INFORMATIKA

Strojové učení a deep learning

PETR OSIČKA

Přírodovědecká fakulta UP, Olomouc

Deep learning je soubor metod z důležité oblasti umělé inteligence s mnoha aplikacemi – strojového učení. Metody strojového učení umožňují počítačovým programům zlepšovat se ve své činnosti s tím, jak získávají zkušenosti. To se hodí zejména v situacích, kdy je obtížné pro danou úlohu najít přesný algoritmus, který by spolehlivě fungoval. Takto obtížné jsou například úlohy, které lidé řeší intuitivně správně: rozpoznávání objektů na obrazcích, rozpoznání nálady člověka podle obličeje apod. Metody deep learning jsou specifické tím, že doménu úlohy, kterou řeší, reprezentují jako hierarchii konceptů, ve které jsou komplikované koncepty tvořeny jako kombinace jednodušších, a abstraktnější koncepty jako kombinace těch méně abstraktních. Jméno deep learning se odvozuje od toho, že tato hierarchie má mnoho vrstev. Deep learning zažil v několika minulých letech velký boom, díky novým algoritmům, specializovanému hardwaru i možnosti provádět výpočty na grafických kartách. Mezi významné aplikace, které se dostaly do povědomí veřejnosti, patří zejména ty zabývající se analýzou obrazových dat: automatické řízení automobilu, případně automatické úpravy obrázků a videa, například takzvané deepfakes.

V článku se budeme nejdříve zabývat strojovým učením obecně. Uvedeme, jaké typy úloh lze pomocí strojového učení řešit, a na jaké problémy přitom můžeme narazit. Krátce představíme i dva konkrétní příklady metod strojového učení – lineární regresi a rozhodovací stromy. Nakonec popíšeme jednu metodu strojového učení, kterou můžeme zařadit do deep learning a na ní ukážeme, v čem je deep learning v rámci strojového učení specifický.

1. Strojové učení

Nejdříve se pokusíme definovat, co to znamená, že se počítačový program učí ze zkušeností. Definice to bude velmi obecná, nicméně všechny její části ve zbytku článku upřesníme.

> *Řekneme, že se počítačový program řešící nějakou úlohu učí ze zkušeností, pokud se výkon tohoto programu měřený pomocí nějaké metriky P zlepšuje s tím, jak program získává zkušenosti.*

Nejdříve si ujasníme, jaký typ úloh může program schopný se učit řešit.

Klasifikace Programu předložíme nějaký objekt a program rozhodne, do které z konečného počtu kategorií daný objekt patří. Program tak většinou počítá funkci $f\colon \mathbb{R}^n \to \{1, 2, 3, \ldots, k\}$. Objekt na vstupu je popsán pomocí vektoru n reálných čísel a kategorie jsou očíslovány pomocí $1, 2, \ldots, k$. Příkladem klasifikační úlohy je určení, který z k předmětů se nachází na fotografii, či zda se daný předmět na fotografii nachází nebo ne (v tomto případě máme pouze dvě kategorie – ano a ne). V tomto konkrétním příkladě jsou vstupem algoritmu hodnoty barevných složek jednotlivých pixelů fotografie. Dalším příkladem je filtrování spamu, kdy je vstupem text mailu a program podle něj rozhodne, jestli je mail spam. Variantou klasifikace je klasifikace s chybějícími hodnotami, kdy vstupní objekt není popsán úplným vektorem reálných hodnot, ale některé hodnoty chybí. Taková úloha se může vyskytnout například ve zdravotnictví. Chtěli bychom, aby program pro predikci diagnózy pacienta na základě výsledků jeho testů byl schopen fungovat, i když výsledky některých testů nemá k dispozici.

Regrese Úloha regrese je velmi podobná klasifikaci. Liší se pouze v tom, že výsledek programu nemusí být jedno z předem určené konečné množiny čísel, ale libovolné reálné číslo. Program tedy realizuje funkci $f\colon \mathbb{R}^n \to \mathbb{R}$. Příkladem regresní úlohy je predikce budoucí ceny nějaké komodity (regrese je využívána například v algoritmickém obchodování) nebo predikce toho, kolik klient přinese bance nebo pojišťovně zisku.

Strukturovaný výstup Program převádí vstupní objekt na jiný objekt, který není číslem, ale má komplikovanější strukturu. Mezi úlohy se strukturovaným výstupem patří například transkripce, při níž převádíme vstupní objekt do textové podoby. Příkladem jsou strojové rozpoznávání znaků (OCR), rozpoznávání SPZ na fotografiích automobilů, rozpoznávání

popisných čísel z fotografií domů (prováděno například v programu Google Street View), převod mluveného slova na text či do jiné srozumitelné podoby. Zajímavé jsou i úlohy, kdy převádíme textová data opět na text, například automatický překlad mezi různými přirozenými jazyky (příkladem je Google Translate). Počítačové zpracování přirozeného jazyka je jednou z oblastí, kde metody strojového učení nachází nejvíce uplatnění.

Detekce anomálií Program prochází sekvencí objektů a označí objekty, které jsou pro danou sekvenci netypické. Pokud se například program naučí rozpoznávat obvyklé chování majitele kreditní karty, může při jejím zneužití upozornit banku na nezvyklé chování. Banka pak může kartu zablokovat, kontaktovat majitele apod.

Syntéza a vzorkování Programu předložíme množinu dat a pak chceme, aby generoval (náhodná) data podobná těm, která jsme mu předložili. Takový program se potom hodí v situacích, kdy potřebujeme generovat velké množství dat, jejichž ruční vytvoření by bylo nákladné nebo zdlouhavé. Například pro generování textur velkých objektů v počítačových hrách, generování náhodných obrázků apod. Do této kategorie můžeme také zařadit řečovou syntézu, kdy počítač podle předloženého textu vytváří mluvenou podobu daného textu.

Doplnění chybějících hodnot Programu předložíme data, ve kterých některé hodnoty chybí, a chceme, aby program chybějící hodnoty doplnil. Na rozdíl od úlohy klasifikace není pevně dáno, co má program predikovat. Doplňování chybějících hodnot lze využít například v doporučovacích systémech. Například při doporučování knih bychom chtěli uživateli doporučit knihu, která se mu bude líbit, ale kterou ještě nečetl. V datech zachycujících hodnocení knih uživateli je to, jak se mu dosud nepřečtěná knihu bude líbit, jistě chybějící hodnota.

Ve zbytku článku se pro jednoduchost zaměříme jenom na úlohy klasifikace a regrese.

2. Učení, model

Počítačový program může získat zkušenosti (tedy něco se naučit) pomocí učicího algoritmu z předložených trénovacích dat – množiny příkladů, ze kterých chceme, aby se program učil. Budeme předpokládat, že každý

příklad je vyjádřen jako vektor reálných čísel

$$\boldsymbol{x} \in \mathbb{R}^n.$$

Jednotlivým položkám $\boldsymbol{x}$ říkáme vlastnosti či atributy (anglicky features, attributes). V konkrétních aplikacích je nutné je nějakým způsobem získat ze skutečných objektů, kterými se zabýváme. Pokud se například jedná o lékařská data, kde jsou skutečnými objekty pacienti, může být v trénovacích datech pacient reprezentován jako vektor složený z pacientova tlaku, teploty apod., tedy z čísel, která jsme schopni získat pomocí lékařských testů. V úlohách klasifikace a regrese je potom každý příklad doplněn správným výsledkem. Připomeňme, že v případě klasifikace je výsledkem jedno z konečného počtu čísel, která odpovídají třídám, do kterých objekty klasifikujeme. V případě regrese je správným výsledkem libovolné reálné číslo. Úkolem učicího algoritmu je přečíst trénovací data a nějakým způsobem odvodit a zapamatovat si správný vztah mezi příklady a správnými výsledky. Učení, které pracuje takovým způsobem, říkáme učení s učitelem (anglicky supervised learning). Existují i jiné druhy učení, které jsou potřeba pro jiné úlohy než klasifikace a regrese. Učení bez učitele zpracovává data, která neobsahují správné výsledky (pojem správný výsledek v daném kontextu nedává vůbec smysl). Cílem algoritmu je naučit se vnitřní strukturu dané tréninkové množiny, která se poté dá využít; například při syntéze objektů podobných těm z trénovací množiny, při hledání anomálií apod. Posledním význačným typem učení je tzv. reinforcement learning. Při něm učicímu algoritmu nestačí pouze přečíst trénovací data, ale navíc neustále komunikuje se svým okolím a z této komunikace se dále učí. Například program pro hraní deskové hry Go, který nedávno porazil nejlepší lidské hráče, se Go naučil hrát tím, že hrál sám se sebou. Pro učení využil informace o tom, které strategie byly v těchto partiích úspěšné.

Protože se ve zbytku článku chceme zabývat klasifikací a regresí, zaměříme se pouze na učení s učitelem.

Soubor informací o tom, jakým způsobem si program pamatuje to, co se naučil, a jakým způsobem to pak využije ke klasifikaci či regresi (tedy jakým způsobem program provede výpočet, kdy pro daný vstup spočítá výstup: klasifikační třídu nebo reálné číslo) označujeme jako *model*. Obecně můžeme model definovat jako funkci

$$y = f(\boldsymbol{\theta}, \boldsymbol{x}),$$

kde $\boldsymbol{\theta}$ jsou tzv. parametry modelu. Pomocí hodnoty parametrů si model pamatuje, co se naučil. Obecně o $\boldsymbol{\theta}$ a f více říct nemůžeme. U konkrétního

modelu ovšem obojí musíme přesně specifikovat: musíme říci, jaké parametry model má, a musíme říci, jak f pro dané $\boldsymbol{\theta}$ a $\boldsymbol{x}$ spočítá y. Úlohou učicího algoritmu pak je na základě trénovacích dat vhodné $\boldsymbol{\theta}$ nalézt.

Jaké $\boldsymbol{\theta}$ chceme, aby učicí algoritmus vrátil? Pro začátek se spokojíme s tím, aby model správně klasifikoval (či počítal regresi pro) příklady z trénovacích dat. Samotné učení většinou probíhá tak, že zavedeme tzv. chybovou funkci zachycující jak dobře (resp. špatně) model pro konkrétní $\boldsymbol{\theta}$ na trénovacích datech funguje. Hodnoty parametrů poté upravujeme s cílem chybovou funkci minimalizovat.

3. Lineární regrese

Lineární regrese je model pro úlohu regrese. Model je dán jako

$$y = \boldsymbol{w} \cdot \boldsymbol{x},$$

kde jsou parametry modelu dány vektorem $\boldsymbol{w} \in \mathbb{R}^n$, $\boldsymbol{x} \in \mathbb{R}^n$ je vstup, a $y \in \mathbb{R}$ je pak výsledek. Pomocí $\cdot$ značíme skalární součin vektorů. Jednotlivé složky vektoru $\boldsymbol{w}$ můžeme interpretovat jako váhy, jaké přikládáme jednotlivým složkám $\boldsymbol{x}$. Pokud je váha některé složky $\boldsymbol{w}$ vysoká, má změna příslušné složky $\boldsymbol{x}$ velký vliv na hodnotu y. Příklad modelu můžeme nalézt na obr. 1.

Přesnost modelu měříme pomocí střední kvadratické chyby (anglicky mean squared error, zkratka MSE). Předpokládejme, že trénovací množina je složena z m dvojic ve tvaru $(\boldsymbol{x}, y)$, kde $\boldsymbol{x} \in \mathbb{R}^n$ je trénovací příklad a $y \in \mathbb{R}$ je očekávaná správná odpověď. Technicky se nám bude hodit, když příklady uspořádáme do matice $\boldsymbol{X}$ o rozměrech $m \times n$ tak, že řádky $\boldsymbol{X}$ odpovídají jednotlivým příkladům. Jim odpovídající správné odpovědi uspořádáme do vektoru $\boldsymbol{y}$. Výsledky, které model spočítá pro jednotlivé příklady, pak shromáždíme do vektoru $\overline{\boldsymbol{y}}$, tak že i-tá složka $\overline{\boldsymbol{y}}$ je výsledek, který dá model pro i-tý příklad. Hodnotu MSE potom spočítáme jako

$$\mathrm{MSE}(\boldsymbol{w}) = \frac{1}{m} \sum_{i=1}^{m} (\overline{\boldsymbol{y}} - \boldsymbol{y})_i^2.$$

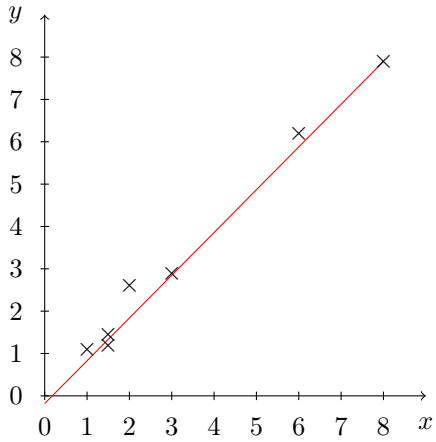
(indexem i tu označujeme i-tou složku vektoru $\overline{\boldsymbol{y}} - \boldsymbol{y}$). Vidíme, že čím více se složky vektoru $\overline{\boldsymbol{y}}$, tedy výsledky spočítané modelem, liší od příslušných složek vektoru $\boldsymbol{y}$, tedy od správných výsledků, tím je $\mathrm{MSE}(\boldsymbol{w})$ větší. Také si všimněme, že MSE uvažujeme jako funkci závislou na vektoru vah a na obsahu trénovací množiny. Při učení je ovšem trénovací množina fixní.

Trénovací množina

x	y
1	1.1
1.5	1.2
2	2.6
3	2.9
6	6.2
8	7.9

Model a jeho přesnost

Model:	$y = 1.0129 \cdot x - 0.181$
MSE:	0.086



Obr. 1 Naučený model lineární regrese. Příklady z trénovací množiny jsou reálná čísla. Model i trénovací množina je zachycena pomocí grafu vpravo. Symboly × značí jednotlivé příklady z trénovací množiny, červená přímka odpovídá naučenému modelu. Přesněji, je to množina bodů (x, y), pro které platí rovnost $y = 1.0129 \cdot x - 0.181$.

Učicí algoritmus pro lineární regresi má za cíl najít takový vektor $\boldsymbol{w}$, který minimalizuje MSE($\boldsymbol{w}$). Za tímto účelem hledáme takový vektor, pro který je gradient MSE nulový. Matematicky zručný čtenář by jistě po chvíli počítání přišel na to, že v takovém případě musí platit

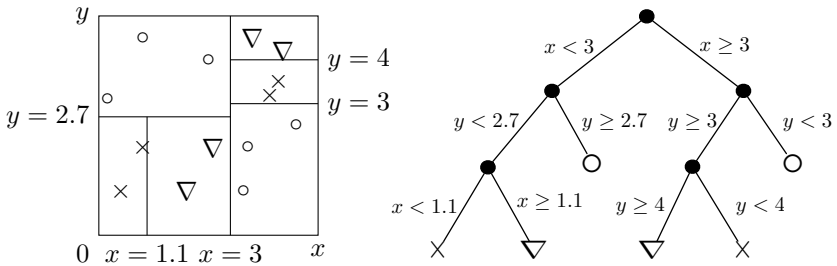
$$\boldsymbol{w} = (\boldsymbol{X}^T \boldsymbol{X})^{-1} \boldsymbol{X}^T \boldsymbol{y}.$$

Do posledního řádku dosadíme za $\boldsymbol{X}$ a $\boldsymbol{y}$ (tedy dosadíme data z trénovací množiny) a spočítáme vzniklé normálové rovnice (jak rovnice vyřešit z prostorových důvodů vynecháme).

Nakonec poznamenejme, že lineární regrese je ve skutečnosti dána

$$y = \boldsymbol{w} \cdot \boldsymbol{x} + b,$$

kde $b \in \mathbb{R}$ je další parametr, kterému říkáme *bias*. Model pořád počítá lineární funkci, jenom nyní nemusí procházet počátkem. Pro nulový vektor model spočítá hodnotu b, nikoliv 0.



Obr. 2 Příklad rozhodovacího stromu. Trénovací množina, jejíž příklady jsou tvořeny body z $\mathbb{R}^2$, je zobrazena vlevo. Body klasifikujeme do tří tříd označených symboly $\times, \nabla$ a $\circ$. Příslušný rozhodovací strom je zobrazen vpravo. Z obrázků lze vyčíst intuici na pozadí tohoto příkladu. Každému dělícímu uzlu ve stromu odpovídá jeden z obdélníků na obrázku vlevo a úsečka, která tento obdélník rozděluje na dva podobdélníky. Každý z podobdélníků odpovídá jedné hraně vedoucí z uzlu do jeho potomků. Při klasifikaci začínáme s celým obdélníkem. Při přechodu do potomka aktuálního uzlu si pak vybíráme obdélník, pro který je podmínka na příslušné hraně ve stromu pravdivá. Náš rozhodovací strom je binární, to obecně nemusí platit.

4. Rozhodovací stromy

Rozhodovací strom je model určený pro klasifikaci. Je schopen zpracovat jak numerické tak kategorické atributy. Kategorický atribut je takový atribut, který může nabývat pouze konečného počtu různých hodnot. Rozhodovací strom je kořenový strom, který obsahuje dva typy uzlů. Prvním typem uzlu je list. Každý list v rozhodovacím stromu má přiřazenu jednu klasifikační třídu, přičemž ale může mít více listů přiřazenu stejnou třídu. Zbývající uzly stromu, tj. vnitřní uzly a kořen, nazýváme dělicí uzly. Každý z nich má přiřazen jeden atribut z trénovací množiny. Každé z hran vedoucí z dělícího uzlu do jeho potomků je přiřazena jedna logická podmínka, jejíž pravdivost závisí na hodnotě atributu, který je dělicímu uzlu přiřazen. Přitom pro každou hodnotu tohoto atributu musí být pravdivá právě jedna z podmínek. Při klasifikaci procházíme strom od kořene do listu, výsledkem je pak třída, která je cílovému listu přiřazena. Přitom v každém dělicím uzlu vyhodnotíme podmínky přiřazené hranám vedoucím do jeho potomků a přejdeme do potomka, který odpovídá hraně, jejíž podmínka je pro klasifikovaný vektor pravdivá. Víme, že takový potomek je vždy právě jeden a průchod stromem je tedy jednoznačný.

Kvalitu rozhodovacího stromu můžeme určit pomocí dvou kritérií. Samozřejmě chceme, aby strom co nejlépe klasifikoval (tedy vracel co nejvíce správných výsledků). Chceme ale také, aby strom nebyl příliš velký.

Učicí algoritmy, které na základě trénovací množiny rozhodovací strom sestaví, vycházejí z následující myšlenky. Strom je konstruován postupně od kořene k listům pomocí procedury Split. Vstupem této procedury je trénovací množina, jejím výsledkem je rozhodovací strom. Procedura nejdříve vytvoří nový uzel a rozhodne, jestli bude listovým uzlem. Pokud ano, přiřadí mu příslušnou třídu a vrátí ho jako výsledek. Pokud není uzel listovým, nalezne vhodný atribut, který vytvořenému uzlu přiřadí, určí počet potomků a navrhne vhodné podmínky pro hrany do těchto potomků vedoucích. Pro každou z těchto podmínek rekurzivně zavolá Split s argumentem rovným té části tréninkové množiny, pro niž je podmínka pravdivá. Kořen stromu, který toto rekurzivní volání vrátí, pak připojí jako potomka aktuálního uzlu.

Konkrétní učicí algoritmy se liší především ve způsobu nalezení vhodného dělicího atributu, počtu potomků a jim odpovídajících podmínek. Vysvětlení konkrétního algoritmu je mimo rozsah článku, uvedeme tedy alespoň jména nejznámějších algoritmů. Jsou to ID3, C4.5, CART, CHAID, MARS. Pokročilé algoritmy během konstrukce stromu provádějí různé optimalizace, vypouštějí části stromu, které příliš nezhorší klasifikační chybu apod.

5. Přeučení a nedoučení

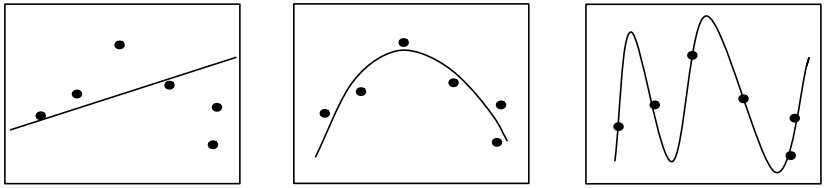
V předchozích kapitolách jsme přesnost naučeného modelu testovali na datech z trénovací množiny. Ve skutečnosti ovšem model učíme kvůli tomu, abychom uměli klasifikovat příklady, které se v tréninkové množině nevyskytují, tedy kvůli jeho schopnosti *generalizovat*. Pro další úvahy tedy musíme rozlišovat trénovací chybu a testovací chybu. Trénovací chyba je chybou, kterou měříme na příkladech z trénovací množiny, kdežto testovací chybu měříme na datech, které do trénovací množiny nepatří. Vztah mezi oběma typy chyb je předmětem bádání v oblasti nazývané *statistická teorie učení* (anglicky statistical learning theory). Abychom byli schopni něco rozumného říci, musíme zavést předpoklady o trénovacích a testovacích datech. Předpokládáme, že trénovací i testovací příklady vzorkujeme se stejným pravděpodobnostním rozložením, a že jednotlivé příklady jsou na sobě nezávislé. Po chvilce uvažování zjistíme, že očekávaná tréninková chyba by měla být menší než očekávaná testovací chyba. Po rozumném

učicím algoritmu tedy chceme, aby měl malou trénovací chybu a současně malý rozdíl mezi trénovací a testovací chybou. Pokud má model po naučení velkou trénovací chybu, říkáme o něm že je *nedoučený* (anglicky underfitted). Pokud má malou trénovací chybu, ale velký rozdíl mezi tréninkovou a testovací chybou, je model *přeučený* (anglicky overfitted).

Nedoučenost a přeučenost souvisí s *kapacitou* modelu, mírou toho jak složitá data je schopen se model naučit. Hlavní složku kapacity tvoří velikost množiny funkcí, které je model pro rozsah hodnot svých parametrů schopen počítat – tzv. *prostor hypotéz*. Například u lineární regrese tvoří prostor hypotéz lineární funkce. Pokud bychom však uvážili polynomický model stupně s jako

$$y = b + \boldsymbol{w}_1 \cdot \boldsymbol{x}^1 + \boldsymbol{w}_2 \cdot \boldsymbol{x}^2 + \ldots + \boldsymbol{w}_s \cdot \boldsymbol{x}^s$$

(index $\boldsymbol{w}$ nyní čísluje vektory, nikoli složky vektorů, položky vektoru $\boldsymbol{x}^c$ jsou c-té mocniny položek vektoru $\boldsymbol{x}$), vidíme, že se zvyšujícím se s ostře roste i prostor hypotéz. Kapacita modelu by měla odpovídat složitosti dat. Pokud je kapacita modelu menší, než je potřeba, dojde k nedoučení – model se není schopen naučit ani tréninkovou množinu. Pokud je naopak kapacita modelu příliš vysoká, může dojít k přeučení. Příklad tohoto fenoménu je na obr. 3. Podotkněme také, že kapacita modelu záleží nejenom na prostoru hypotéz, ale i na schopnosti učicího algoritmu nalézt v tomto prostoru tu správnou funkci.



Obr. 3 Kapacita, přeučení a nedoučení. Trénovací příklady jsou vybrány tak, aby ležely přibližně na kvadrice. Lineární regrese na obrázku vpravo není schopna se naučit s malou trénovací chybou. Kvadratická funkce uprostřed odpovídá složitosti úlohy, má malou trénovací i testovací chybu. Polynom vysokého stupně na obrázku vpravo je schopen se naučit s malou trénovací chybou, testovací chybu má ovšem velkou.

Na závěr kapitoly ještě uvedeme dva význačné výsledky, které statistická teorie učení přinesla. První lze bez technických podrobností formulovat následovně: *Rozdíl mezi trénovací a testovací chybou lze seshora ome-*

zit kvantitou, která roste s kapacitou modelu. Tato kvantita nicméně klesá s počtem příkladů v tréninkové množině. Pokud tedy dochází k přetrénování, můžeme se situaci pokusit vyřešit rozšířením tréninkové množiny. Druhý výsledek je tzv. No free lunch theorem, který říká, že *pokud uvážíme průměr přes všechna pravděpodobnostní rozdělení dat, pak všechny učicí algoritmy vedou ke stejné testovací chybě.* Tedy v jistém smyslu neexistuje žádný nejlepší učicí algoritmus. Naštěstí v praxi vždy pracujeme s daty, která odpovídají konkrétnímu pravděpodobnostnímu rozdělení a tomuto rozdělení můžeme učicí algoritmy přizpůsobit.

6. Deep learning

Připomeňme, že deep learning je kolekce metod, které komplexní data reprezentují jako hierarchii konceptů, v níž složitější, abstraktnější koncepty tvoříme z jednodušších, méně abstraktních konceptů. V této kapitole popíšeme nejpopulárnější model z této kategorie – *dopřednou vícevrstvou neuronovou síť*, někdy také nazývanou *vícevrstvý perceptron.*

Neuronová síť se skládá z (umělých) neuronů, malých jednotek počítajících jednoduchou funkci. Tato funkce přiřazuje vektoru $\boldsymbol{x} \in \mathbb{R}^n$ číslo $y \in \mathbb{R}$. Je definovaná (uvážíme-li ji jako model)

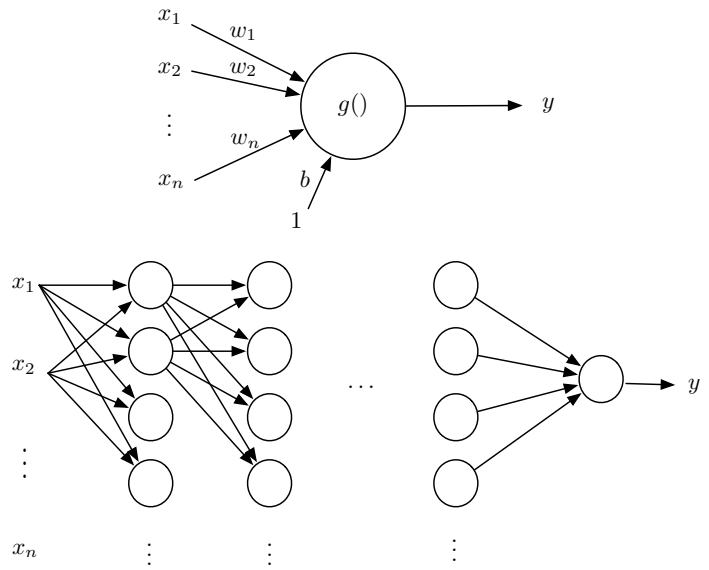
$$f(\boldsymbol{w}, \boldsymbol{x}) = g(\boldsymbol{w} \cdot \boldsymbol{x} + b).$$

Vektor $\boldsymbol{w} \in \mathbb{R}^n$ a číslo $b \in R$ jsou parametry modelu, $\boldsymbol{w}$ označujeme jako vektor *vah* a b je bias. Připomeňme, že operace $\cdot$ je skalární součin dvou vektorů. Funkce $g\colon \mathbb{R} \to \mathbb{R}$ je tzv. *aktivační funkce*, výsledné hodnotě říkáme *aktivace* neuronu. Mezi často používané aktivační funkce patří identita $g(x) = x$ (pak je neuron v podstatě lineární regresí), RLU (zkratka anglického rectifier linear unit) $g(x) = \max(0, x)$, a logistická funkce $g(x) = 1/(1 + e^x)$. Grafické znázornění jednoho neuronu lze nalézt na obr. 4.

Umělý neuron je inspirován tím, jak funguje biologický neuron v lidském mozku. Biologický neuron je buňka, která je schopna po synapsích, tj. spojeních s ostatními neurony, přijímat elektrické signály. Tyto signály potom agregovat, spočítat z nich svoji aktivaci a jako elektrický signál ji po synapsích poslat dalším neuronům.

Ve vícevrstvém perceptronu jsou neurony rozděleny do lineárně uspořádané sekvence vrstev, kde vrstvou myslíme množinu neuronů. Předpokládejme, že máme vrstvy očíslovány čísly $1, 2, \ldots, d$. Neurony z jednotlivých

vrstev jsou propojeny tak, aby vektor aktivací neuronů z vrstvy i sloužil jako vstup neuronů ve vrstvě $i+1$. V poslední vrstvě se nachází pouze jeden vektor. Všechny vrstvy mimo poslední označujeme jako *skryté*. Příklad vícevrstvého perceptronu je na obr. 4 dole.

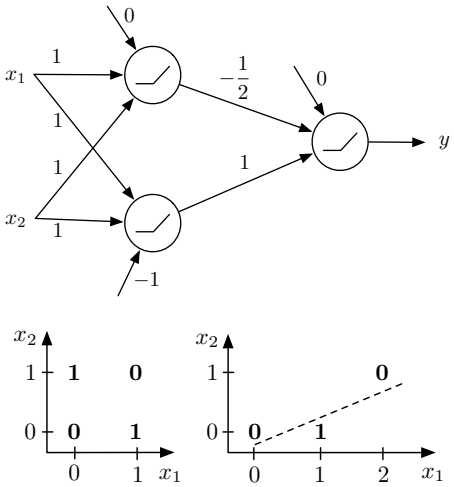


Obr. 4 Grafická reprezentace jednoho neuronu (nahoře) a dopředné vícevrstvé perceptronové sítě (dole). Váhy a některé synapse jsou pro přehlednost vynechány.

Při výpočtu vícevrstvý perceptron nejdříve použije vstup celé sítě jako vstup pro všechny neurony první skryté vrstvy, které spočítají svoje aktivace. Ty poté slouží jako vstup pro neurony druhé vrstvy, jejichž aktivace slouží jako vstup neuronů třetí vrstvy. Takto výpočet pokračuje dále až k poslední vrstvě. Aktivace jejího jediného neuronu je pak výstupem celé sítě.

Proč můžeme vícevrstvý perceptron počítat mezi metody deep learning? Pro jednoduchost uvažme situaci, kdy neuron použijeme ke klasifikaci do dvou tříd. To můžeme zařídit tak, že budeme aktivaci neuronu porovnávat s nějakou prahovou hodnotou. Pokud bude aktivace větší, vrátíme první třídu, pokud bude menší, vrátíme druhou třídu. Jeden neuron je scho-

pen správně klasifikovat pouze lineárně oddělitelné množiny vstupů, tj. množiny, ve kterých můžeme všechny příklady patřící do jedné klasifikační třídy oddělit od příkladů patřící do druhé třídy pomocí hyperroviny (připomeňme, že vstupy sítě jsou body v n-rozměrném prostoru). V případě dvourozměrného prostoru, tedy plochy, je hyperrovinou přímka. Z tohoto pohledu může v datech jeden neuron rozpoznávat jenom velmi jednoduché koncepty. Vícevrstvý perceptron toto omezení překonává. Neurony v první vrstvě sice rozpoznávají jednoduché koncepty z trénovacích dat, ale neurony v druhé vrstvě rozpoznávají jednoduché koncepty z aktivací neuronů z první vrstvy a to už nejsou jednoduché koncepty z tréninkové množiny, ale jejich kombinace! Příklad tohoto fenoménu vidíme na obr. 5, který zachycuje dvouvrstvý perceptron, který počítá logickou funkci XOR; jeden neuron to přitom nedokáže. Obecně můžeme říci, že čím hlubší vrstva je, tím složitější koncepty umí v datech rozpoznat. Výsledek celé sítě je spočítán z těch nejsložitějších.



Obr. 5 Dopředná vícevrstvá perceptronová síť počítající fukci XOR (nahoře). Jako aktivační funkce je použita RLU. Na dolním grafu zleva vidíme zobrazení funkce XOR ve dvourozměrném prostoru. Snadno vypozorujeme, že nelze nalézt přímku tak, aby v tomto prostoru oddělovala hodnoty **0** od hodnot **1**. Na vedlejším grafu jsou jednotlivé položky zobrazeny po průchodu skrytou vrstvou sítě. Osa x_1 zachycuje aktivaci horního neuronu, osa x_2 dolního neuronu skryté vrstvy. Vidíme, že nyní lze **0** a **1** oddělit přímkou.

Učicí algoritmus pro vícevrstvý perceptron je založen na *gradientní metodě*. Pokud model vyjádříme jako $f(\boldsymbol{\theta}, \boldsymbol{x})$, kde $\boldsymbol{\theta}$ jsou parametry modelu a chybová funkce L, kterou uvažujeme jako závislou na $\boldsymbol{\theta}$, je diferencovatelná (příkladem chybové funkce je například MSE z kapitoly o lineární regresi), pak můžeme spočítat její gradient $\nabla L(\boldsymbol{\theta})$. Ten ukazuje směrem, kterým chybová funkce nejvíce roste, opačným směrem tedy nejvíce klesá. Protože chceme L minimalizovat, „posuneme“ $\boldsymbol{\theta}$ tímto směrem. V algoritmu tedy nejdříve vygenerujeme počáteční parametry $\boldsymbol{\theta}_1$, pak s pomocí trénovací množiny (použijeme buď jeden příklad nebo více příkladů v závislosti na konkrétním modelu a algoritmu) spočteme gradient chybové funkce a s jeho pomocí upravíme parametry modelu vztahem

$$\boldsymbol{\theta}_{i+1} = \boldsymbol{\theta}_i - \mu \nabla L(\boldsymbol{\theta}_i)$$

(μ je učicí konstanta určující velikost „posunutí“). V případě vícevrstvého perceptronu lze gradient spočítat metodou, na kterou se lze dívat jako na zpětný výpočet sítě, kterým se vzniklá chyba šíří od nejhlubších vrstev až k první skryté vrstvě. Odtud také název algoritmu – *backpropagation*.

Velkou výhodou vícevrstvého perceptronu je, že jeho dopředný výpočet i učicí algoritmus jdou snadno paralelizovat. Výpočty aktivací všech neuronů z jedné vrstvy můžeme totiž počítat v různých vláknech a není je potřeba synchronizovat. Pouze je nutné pomocí bariéry synchronizovat výpočet mezi sousedním vrstvami. V minulých letech se také objevilo několik nových, výpočetně velmi efektivních učicích algoritmů. Pro řešení konkrétní úlohy tak lze použít síť s dostatečným počtem skrytých vrstev, kterou lze efektivně učit i používat.

Zájemcům o hlubší informace o deep learning doporučuji knihu [1], která je zdarma přístupná na adrese http://www.deeplearningbook.org/.

Literatura

[1] *Goodfellow, I., Bengio, Y., Courville, A.*: Deep Learning. MIT Press, 2016. Dostupné z: http://www.deeplearningbook.org/.