do úvahy všechny výstupní neurony (v případě výstupní vrstvy se tak jedná o ostatní třídy). Součet hodnot všech neuronů ve vrstvě s aktivační funkcí softmax bude 1, takže použijeme-li tuto funkci ve výstupní vrstvě místo sigmoidy, můžeme hodnoty neuronů interpretovat jako pravděpodobnosti:

Proměnná ***K***ve jmenovateli vzorce představuje počet tříd. Celý jmenovatel je tzv. *normovací konstanta* (*normalization term*), který je zodpovědný za onu pravděpodobnostní distribuci (součet výsledných hodnot vyjde jako 1). V případě určování z více než 2 tříd se s použitím softmax jedná o *mnohonásobnou logistickou regresi* (*multinomial logistic regression*).9

Poté, co data projdou všemi vrstvami sítě, je potřeba vypočítat ztrátové skóre a následně posílit nebo oslabit synapse mezi neurony (optimalizovat váhové koeficienty).

Diagram

Description automatically generated

Obr. 4.4 Schéma MLP s jednou skrytou vrstvou se zvýrazněnými biasy

**4.4 Ztrátová funkce – křížová entropie**

Jak jsem již vysvětloval ve 2. kapitole, jedním z hlavních pilířů učení neuronových sítí je ztrátová funkce. Ta slouží k výpočtu ztrátového skóre, které se následně snažíme minimalizovat různými metodami optimalizace pro co nejlepší výkon sítě – v případě ADALINE se jednalo o jednoduchou kvadratickou sumu odchylek. Samozřejmě nejsme vázání pouze jednou ztrátovou funkcí na každý typ sítě, v praxi se s výběrem hodně experimentuje. Při MLP můžeme použít například ztrátovou funkci (v teorii informace je tato funkce označována je křížová entropie **–** „cross entropy“**):**

Výraz představuje logistickou aktivaci *i*-tého příkladu1, o které jsem mluvil v podkapitole 4.3.1. Tuto funkci ještě musíme upravit tak, aby brala v úvahu všechny výstupní aktivační jednotky **t** naší sítě:

4.4.1 Regularizace ztrátové funkce

V tomto odstavci zmíníme často používaný způsob, jak dále zdokonalit výkon sítě, zejména v souvislosti s efektem přetrénování. Přetrénování vlastně znamená, že síť se během učení nastavuje na přesný vzorec trénovací množiny, tedy doslova si síť přehnaně věrně zapamatuje a reprodukuje trénovací příklady. Velmi dobře tedy zafunguje právě na těchto příkladech, ale na jakékoliv odchylce od nich je její odhad velmi špatný. Mluvíme pak o *generalizační chybě* (chybě zevšeobecňování příkladů), což je vlastně jeden z ústředních problémů neuronových sítí. 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. Způsob, jak toto potlačit, je přičíst ke ztrátové funkci člen úměrně závisející na velikosti všech synapsí (vah). Následná optimalizace ztrátové funkce (její minimalizace) bude tedy usilovat o rovnoměrné zmenšení všech vazeb, a tím také k vyrovnávání jejich vzájemné velikosti. 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.

Ztrátová funkce s L2 regularizací bude tedy vypadat takto:

neboli:

**+**

Při použití metody gradientního sestupu se při následném derivování ztrátové funkce součet kvadrátů změní na prostý lineární součet a zjednodušeně řečeno se tedy v algoritmu při optimalizaci bude jednat pouze o přičtení váženého součtu všech váhových koeficientů vrstvy ***l***. Použití L2 regularizace není zásadní pro všechny typy sítí, avšak výkon modelu se často může výrazně zlepšit (jak jsem již zmínil, L2 značně snižuje šanci na přetrénování sítě).

**4.5 Zpětné šíření chyby**

*Algoritmus zpětného šíření chyby* (*backpropagation algorithm*) je velmi mocný nástroj pro optimalizaci váhových koeficientů mnoha typů neuronových sítí. V podstatě stojí v pozadí většiny současných optimalizačních algoritmů. Odvození tohoto postupu není až tak technicky náročné, nicméně je poněkud zdlouhavé a málo přehledné, protože pracuje doslova v „džungli indexů“. Naštěstí mají výsledné vztahy docela jasnou interpretaci, a dávají tak vhled do toho, co se v síti při výpočtu děje.

O co se tedy v tomto algoritmu jedná? Jde o to, že při metodě gradientního sestupu je třeba znát gradient, tedy soubor parciálních derivací ztrátové funkce podle všech vah a biasů v síti (viz podkapitola o GD – při hledání minima v krajině vah a biasů se pohybujeme v opačném směru gradientu podle těchto parametrů). V případě varianty jednovrstvého perceptronu – ADALINE – to bylo ještě jednoduché, gradient ztrátové funkce – tedy příslušné derivace – bylo jednoduché spočítat, protože ztrátová funkce byla přímo funkcí příslušných vah a jejich derivace byla tedy lineární (viz příslušný výpočet v případě funkce SSE). Nicméně v případě skutečných vícevrstvých neuronových sítí narážíme na tento výpočetní problém: ztrátová funkce je funkcí aktivací výstupních neuronů, jejichž aktivace je funkcí aktivací neuronů předešlé vrstvy, jejich aktivace je funkcí aktivací neuronů předešlé vrstvy… atd. atd., přičemž máme najít derivace ztrátové funkce podle vah v každé libovolné vrstvě. Matematicky se jedná o problém derivování mnohanásobně složené funkce, což vede na součiny postupně se hromadících aktivací v pořadí od poslední – výstupní vrstvy. Přeskočíme tedy dlouhý odvozovací řetězec plný stromovitých indexovaných struktur a podíváme se na výsledek v případě naší modelové dvouvrstvé sítě.

Nazvěme výstupní chybou sítě prostý rozdíl výstupu sítě ***a(out)*** a správného cílového výsledku ***y*** daného příkladu, přičemž bereme v úvahu vektorový charakter celé výstupní vrstvy:

***δ(out) = a(out) – y***

Pak je chyba skryté vrstvy dána výrazem:

***δ(h) = δ(out)(W(out))T ʘ***

kde ***δ(h)*** je opět vektor chyb celé skryté vrstvy a symbol ***ʘ*** označuje tzv. Hadamardův součin (*Hadamard product*), který není nic jiného, než součin obou vektorů po jednotlivých složkách (tedy výsledek je nový vektor, jehož i-tá složka je součinem i-tých složek jednotlivých vektorů v součinu). Faktor derivací v tomto součinu závisí na tvaru aktivační funkce a může být tímto tvarem zjednodušen, což je právě případ sigmoidy, nicméně pro další úvahu není tento faktor důležitý. Důležité je, že vektor chyb skryté vrstvy je získán maticovým násobením vektoru výstupní chyby převrácenou maticí vah (transpozice). Můžeme si tento součin ručně rozepsat, ale názorněji vidíme přímo na obrázku níže, že chybu skryté vrstvy získáme, jako bychom chybu výstupní vrstvy tlačili naopak (proti proudu dopředného výpočtu) i s násobením příslušnými váhovými koeficienty (tato symetrie je dána právě převrácením – transpozicí matice vah). Takto se výstupní chyby „protlačí“ přes všechny váhové vektory ***W*** zpět k neuronu. Jeho chybu počítáme všemi možnými cestami, které jsou v síti možné (stromovitá struktura synapsí), a nakonec se všechny možné příspěvky od těchto různých cest sečtou. Právě proto se metoda jmenuje „algoritmus zpětného šíření chyby“. Zahlédnout tento postup ve výše zmíněné formuli vyžaduje trochu práce a cviku, proto volím tento jazykový a obrázkový opis. 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. Pro úplnost uvádím jejich formu:

***J(W) =***

***J(W) =***

kde ajsou aktivace j-tého neuronu skryté, respektive vstupní vrstvy. Tyto vztahy fungují přímo pro naši demonstrační dvouvrstvou síť, nicméně v případě vícevrstvých sítí to je naprosto obdobné. Vztahy mezi vstupní, skrytou a výstupní vrstvou jsou pak obecně vztahy mezi jakoukoliv vrstvou předcházející dané vrstvě, danou vrstvou a vrstvou následující.

Tako podkapitola była trochu náročnější na představivost, nicméně to asi dobře demonstruje, proč tento algoritmus „backpropagation”, ačkoliv je již skoro 40 let teoreticky známý, mohl dojít svého praktického uplatnění až s příchodem rychlých a velkokapacitních grafických karet a moderního optimalizačního software.



Obr. 4.9 Popsané schéma algoritmu zpětného šíření chyby

**4.6 Souhrn**

MLP (vícevrtsvý perceptron) je umělá neuronová síť schopná provádět multinomiální klasifikaci. Skládá se z vstupní vrstvy (příznaky příkladu), jedné nebo více skrytých vrstev (kvůli přidání nelinearity) a výstupní vrstvy (ta obsahuje výsledné hodnoty). Každá vrstva (kromě vstupní) aplikuje na hodnoty svých neuronů (lineární vstupy) jisté aktivační funkce. Jelikož MLP je hustě propojená síť, každý neuron jedné vrstvy je propojen s každým neuronem další vrstvy synapsemi, které představují váhy. Všechny vrstvy kromě výstupní obsahují také bias (synapse s biasem mají hodnotu 1). 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. Trénování opět probíhá v epochách. Na konci toho procesu je model schopen provádět multinomiální (nebo binární) klasifikaci.

**4.7 Implementace**

Implementace MLP v Pythonu bude nejsložitější a výpočetně nejnáročnější ze všech dosavadních příkladů. To nám však poskytuje dobrou příležitost k demonstraci síly a efektivity populárního frameworku *Keras* v rámci knihovny *TensorFlow* (viz kapitola 1). Jak jsem již vysvětloval v úvodu této práce, jedná se o knihovnu, která byla vyvinuta společností Google specificky pro účely strojového učení. Uživatelé v ní dokážou jednoduše implementovat obrovské množství běžných algoritmů i velmi složitých neuronových sítí, jejichž trénování probíhá relativně rychle díky skvělé optimalizaci výpočtů, kterou TensorFlow zprostředkovává. Framework Keras poté slouží jako uživatelsky přívětivější nadstavba této knihovny. V následujícím příkladu tedy nejdříve představím implementaci MLP v prostém Pythonu, a poté pro porovnání představím ten stejný algoritmus napsaný ve frameworku Keras.

**4.7.1 MNIST**

Další výhodou MLP oproti předchozím příkladům je kromě lepší optimalizace vah schopnost multinomiální klasifikace. Pro demonstraci rozdělování dat do více tříd budu proto využívat databázi MNIST namísto Iris. Jedná se velmi oblíbenou databázi obsahující 60 000 trénovacích a 10 000 testovacích černobílých obrázků rukou psaných číslic s rozměry 28x28 pixelů. V následujícím příkladu budu pracovat celkem s 10 třídami (číslice 0-9), takže se nejedná o algoritmus binární klasifikace, jako tomu bylo perceptronu a ADALINE, ale jde o již zmíněnou multinomiální klasifikaci. MLP se tedy bude snažit naučit, jak vypadají jednotlivé číslice, aby je následně dokázal přečíst. MNIST lze v Pythonu načíst mnoha způsoby, já v tomto příkladu pracuji se soubory staženými z této stránky: <http://yann.lecun.com/exdb/mnist/>

O zpracování stažených souborů tak, aby se poté daly jednoduše načítat z jediného souboru se stará následující kód:



Kód 7 – load\_mnist.py

**4.7.2 MLP v prostém Pythonu**

Implementace MLP v prostém Pythonu je o poznání složitější než u předchozích příkladů (hlavně kvůli zpětnému šíření chyby), proto se pokusím vysvětlit hlavně klíčové části programu, aby tato kapitola nebyla zbytečně příliš dlouhá. Neuronovou síť reprezentuje třída *MLP\_NeuralNetwork()*:



Kód 5 – MLP.py

Funkce parametrů třídy *MLP\_NeuralNetwork()* jsou z kódu celkem jasné. Matoucí by mohl být parametr *minibatch\_size*, který určuje, kolik trénovacích příkladů bude obsahovat každá dávka, na kterých bude jednotlivě probíhat gradientní optimalizace. Tento přístup je výpočetně efektivnější než práce s celým souborem najednou (viz kapitola 3). O trénování algoritmu se opět stará (tentokrát poměrně složitá) metoda *fit(X\_train, y\_train, X\_valid, y\_valid)*:



Kód 5 – MLP.py

Tato metoda je dále rozdělena na celkem tři hlavní části. Nejdříve probíhá náhodná inicializace vah a výpočet aktivací jednotlivých neuronů (skrytá i výstupní vrstva používají jako aktivační funkci sigmoidu):



Kód 5 – MLP.py

Jak je z kódu vidět, hodnoty všech biasů *self.b\_h* a *self.b\_out* jsou na počátku trénování určeny jako 0. Dále je deklarován atribut *self.eval\_*, který bude sloužit k ukládání evaluačních dat z každé epochy (ta nám budou sloužit ke sledování pokroku v učení algoritmu). Tento kód zároveň odkazuje na metodu *\_onehot(y, n\_classes)*:



Kód 5 – MLP.py

Tato metoda převádí vektor cílových tříd na matici s one-hot reprezentacemi těchto tříd. One-hot je v oblasti strojového učení běžně používané kódování, které každou cílovou třídu převede na vektor nul, kde jediná jednička představuje onu třídu. Například cílové třídy 0-5 by ve one-hot kódování vypadaly takto:

Tímto způsobem se dá celý vektor cílových tříd trénovací datové množiny zapsat jako matice 10x60000. Druhou částí metody *fit(X\_train, y\_train, X\_valid, y\_valid)* je výpočet zpětného šíření chyby:



Kód 5 – MLP.py

Poslední částí je evaluace modelu. V této částí algoritmus vypočítá momentální ztrátové skóre a přesnost predikcí na trénovací a validační datové množině. Validační příklady slouží k objektivnímu posouzení přesnosti predikce na nových (předtím neviděných) datech během trénování (testovací příklady se používají až s natrénovaným modelem):



Kód 5 – MLP.py

Nebudu zde popisovat pomocné metody pro výpočet dopředného šíření, ztrátové a logistické aktivační funkce, jelikož se jedná pouze o přepsání uvedených rovnic do Pythonu. Veškerý kód je uveden v příloze s podrobnými komentáři. K natrénování sítě jsem použil následující kus kódu:



Kód 6 – mlp\_mnist.py

Chart

Description automatically generatedNásledující graf 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:

Obr. 4.5 Graf zobrazující průměrné hodnoty křížové entropie během 300 epoch

Na grafu 4.6 zase můžeme vidět rozdíl mezi přesnostmi predikcí na trénovacím a validačním datovém souboru během učení (právě tento rozdíl demonstruje, proč je dobrou praxí používat validační množinu k objektivnímu zhodnocení pokroku sítě během jejího trénování):

Chart

Description automatically generated

Obr. 4.6 Graf znázorňující rozdíl mezi přesnostmi predikcí na trénovací a validační datové množině během trénování

Pro ukázku fungování natrénovaného MLP můžeme na obrázku 4.7 vidět číslo 7, které natrénovaný model určil správně. Na obrázku 4.8 se zase nachází číslo 2, které neuronová síť chybně určila jako číslo 9. Jedná se o číslice z testovací části databáze MNIST:

Chart, scatter chart

Description automatically generatedChart, histogram

Description automatically generated

Obr 4.8 Číslice 2 chybně určená jako číslo 9

Obr 4.7 Správně rozpoznaná Číslice 7

Přesnost natrénovaného MLP byla 98,06 %.

**4.7.3 MLP v TensorFlow Keras**

V minulé kapitole jsme si ukázali implementaci MLP v prostém Pythonu. Nyní chci demonstrovat sílu a efektivitu frameworku Keras v rámci knihovny TensorFlow. Nejdříve musíme importovat potřebné balíčky z Kerasu:



Kód 8 – Keras\_MLP.py

Jak můžeme vidět, framework *keras* je importován z knihovny *tensorflow*. Na prvním řádku je importována databáze MNIST přímo z frameworku Keras, což je daleko efektivnější způsob načítaní tohoto souboru, než jaký jsem použil při implementaci MLP v prostém Pythonu. Dále je z Kerasu importován *Sequential* model (jedná se lineární typ modelu) a *Dense* vrstva (hustá – plně propojená). Funkce *to\_categorical* slouží k zakódování cílových tříd podle one-hot.

Následující část kódu se věnuje přípravě datového souboru:



Kód 8 – Keras\_MLP.py

Poté zbývá pouze sestavit model a spustit trénování:



Kód 8 – Keras\_MLP.py

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. Je nepochybně kratší, jednodušší na čtení, výpočetně efektivnější a snadno upravovatelný – to vše díky knihovně TensorFlow a frameworku Keras. Přesnost tohoto MLP byla 95,5 % a ztrátové skóre na konci trénování 0,15.

**5. Závěr**

První dvě kapitoly této práci vysvětlily základní i některé pokročilejší algoritmy a koncepty z oblasti neuronových sítí. Obě kapitoly byly obohaceny o detailní matematická odvození a vysvětlenou implementaci uvedených algoritmů v prostém programovacím jazyku Python. V poslední kapitole jsme udělali skok od jednoho umělého neuronu a neuronové síti skládající se z více vrstev a vysvětlili jsme si, jakými matematickými pravidly se řídí její trénování. Následně jsem opět provedl implementaci v prostém Pythonu s praktickou ukázkou automatického rozpoznávání rukou psaných číslic. Zároveň jsem demonstroval sílu a efektivitu moderní knihovny TensorFlow a frameworku Keras pro strojové učení, když jsem naprosto stejnou síť s použitím těchto technologií implementoval výrazně jednodušším způsobem.

**Seznam použité literatury**

1 - RASCHKA, Sebastian a Vahid MIRJALILI. *Python machine learning: machine learning and deep learning with Python, scikit-learn, and TensorFlow*. Third edition. Birmingham: Pack publishing, [2019]. ISBN 978-1-78712-593-3.

2 - CHOLLET, François. *Deep learning v jazyku Python: knihovny Keras, Tensorflow*. Přeložil Rudolf PECINOVSKÝ. Praha: Grada Publishing, 2019. Knihovna programátora (Grada). ISBN 978-80-247-3100-1.

3 - RAMACHANDRAN, V. S. *Mozek a jeho tajemství, aneb, Pátrání neurovědců po tom, co nás činí lidmi*. Praha: Dybbuk, 2013. ISBN 978-80-7438-080-8

4 - JONTY, S. The MCP Neuron. [ONLINE] **2017**. https://jontysinai.github.io/jekyll/update/2017/09/24/the-mcp-neuron.html (accessed Aug 04, 2022)

5 - SAGAR, S. What the Hell is Perceptron? | The Fundamentals of Neural Networks. *Towards Data Science* [ONLINE] **2017**. https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53 (accessed Aug 04, 2022)

6 - RASCHKA, S. What is the difference between a Perceptron, Adaline, and neural network model? *Machine Learning FAQ* [ONLINE] **2022**. https://sebastianraschka.com/faq/docs/diff-perceptron-adaline-neuralnet.html (accessed Aug 04, 2022)

7 - ROBERT, K. Gradient Descent Algorithm — a deep dive. *Towards Data Science* [ONLINE] **2021**. https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21 (accessed Aug 04, 2022)

8 - Hidden Layer Definition. *DeepAI* [ONLINE] **2022**. https://deepai.org/machine-learning-glossary-and-terms/hidden-layer-machine-learning (accessed Aug 04, 2022)

9 – WOOD, T. Softmax Function Definition. *DeepAI* [ONLINE] **2022**. https://deepai.org/machine-learning-glossary-and-terms/softmax-layer (accessed Aug 04, 2022)

**Zdroje obrazových příloh**

Obr. 1.1

<https://flatironschool.com/blog/deep-learning-vs-machine-learning/>

Obr. 1.2

<https://www.pinterest.com/pin/281334307959053006/>

Obr. 2.1

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch02/images/02_01.png>

Obr. 2.2

<https://sebastianraschka.com/pdf/lecture-notes/stat479ss19/L03_perceptron_slides.pdf>

Obr. 2.3

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch02/images/02_02.png>

Obr. 2.4

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

Obr. 2.5

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch02/images/02_03.png>

Obr. 2.6

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch01/images/01_08.png>

Obr. 3.1

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

Obr. 3.2

<https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21>

Obr. 3.3

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

Obr. 3.4

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

Obr. 3.5

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

Obr. 3.6

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

Obr. 3.7

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

Obr. 3.8

<https://medium.com/analytics-vidhya/stochastic-gradient-descent-1ab661fabf89>

Obr. 4.1

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

Obr. 4.2

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch12/images/12_02.png>

Obr. 4.3

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch12/images/12_04.png>

Obr. 4.4

<https://github.com/rasbt/python-machine-learning-book-3rd-edition/blob/master/ch12/images/12_03.png>

Obr 4.9

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. 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. 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. Keras\_MLP.py

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