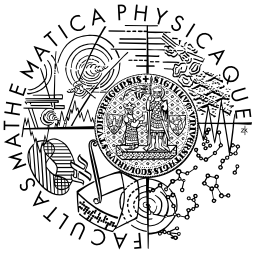


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jiří Szotkowski

Optimalizační algoritmy pro trénování neuronových sítí

Katedra numerické matematiky

Vedoucí bakalářské práce: doc. RNDr. Václav Kučera, Ph.D.

Studijní program: Obecná matematika

Studijní obor: MOMP

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Upřímně děkuji svému vedoucímu doc. RNDr. Václavu Kučerovi, Ph. D. za odborný dohled, velké množství času, které mé práci věnoval, a v neposlední řadě také za příjemně strávený čas během konzultací k této práci.

Velké díky patří také mé rodině a přítelkyni za podporu během celé doby studia.

Název práce: Optimalizační algoritmy pro trénování neuronových sítí

Autor: Jiří Szotkowski

Katedra: Katedra numerické matematiky

Vedoucí bakalářské práce: doc. RNDr. Václav Kučera, Ph.D., Katedra numerické matematiky

Abstrakt: V práci se zabýváme základním popisem neuronových sítí a jejich tréninkem, ke kterému se používají numerické optimalizační algoritmy. První kapitola popisuje fungování neuronové sítě z pohledu laika, v druhé kapitole nalezneme rigorózní, matematickou formulaci neuronové sítě a dalších souvisejících pojmů. Třetí kapitola se pak věnuje optimalizačním algoritmům, které se k trénování sítí používají, zejména jde o algoritmy odvozené z metody největšího spádu. Ve čtvrté kapitole nakonec testujeme jednotlivé optimalizační metody, a to jak na tréninku neuronových sítí, tak i na akademických příkladech.

Klíčová slova: Optimalizace, neuronové sítě, metoda největšího spádu

Title: Optimization algorithms for neural network training

Author: Jiří Szotkowski

Department: Department of Numerical Mathematics

Supervisor: doc. RNDr. Václav Kučera, Ph.D., Department of Numerical Mathematics

Abstract: This thesis presents a basic description of neural networks and optimization algorithms that are commonly used to train neural networks. The first chapter describes the functioning of a neural network from a layman's perspective, while the second chapter presents a rigorous, mathematical formulation of the neural network and other related concepts. The third chapter focuses on optimization algorithms used for training networks, particularly those derived from the method of steepest descent. Finally, in the fourth chapter, we evaluate individual optimization methods, both in training neural networks and in academic examples.

Keywords: Optimization, neural networks, steepest descent method

Obsah

Úvod

Neuronové sítě jsou v dnešní době populárním nástrojem v oblasti umělé inteligence. Jedná se o výpočetní model složený z jednotek, které nazýváme neurony. Název základní stavební jednotky sítě vznikl na základě inspirace neurony, které nalezneme v lidském mozku; o tom, kolik toho mají síťové neurony společného s neurony v lidském mozku, se dozvíme více už jen o pár stránek dále.

Podle článku [1] vznikly neuronové sítě podobné těm, které používáme dnes již v roce 1958. Tehdy Frank Rosenblatt přišel s nápadem multilayer-perceptronu, jehož struktura už vypadala jako struktura některých dnešních sítí, a který se učil v poslední své vrstvě.

Lepší tréninkové algoritmy se objevily kolem roku 1965 a v roce 1970 Seppo Linnainmaa poprvé představil algoritmus, který je dnes znám jako *zpětné šíření chyby* (angl. *backpropagation*). Tento algoritmus se dodnes používá pro výpočet gradientu ztrátové funkce neuronové sítě, a teprve znalost tohoto gradientu umožnila efektivně trénovat neuronové sítě.

Následoval další vývoj neuronových sítí a se zlepšujícím se hardwarem sítě zvládaly řešit i zajímavé počítačové úlohy, na které do té doby neexistovaly efektivní algoritmy. O velkém potenciálu neuronových sítí přesvědčil celý svět hlavně jazykový model ChatGPT, který se v roce 2023 stal populární po celém světě. Za tímto úspěchem stojí mj. i velká neuronová síť, která se nachází v jádru tohoto modelu.

Mimo jazykové modely se neuronové sítě dnes používají také pro klasifikaci a zpracování zvuku a obrazu, při kontrole procesů nebo při kontrolách kvality, aplikaci nachází také v medicínské diagnóze - klasifikace lékařských snímků. Další aplikací neuronových sítí jsou finanční predikce založené na historických datech jiných finančních nástrojů, sítě se taktéž používají v oblasti marketingu, kde se využívají pro behaviorální analýzu a cílený marketing. Poslední aplikací, kterou zde uvedeme, je identifikace jednotlivých chemických složek ve zkoumaném vzorku pomocí neuronových sítí.

V této práci od základu popíšeme z jakých komponent se taková neuronová síť skládá a jak funguje. Odvodíme také již zmíněný algoritmus zpětného šíření chyby, a především se budeme zajímat o trénování neuronových sítí, které není nic jiného než numerická optimalizace jisté ztrátové funkce. Pro tuto optimalizaci se osvědčilo používat různé varianty metody největšího spádu, které, jak si ukážeme, fungují velmi dobře i pro numerickou optimalizaci mimo neuronové sítě. Svět neuronových sítí a optimalizace prozkoumáme také z praktického hlediska, neboť vše budeme také implementovat a testovat.

V první kapitole popíšeme neuronové sítě z pohledu laika. Seznámíme se s potřebnými pojmy a pokusíme se motivovat kapitoly příští, především kapitolu druhou.

Právě v druhé kapitole si rigorózně zavedeme neuronové sítě jakožto ryze matematický objekt. Začneme zde již mluvit i o trénování neuronové sítě - důležitými objekty zde budou mj. tréninkové datasety a ztrátová funkce sítě. Připravíme si tak půdu pro odvození algoritmu zpětného šíření chyby, kterým uzavřeme druhou kapitolu.

Optimalizačním algoritmům věnujeme třetí kapitolu. Na začátku kapitoly se

budeme chvíli věnovat metodě největšího spádu a teorii s ní spojené. Právě z metody největšího spádu je totiž odvozena většina optimalizačních metod, které se dnes pro trénování neuronových sítí používají nejvíce. Pro zajímavost uvedeme také jednu nespádovou optimalizační metodu - Nelder-Meadovu simplexovou metodu. Na konci kapitoly také představíme stochastické verze dříve zmíněných optimalizačních metod, které umožňují trénovat neuronové sítě daleko rychleji a efektivněji.

Ve čtvrté kapitole se budeme zabývat testováním optimalizačních metod na třech různých úlohách. Všechny optimalizační algoritmy otestujeme na úloze hledání minima akademických funkcí. Dále otestujeme klasické verze představených optimalizačních metod při tréninku malé neuronové sítě, pomocí které budeme řešit úlohu klasifikace ropných vrtů. V této malé neuronové síti rovněž otestujeme i Nelder-Meadovu simplexovou metodu, která se pro tento účel běžně nepoužívá. Nakonec otestujeme i stochastické verze metod spádového typu, a to na kanonickém příkladě rozpoznávání rukou psaných číslic, k jehož řešení použijeme už trochu větší neuronovou síť.

Hlavními zdroji informací o neuronových sítích jsou pro tuto práci [2], [3] a [4]. Zdrojem pro optimalizační metody je převážně [5], nicméně v textu dále odkážeme i na jiné zdroje, ze kterých jsme čerpali informace specifické pro jednotlivé metody. Testovací příklad s ropnými vrty je inspirovaný podobným z [4], a data pro úlohu klasifikace rukou psaných číslic pochází z [6]. Při implementování neuronové sítě jsme se nakonec inspirovali implementací uvedenou v [2].

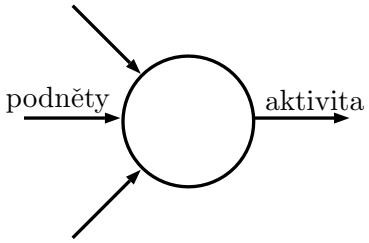
Text se snažíme psát pedagogicky, a tak od čtenáře neočekáváme žádné znalosti v oblasti umělé inteligence nebo neuronových sítí. Všechny pojmy z této oblasti budou řádně definovány a k úplnému porozumění by čtenáři mělo stačit vědět, co jsou to parciální derivace a řetízkové pravidlo.

1. Neuronové sítě laicky

V této kapitole popíšeme myšlenku, která stojí za fungováním neuronové sítě. Hlavním cílem kapitoly je pomoci čtenáři pochopit fungování neuronové sítě intuitivně a laicky. Konkrétněji a podrobněji se fungováním sítě budeme zabývat v příští kapitole. Jako doplňující výklad mohou velmi dobře posloužit [2] a [3].

Nejprve si popíšeme neuron - co si pod tímto pojmem představit a jak neuron funguje. I přesto, že se toto pojmenování pro základní stavební jednotku neuronové sítě uchytilo, neuron z neuronové sítě nemá s klasickým neuronem, který známe z biologie moc společného.

Začněme však dvěma podobnostmi. První podobností je fakt, že oba zmíněné neurony budou reagovat na určité podněty. Tou druhou, že zde, tak jako u biologických neuronů, nás bude zajímat aktivita neuronu, tj. zda je neuron aktivní nebo pasivní. V našem případě bude tato aktivita určená číslem v intervalu (0,1) - větší číslo udává větší aktivitu neuronu, nižší číslo pak znamená, že neuron je spíše pasivní.



Obrázek 1.1: Takto obvykle znázorňujeme neuron. Kulička znázorňuje tělo neuronu. Každá šipka směřující do kuličky (vždy vlevo) znázorňuje jeden podnět/vstup. Šipka směřující ven z kuličky (vždy vpravo) znázorňuje aktivitu/výstup neuronu. Zpravidla se místo šipek kreslí pouze krátké úsečky, neboť z polohy úsečky je jasné, co znázorňuje.

I přes to, že neuron obvykle nebude vykazovat aktivitu 1 nebo 0, budeme pozorovat, ke které z těchto hodnot má aktivita neuronu blíže. Neuron (popř. celou neuronovou síť) obvykle budeme chtít naučit řešit nějaký problém, minimálně budeme potřebovat aby neuron reagoval na jisté podněty, a abychom jeho reakce dokázali sledovat.

Příklad. Řekněme, že chceme pomoci s rozhodnutím, zda jít večer do kina, nebo nikoliv. Řekněme dále, že se budeme rozhodovat na základě dvou kritérií - předpovědi venkovní teploty po skončení filmu a hodnocení filmu, na který chceme jít. S problémem by nám mohl pomoci neuron, který dokáže reagovat na dva podněty, kterými budou právě dvě zmíněná rozhodovací kritéria. Každý večer bychom mohli sledovat aktivitu neuronu a rozhodnout se, zda kino navštívíme, nebo ne, podle toho, zda je neuron spíše aktivní, nebo pasivní. Tedy pokud by neuron vykazoval aktivitu 0.9, rozhodli bychom se do kina jít, naopak by tomu bylo při hodnotě aktivity rovné 0.4.

Zda bychom byli posléze spokojeni s rozhodnutím, které za nás neuron učinil už je druhá věc. Některý neuron nás může posílat na kvalitní filmy jen za hezkého počasí, jiný může činit rozhodnutí, se kterými naopak moc spokojeni nebudeme.

Úloha hledání toho správného neuronu pak blízce souvisí právě se strojovým učením. O tom ale až později.

△

Teď konkrétněji k tomu, jaké budeme klást požadavky na podněty, a jakým způsobem na ně budou neurony reagovat. Podnětů budeme požadovat vždy nejvýše konečný počet a budeme uvažovat jen číselné hodnoty podnětů. Tedy ve výše zmíněném příkladu bychom museli nalézt způsob, jak reálným číslem reprezentovat venkovní teplotu po skončení filmu a taky jak reprezentovat kvalitu filmu. Taktéž se osvědčuje daná reálná čísla odpovídající podnětům škálovat tak, aby spadala do intervalu [0,1].

Vraťme se k příkladu s kinem; rozmyslíme si, jak by se dala reprezentovat rozhodovací kritéria číslem z intervalu [0,1].

Příklad. Kvalitu filmu můžeme přirozeně reprezentovat jako průměr hodnocení kritiků na stupnici $0-100$ bodů. Vezměme tedy toto číslo, vydělme jej 100 a dostaneme číslo z [0,1] tak, jak jsme chtěli. O teplotě po skončení filmu (ve stupních Celsia) pak můžeme předpokládat, že nevypadne z intervalu $[-40, 60]$, tedy můžeme tento interval škálovat na [0,1]. Určitě existují i rozumnější způsoby, jak reprezentovat tyto vstupní údaje, berme tento příklad opravdu jen jako jednu z možností, ať už by se v praxi ukázala být dobrá, nebo spíše ne.

△

A jak může neuron na tyto podněty reagovat? Způsobů je nespočetně, nicméně my si vybereme jeden konkrétní, který se ukázal být dostačující pro řešení různých problémů z reálného světa, a který je zároveň dost jednoduchý, aby se daly provádět výpočty a úvahy, které vedou k trénování neuronové sítě.

Řekněme, že neuron má reagovat na $k \in \mathbb{N}$ podnětů, což jsou - podle výše zmíněných požadavků - nějaká čísla $a_1, a_2, \ldots, a_k \in [0, 1]$. V takovém případě si neuron bude pamatovat pevná čísla $w_1, w_2, \ldots, w_k \in \mathbb{R}$, a taky číslo $b \in \mathbb{R}$, a na podněty bude reagovat následovně. Spočítá tzv. převážené vstupy - číslo $z \in \mathbb{R}$ dané vzorcem:

$$z = w_1 a_1 + w_2 a_2 + \cdots + w_k a_k + b,$$

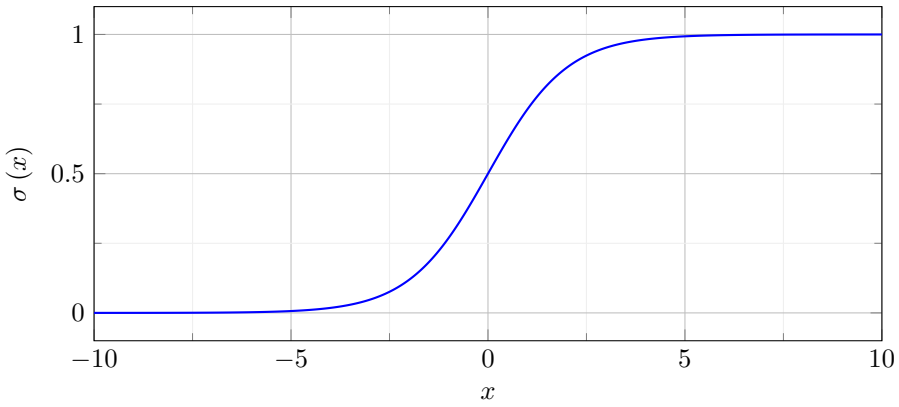
a nakonec toto číslo přeškáluje do intervalu (0,1) pomocí funkce $\sigma : \mathbb{R} \to (0,1)$ s předpisem:

$$\sigma(z) = \frac{1}{1 + e^{-z}}, \quad z \in \mathbb{R}.$$

Této funkci se často říká *sigmoida* (v anglické literatuře sigmoid function), viz Obrázek 1.2. Tedy celkem výslednou aktivitou neuronu bude číslo $a \in (0,1)$:

$$a = \sigma(z) = \sigma(w_1 a_1 + w_2 a_2 + \cdots + w_k a_k + b).$$

Číslům $w_1, w_2, \ldots, w_k$ říkáme *váhy* (někdy *synaptické váhy*, v anglické literatuře pak *weights*), neboť udávají, jak relevantní je příslušný podnět. Například, kdyby platilo $w_1 = w_2 = \cdots = w_{k-1} = 0$ a $w_k = 1$, pak zřejmě podněty $a_1, a_2, \ldots, a_{k-1}$ nemají na aktivitu neuronu vůbec žádný vliv, a ze všech podnětů ovlivňuje aktivitu neuronu pouze podnět a_k.



Obrázek 1.2: Graf funkce σ.

	$(0{,}6; 0{,}9)$	$(0{,}6; 0{,}1)$	$(0{,}3; 0{,}9)$	$(0{,}3; 0{,}1)$
aktivita n_1	0,550	0,354	0,475	0,289
aktivita n_2	0,269	0,450	0,332	0,525

Tabulka 1.1: Aktivity neuronů n_1, n_2 při podnětech $(a_1; a_2)$. Hodnoty uvádíme zaokrouhlené na tři desetinná místa.

Číslo b pak nazýváme *práh* (v anglické literatuře *bias*), a udává, zda je tendence neuronu býti spíše aktivní, nebo spíše pasivní.

Všimněme si, že váhy s prahem jednoznačně udávají aktivitu neuronu na všech možných vstupech, tj. pro všechny možné kombinace podnětů.

Vraťme se opět k příkladu s kinem.

Příklad. Uvažujme čtyři situace podle toho, kolik bude venku stupňů po skončení filmu, a podle hodnocení filmu; dále uvažujme dva neurony, jež reagují právě na tyto dva podněty. Označme venkovní teplotu po skončení filmu jako podnět a_1 a hodnocení filmu jako podnět a_2. Neuron n_1 charakterizujme vahami $w_1^1 = w_2^1 = 1$ a prahem $b^1 = -1{,}3$. Neuron n_2 charakterizujme vahami $w_1^2 = w_2^2 = -1$ a prahem $b^2 = 0{,}5$. Podívejme se na aktivity neuronů pro různé podněty a_1, a_2; konkrétně pro hodnoty $a_1 = 0{,}6$, odpovídající teplotě 20 °C, $a_1 = 0{,}3$, odpovídající teplotě -10 °C, $a_2 = 0{,}9$, odpovídající hodnocení 90 bodů ze 100, a nakonec taky $a_2 = 0{,}1$, což odpovídá hodnocení 10 bodů ze 100. Všechny hodnoty podnětů byly přeškálovány dříve zmíněným způsobem.

Jak by tedy situace dopadla, pokud bychom se rozhodovali na základě aktivity neuronů, tedy zda je neuron spíše aktivní, nebo spíše pasivní, tedy zda je příslušná aktivita neuronu větší nebo menší než 0,5? Neuron n_1 by nás poslal do kina jen v případě, že film je dobře hodnocený a zároveň venkovní teplota je příjemná, tj. jen v případě $a_1 = 0{,}6$, $a_2 = 0{,}9$, jelikož podle údajů v Tabulce 1.1 je tato konfigurace jediná, kdy je aktivita neuronu větší, než 0,5. Naopak neuron n_2 by nás z domu pustil jen v největší zimě a na nejhůř hodnocený film.

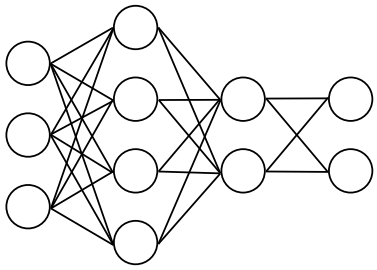
△

Kdybychom se chtěli rozhodovat na základě aktivity neuronu v praxi, potřebovali bychom tedy najít ty „správné“ váhy, ten „správný“ práh, aby nám rozhodnutí neuronu co nejlépe vyhovovalo. Později uvidíme, že tato úloha (nale-

zení optimálních parametrů, tj. vah a prahu, které dávají dobré výsledky) se dá řešit iteračně, a není třeba manuálně zkoušet nebo hledat, která kombinace dá lepší výsledek. Tento proces pak můžeme brát jako jistý „trénink“ neuronu, aby vykazoval aktivitu tak, jak my potřebujeme.

Nicméně v praxi zjistíme, že neuron samotný nám často stačit nebude. Později v této práci (viz Kapitola 4.4) budeme pomocí neuronových sítí řešit např. kanonický problém rozpoznávání rukou psaných číslic. Budeme „trénovat“ neuronovou síť, aby obrázku rukou psané číslice přiřadila prvek množiny $\{0, 1, \ldots, 9\}$ podle toho, která číslice se na obrázku nachází. Pro tento problém není moc praktické použít jeden jediný neuron, neboť z jednoho neuronu chceme brát pouze informaci, zda je spíše aktivní, nebo spíše pasivní. V praxi se osvědčuje pracovat spíše se systémem neuronů - tzv. neuronovou sítí - výstupem je pak aktivita vícero neuronů. Pro tuto úlohu bychom tedy mohli zvolit jako výstup sítě aktivitu deseti neuronů - každé číslici přísluší právě jeden neuron - odpověď neuronové sítě pak můžeme vnímat jako číslici příslušnou neuronu, který je nejaktivnější. Proto jsme taky dříve požadovali, aby aktivita neuronu byla číslo z $(0, 1)$, a ne z $\{0,1\}$; v druhém případě bychom totiž často nemohli porovnat, který z neuronů je nejaktivnější.

Přesuňme se tedy od neuronu k neuronové síti samotné. Jak už název samotný napovídá, neuronovou sítí rozumíme jisté spojení více neuronů na sebe. Nejlépe lze strukturu sítě popsat například následujícím obrázkem (Obrázek 1.3).



Obrázek 1.3: Jednoduchá neuronová síť se třemi vstupy, dvěma skrytými vrstvami skládajících se postupně ze čtyř a ze dvou neuronů, a s výstupní vrstvou se dvěma neurony.

Můžeme si všimnout, že neurony v první vrstvě (sloupeček úplně vlevo) postrádají šipky znázorňující vstupy/podněty; nejedná se totiž o neurony, ale o značení vstupního vektoru neuronové sítě. Místo první vrstvy si tedy můžeme představovat konkrétní hodnoty podnětů, tedy čísla. Od druhé vrstvy už ale na obrázku nacházíme neurony takové, na jaké jsme zvyklí z předchozího. Taky je zvykem nekreslit do obrázku neuronové sítě šipky, které přísluší aktivitám neuronu.

Nejdůležitější, co si z obrázku odnést, je ale princip, jak jsou vrstvy neuronů na sebe napojené. Každá vrstva neuronů dostává na vstupu aktivity neuronů z předchozích vrstev (aktivity neuronů z první vrstvy bereme jako podněty neuronů v druhé vrstvě atp.). Přitom každý neuron v dané vrstvě vnímá právě tolik podnětů, kolik je neuronů v předchozí vrstvě.

O první vrstvě často mluvíme jako o *vstupní vrstvě*, poslední vrstvu často nazýváme *výstupní vrstvou* a o ostatních vrstvách se říká, že jsou tzv. *skryté*.

A nyní trochu k tomu, jak se dá neuronová síť „trénovat“. Abychom dokázali neuronové síti říct, jak spokojeni s její odpovědí jsme, definujeme tzv. *ztrá-*

tovou funkci. Tato funkce vezme na vstupu dvě informace: odpověď neuronové sítě a námi očekávanou (správnou) odpověď. Těmto dvěma informacím ztrátová funkce přiřadí nezáporné reálné číslo, které vyjadřuje, jak špatná je odpověď neuronové sítě.

Příklad. Mějme neuronovou síť se strukturou jako na Obrázku 1.3. Řekněme, že neuronovou síť necháme zpracovat jistý vstup, při kterém očekáváme, že první výstupní neuron bude vykazovat aktivitu 1 a druhý neuron aktivitu 0. Pro tento konkrétní vstup můžeme uvažovat ztrátu danou funkcí $s : (0,1)^2 \to [0,\infty)$ s předpisem:

$$s\left(a_1, a_2\right) = \left(a_1 - 1\right)^2 + \left(a_2 - 0\right)^2, \tag{1.1}$$

kde a_1, a_2 jsou postupně aktivity prvního a druhého neuronu v poslední vrstvě sítě. Snadno si můžeme rozmyslet, že funkce s bude nabývat hodnot blízko nule v případě, kdy se odpověď neuronové síti blíží očekávané odpovědi; naopak, bude-li a_1 blízko nuly a a_2 blízko jedničky, bude se ztráta blížit k číslu 2.

Funkci s obvykle nazýváme *součet čtverců chyb*. Na funkci s bude založena ztrátová funkce, kterou budeme používat.

△

V předchozím příkladu přiřazovala funkce s ztrátu na jednom konkrétním vstupu jedné neuronové síti. Rozmysleme si, že funkci s předpisem (1.1) lze zobecnit na ztrátovou funkci splňující požadavky popsané v odstavci nad příkladem.

Příklad. Uvažujme opět neuronovou síť z Obrázku 1.3. Jako ztrátovou funkci můžeme definovat $s : (0,1)^2 \times \{0,1\}^2 \to [0, \infty)$ s předpisem:

$$s\left(\left(a_1, a_2\right), \left(y_1, y_2\right)\right) = \left(a_1 - y_1\right)^2 + \left(a_2 - y_2\right)^2,$$

kde a_1, a_2 jsou postupně aktivity prvního a druhého neuronu v poslední vrstvě sítě, a kde y_1, y_2 jsou postupně očekávané odpovědi prvního a druhého neuronu v poslední vrstvě.

△

I když pro neuronové sítě existují i jiné (pro určité situace vhodnější) ztrátové funkce vedle součtu čtverců chyb, budeme pro jednoduchost v celé práci pracovat s touto ztrátovou funkcí a jejími variantami.

V předchozích dvou příkladech jsme uvedli ztrátové funkce, které přiřazují ztrátu jednomu konkrétnímu vstupu/výstupu. V praxi se spíše uvažuje ztrátová funkce, která přiřadí ztrátu celému tréninkovému datasetu, tedy jisté kolekci vstupů/výstupů. Tento tréninkový dataset můžeme chápat jako sbírku dvojic složených ze vstupů a příslušných očekávaných výstupů, kterou neuronové síti předáváme jako „studijní materiál“. Přitom tyto datasety musí být obsáhlé, neboť neuronové sítě obvykle používáme se záměrem řešit nějaký problém s nejasným vzorem, který není snadné řešit algoritmicky. Těžko ale umožníme neuronové síti tento vzor odhalit na malém datasetu. Pro jednoduchost budeme v této kapitole uvažovat ztrátovou funkci pro jeden vstup/výstup, tedy budeme uvažovat trénink neuronové sítě na jednom vstupu/výstupu, a popíšeme myšlenku, která stojí za strojovým učením.

Jak jsme viděli na příkladech s neurony, najdeme-li správné váhy a prahy, neurony můžeme použít pro řešení různých úloh, přičemž neuronové sítě pak slouží ještě daleko lépe. Klíčovou otázkou tedy zůstává: „Jak najít ty správné váhy a prahy?“ Uvažujme jeden vstup pro neuronovou síť a jeden očekávaný výstup, příslušný tomuto vstupu. Uvažujme dále hodnoty ztrátové funkce pro různé nastavení parametrů sítě. Pro „lepší“ sadu parametrů můžeme pozorovat menší hodnoty ztrátové funkce, než pozorujeme pro „horší“ sady parametrů. Uvědomme si, že v tuto chvíli, kdy máme pevně zvolený vstup, resp. pevně zvolený tréninkový vzorek, můžeme ztrátovou funkci vnímat jako funkci, která přiřadí parametrům sítě danou ztrátu. Tedy budeme-li minimalizovat funkci přiřazující parametrům sítě ztrátu na daném tréninkovém vzorku, budeme trénovat síť!

Teď by se nám hodilo zapojit do hry prostředky matematické analýzy, zmíněnou funkci zderivovat, najít stacionární body a najít globální minimum. Předpokládejme, že takové globální minimum existuje. V tu chvíli bychom našli opravdu tu nejlepší sadu parametrů, na které by měla síť tu nejmenší možnou ztrátu, a trénink by byl u konce. Bohužel takto jednoduše to nejde. Jednoduchá neuronová síť z Obrázku 1.3 má celkem 32 parametrů. Tedy ztrátová funkce, kterou nyní vnímáme jako funkci, která přiřazuje sadě parametrů ztrátu na pevném tréninkovém vzorku, je nyní závislá na 32 parametrech. To by nebyl problém, kdyby řešení úlohy, kdy jsou všechny parciální derivace ztrátové funkce nulové, nevedlo na soustavy nelineárních rovnic. Navíc pro běžně používané neuronové sítě s miliony parametry bychom museli řešit systém miliónů takovýchto rovnic. Nakonec bychom stejně nedostali hledané minimum, ale pouze stacionární body, ze kterých bychom ještě museli vybrat ten nejlepší.

V praxi se tento problém řeší numerickými optimalizačními metodami, nejčastěji variantami metody největšího spádu. Později v této práci (viz Kapitola 2.2) ukážeme, jak lze pomoci tzv. *algoritmu zpětného šíření chyby* (angl. *backpropagation*) efektivně počítat gradient ztrátové funkce. K optimalizaci pak využíváme faktu, že ztrátová funkce v daném bodě klesá nejvíce v opačném směru vzhledem ke směru gradientu. Intuici k fungování optimalizačních metod se ale budeme věnovat později.

2. Neuronové sítě formálně

Nyní si pojem neuronové sítě zavedeme rigorózně matematicky. Pokud neřekneme jinak, měly by zde uvedené definice odpovídat pojmům z minulé kapitoly. Doplňujícím zdrojem je např. [4].

2.1 Zavedení základních pojmů

Zde budeme postupně od základu definovat všechny pojmy, které budeme potřebovat k definici neuronové sítě. Začněme s již dříve zmíněnou funkcí σ, totiž se sigmoidou.

Definice 1 (funkce σ)**.** *Definujeme funkci σ následujícím předpisem:*

$$\sigma(z) = \frac{1}{1+e^{-z}}, \quad z \in \mathbb{R}.$$

Poznámka. Graf funkce σ jsme viděli již na Obrázku 1.2. Snadno se můžeme přesvědčit o následujících vlastnostech funkce σ.

- $\mathrm{Rng}\,(\sigma) = (0,1)$,
- $\lim\limits_{z\to\infty} \sigma\,(z) = 1$,
- $\lim\limits_{z\to-\infty} \sigma\,(z) = 0$,
- $\sigma\,(0) = 1/2$,
- $\sigma \in \mathcal{C}^\infty\,(\mathbb{R})$.

Později si neuronovou síť zadefinujeme jako složení jistých vektorových zobrazení. Často budeme aplikovat funkci σ nebo její derivaci σ' na nějaký vektor. Proto se nám také bude hodit, pro jednoduchost zápisu, následující značení.

Značení 1. *Nechť $n \in \mathbb{N}$, $a \in \mathbb{R}^n, a = (a_1, a_2, \ldots, a_n)^T$ je reálný aritmetický vektor. Zápisem $\sigma(a)$, resp. $\sigma'(a)$, budeme rozumět obraz vektoru a při zobrazení σ, resp. při zobrazení σ', tzv. po složkách, tj.*

$$\sigma(a) = (\sigma(a_1), \sigma(a_2), \ldots, \sigma(a_n))^T,$$

resp.

$$\sigma'(a) = (\sigma'(a_1), \sigma'(a_2), \ldots, \sigma'(a_n))^T.$$

V minulé kapitole jsme si již trochu představili způsob, jak jsou na sebe jednotlivé neurony napojené. Síť se skládá z vrstev neuronů, přičemž počet vrstev sítě ani počet neuronů v každé vrstvě není striktně dán. Strukturu (neboli architekturu) sítě si můžeme vybrat tak, jak pro daný problém zrovna potřebujeme. Strukturou sítě rozumíme právě to, kolik má síť vrstev, a kolik neuronů najdeme v jednotlivých vrstvách. Následuje formální definice struktury neuronové sítě.

Definice 2 (Struktura neuronové sítě)**.** *Nechť $k \in \mathbb{N}$, $n_0, n_1, \ldots, n_k \in \mathbb{N}$.* Strukturou neuronové sítě *budeme rozumět uspořádanou $(k+1)$-tici $(n_0, n_1, \ldots, n_k)$. Číslo $k+1$ nazýváme* hloubkou neuronové sítě*, číslo n_0 nazýváme* dimenze vstupu*, číslo n_k nazýváme* dimenze výstupu *a číslo n_i nazýváme* dimenze i-té (skryté) vrstvy*, kde $i \in \{1, 2, \ldots, k-1\}$.*

Strukturu neuronové sítě definujeme ještě před neuronovou sítí samotnou, abychom měli předem dané dimenze jednotlivých prostorů, na kterých budeme definovat funkce odpovídající neuronům, popřípadě zobrazení odpovídající jednotlivým vrstvám sítě.

Poznámka. Je-li $(n_0, n_1, \ldots, n_k)$ struktura neuronové sítě, pak hloubka neuronové sítě $k+1$ odpovídá faktu, že síť se skládá z $k+1$ vrstev. Čísla n_i potom udávají, kolik neuronů se nachází v i-té vrstvě sítě, $i \in \{0, 1, \ldots k\}$.

Příklad. Síť z Obrázku 1.3 má strukturu $(3, 4, 2, 2)$. Má celkem 4 vrstvy, v první vrstvě nalezneme 3 neurony, v druhé vrstvě 4 neurony, ve třetí vrstvě 2 neurony a nakonec ve čtvrté vrstvě se nachází také 2 neurony. Připomeňme, že první vrstva není ve skutečnosti vrstva neuronů, ale znázorňuje velikost vstupu sítě.

△

Nyní si zavedeme značení pro prostor parametrů jednotlivých neuronů. Prostor Θ_i^l bude v následující definici značit množinu všech vah a prahů i-tého neuronu l-té vrstvy. Celkem je těchto parametrů $n_{l-1} + 1$ (jedna váha pro každý neuron z předešlé vrstvy a jeden práh), což bude i dimenze tohoto prostoru. Analogicky označíme i prostory všech parametrů jednotlivých vrstev, a nakonec prostor všech parametrů sítě.

Definice 3 (Parametrický prostor sítě)**.** *Nechť $(n_0, n_1, \ldots, n_k)$ je struktura neuronové sítě. Pro každé $l \in \{1, 2, \ldots, k\}$ a $i \in \{1, 2, \ldots, n_l\}$ definujeme* parametrický prostor Θ_i^l i-tého neuronu l-té vrstvy *jakožto množinu parametrů:*

$$\Theta_i^l = \mathbb{R}^{n_{l-1}} \times \mathbb{R}.$$

Dále pro každé $l \in \{1, 2, \ldots, k\}$ položme:

$$\Theta^l = \left(\Theta_i^l\right)^{n_l} = (\mathbb{R}^{n_{l-1}} \times \mathbb{R})^{n_l} = \mathbb{R}^{n_l \times n_{l-1}} \times \mathbb{R}^{n_l}.$$

Parametrický prostor Θ neuronové sítě *pak definujeme jako množinu:*

$$\Theta = \Theta^1 \times \Theta^2 \times \cdots \times \Theta^k.$$

Množinu Θ můžeme chápat jako kolekci všech možných kombinací parametrů sítě. Tedy každý vektor parametrů $\theta \in \Theta$ odpovídá jednomu nastavení všech vah a všech prahů všem neuronům v síti.

Následuje definice neuronu. Zde je potřeba upozornit na odlišnost vzhledem k předchozí kapitole. V Kapitole 1 si neuron pamatoval své parametry a na základě těchto parametrů přiřazoval podnětům aktivity. Od nynějška si neuron již žádné parametry pamatovat nebude; váhy a prahy bude neuron přijímat na vstupu stejně jako podněty.

Definice 4 (Neuron)**.** *Nechť* $(n_0, n_1, \ldots, n_k)$ *je struktura neuronové sítě,* Θ *příslušný parametrický prostor. Nechť dále* $l \in \{1, 2, \ldots, k\}$*,* $i \in \{1, 2, \ldots, n_l\}$*, pak* převáženými vstupy i-tého neuronu l-té vrstvy *rozumíme zobrazení:*

$$z_i^l : \Theta_i^l \times [0,1]^{n_{l-1}} \to \mathbb{R},$$
$$z_i^l \left(w_i^l, b_i^l, a^{l-1}\right) = \sum_{j=1}^{n_{l-1}} w_{i,j}^l a_j^{l-1} + b_i^l,$$

kde

$$(w_i^l, b_i^l) = \left(\left(w_{i,1}^l, w_{i,2}^l, \ldots, w_{i,n_{l-1}}^l\right), b_i^l\right) \in \Theta_i^l,$$
$$a^{l-1} = \left(a_1^{l-1}, a_2^{l-1}, \ldots a_{n_{l-1}}^{l-1}\right) \in [0,1]^{n_{l-1}}.$$

Zde a^{l-1} *je vektor vstupů (výstup předchozí vrstvy). Dále* i-tým neuronem l-té vrstvy f_i^l *rozumíme zobrazení:*

$$f_i^l : \Theta_i^l \times [0,1]^{n_{l-1}} \to (0,1),$$
$$f_i^l = \sigma \circ z_i^l.$$

Poznámka. Díky faktu, že $\mathrm{Rng}\,(\sigma) = (0,1)$, je předchozí definice korektní.

Poznámka. Číslům $w_{i,j}^l$, $l \in \{1, 2, \ldots, k\}$, $i \in \{1, 2, \ldots, n_l\}$, $j \in \{1, 2, \ldots, n_{l-1}\}$ z předchozí definice říkáme *váhy* (někdy *synaptické váhy*, v anglické literatuře pak *weights*), číslům b_i^l, $l \in \{1, 2, \ldots, k\}$, $i \in \{1, 2, \ldots, n_l\}$ z předchozí říkáme *prahy* (v anglické literatuře *biases*). Souhrnně pak těmto číslům říkáme *parametry* (angl. *parameters*).

I přesto, že v předchozí Kapitole 1 jsme neuron chápali jako jednotku s pevnými parametry, bude se nám hodit definovat neuron tak, jak jsme uvedli v Definici 4. Při „trénování“ neuronové sítě budeme hledat „nejvhodnější“ sadu parametrů (co v tomto smyslu znamená pojem „nejvhodnější“ je vysvětleno na začátku Kapitoly 2.2, napovědět může také Definice 9), tedy budeme pracovat pořád se stejnou sítí, které budeme měnit parametry; kdybychom definovali neurony tak, jako v Kapitole 1, narazili bychom na jisté technické problémy.

Ještě před definicí vrstvy neuronů si zadefinujeme matici vah l-té vrstvy. Vrstvu neuronů totiž nebudeme definovat jako vektor neuronů. Použijeme ekvivalentní definici právě pomocí matice vah l-té vrstvy.

Definice 5. *Nechť* $(n_0, n_1, \ldots, n_k)$ *je struktura neuronové sítě,* Θ *příslušný parametrický prostor, nechť* $l \in \{1, 2, \ldots, k\}$*. Mějme pro každé* $i \in \{1, 2, \ldots, n_l\}$

$$(w_i^l, b_i^l) = \left(\left(w_{i,1}^l, w_{i,2}^l, \ldots, w_{i,n_{l-1}}^l\right), b_i^l\right) \in \Theta_i^l,$$

jako v minulé definici. Definujeme matici vah l-té vrstvy *jako matici* W^l*, jejíž* i*-tý řádek je vektor* w_i^l*,* $i \in \{1, 2, \ldots, n_l\}$*, tj. matici* $W^l = (w_{i,j}^l)_{i,j=1}^{n_l, n_{l-1}}$*. Dále definujeme* prahový vektor l-té vrstvy *jako vektor* $b^l = \left(b_1^l, b_2^l, \ldots, b_{n_l}^l\right)^T$*.*

Poznámka. Všimněme si, že $\left(W^l, b^l\right) \in \Theta^l$.

Následuje definice vrstvy neuronů. Podle poznámky za definicí je definice ekvivalentní s intuicí z minulé kapitoly.

Definice 6 (Vrstva neuronů)**.** *Nechť* $(n_0, n_1, \ldots, n_k)$ *je struktura neuronové sítě,* Θ *příslušný parametrický prostor. Pro* $l \in \{1, 2, \ldots, k\}$ *definujeme* převážené vstupy l-té vrstvy *jako zobrazení:*

$$z^l : \Theta^l \times [0,1]^{n_{l-1}} \to \mathbb{R}^{n_l},$$
$$z^l\left(W^l, b^l, a^{l-1}\right) = W^l a^{l-1} + b^l,$$

kde

$$\left(W^l, b^l\right) \in \Theta^l, \quad a^{l-1} \in [0,1]^{n_{l-1}}.$$

Dále l-tou vrstvou neuronů *rozumíme zobrazení:*

$$F^l : \Theta^l \times [0,1]^{n_{l-1}} \to (0,1)^{n_l},$$
$$F^l = \sigma \circ z^l.$$

Poznámka. Díky faktu, že $\mathrm{Rng}\,(\sigma) = (0,1)$ a σ chápeme jako zobrazení po složkách, je předchozí definice korektní.

Poznámka. Nechť $(n_0, n_1, \ldots, n_k)$ je struktura neuronové sítě, Θ příslušný parametrický prostor. Můžeme si rovněž všimnout, že pro $l \in \{1, 2, \ldots, k\}$ platí:

$$z^l(W^l, b^l, a^{l-1}) = \begin{pmatrix} z_1^l(w_1^l, b_1^l, a^{l-1}) \\ z_2^l(w_2^l, b_2^l, a^{l-1}) \\ \vdots \\ z_{n_l}^l(w_{n_l}^l, b_{n_l}^l, a^{l-1}) \end{pmatrix},$$

kde $a^{l-1} \in [0,1]^{n_{l-1}}$, a kde z_i^l jsou převážené vstupy i-tého neuronu l-té vrstvy, $i \in \{1, 2, \ldots, n_l\}$, $w_1^l, w_2^l, \ldots, w_{n_l}^l$ jsou řádky matice W^l, a $b_1^l, b_2^l, \ldots, b_{n_l}^l$ jsou složky vektoru b^l, kde $\left(W^l, b^l\right) \in \Theta^l$. Díky tomu také platí:

$$F^l = \begin{pmatrix} f_1^l \\ f_2^l \\ \vdots \\ f_{n_l}^l \end{pmatrix}.$$

Neuronovou síť už nyní můžeme zadefinovat jednoduše jako složení dříve uvedených zobrazení.

Definice 7 (Neuronová síť)**.** *Nechť* $(n_0, n_1, \ldots, n_k)$ *je struktura neuronové sítě,* Θ *příslušný parametrický prostor.* Neuronovou sítí *budeme rozumět zobrazení*

$$\mathbb{F} : \Theta \times [0,1]^{n_0} \to (0,1)^{n_k},$$

pro každé $\theta = (\theta^1, \theta^2, \ldots, \theta^k) \in \Theta$ *definované předpisem:*

$$\mathbb{F}(\theta, x) = F^k(\theta^k, a^{k-1}),$$

kde

$$a^l = F^l(\theta^l, a^{l-1}), \quad l \in \{1, 2, \ldots, k-1\},$$
$$a^0 = x \in [0,1]^{n_0},$$

kde F^l *jsou* l*-té vrstvy neuronů,* $l \in \{1, 2, \ldots, k\}$.

Nyní si již zadefinujeme objekty potřebné k trénování neuronové sítě. Definice požaduje, aby dimenze tréninkových vzorků a očekávaných odpovědí byly stejné jako dimenze vstupu a výstupu sítě. Podotkněme, že v praxi bude tato závislost opačná - dimenze sítě budeme volit na základě dimenzí, které se vyskytují v úloze, kterou řešíme - tedy velikost/struktura neuronové sítě bude záviset na tom, jaký máme k dispozici tréninkový dataset. Nicméně formálně se nám hodí chápat tréninkový dataset jako v následující definici.

Definice 8 (Tréninkový dataset)**.** *Nechť* $(n_0, n_1, \ldots, n_k)$ *je struktura neuronové sítě. Nechť* $n_0, n_k, N \in \mathbb{N}$*,* tréninkovým datasetem velikosti N *rozumíme posloupnost uspořádaných dvojic* $\{(x_n, y_n)\}_{n=1}^{N}$*, kde pro každé* $n \in \{1, 2, \ldots, N\}$ *vektor* $x_n \in [0,1]^{n_0}$ *nazýváme* tréninkovým vstupem*,* $y_n \in \{0,1\}^{n_k}$ *nazýváme* očekávaným výstupem*. Dvojici* (x_n, y_n) *nazýváme* tréninkovým vzorkem*,* $n \in \{1, 2, \ldots, N\}$*.*

Dále zadefinujeme ztrátovou funkci. Nutno podotknout, že označení „ztrátová funkce“ je trochu nepřesné, neboť obvykle pojmem ztrátová funkce označujeme širší třídu funkcí. V této práci však pro jednoduchost s jinou ztrátovou funkcí pracovat nebudeme, a tak pojem ztrátová funkce budeme používat pro konkrétní funkci.

Definice 9 (Ztrátová funkce)**.** *Nechť* $(n_0, n_1, \ldots, n_k)$ *je struktura neuronové sítě,* Θ *příslušný parametrický prostor, nechť* $\mathbb{F}$ *je příslušná neuronová síť a nakonec nechť* $D = \{(x_n, y_n)\}_{n=1}^{N}$ *je tréninkový dataset velikosti* $N \in \mathbb{N}$*. Definujeme* ztrátovou funkci neuronové sítě $\mathbb{F}$ na datasetu D *jako funkci:*

$$L : \Theta \times \left([0,1]^{n_0} \times \{0,1\}^{n_k} \right)^N \to [0, \infty),$$
$$L(\theta, D) = \frac{1}{N} \sum_{n=1}^{N} s(\mathbb{F}(\theta, x_n), y_n),$$

kde

$$s : (0,1)^{n_k} \times \{0,1\}^{n_k} \to [0, \infty),$$
$$s(a, y) = \frac{1}{2}\|a - y\|^2 = \frac{1}{2} \sum_{i=1}^{n_k} (a_i - y_i)^2.$$

Poznámka. Parametrický prostor sítě bývá zpravidla prostor s dimenzí (mnohem) větší než dva; ztrátovou funkci, jakožto funkci parametrů sítě, si tak nelze jednoduše představit. Nicméně v Kapitole 4.4 uvádíme alespoň řez grafem ztrátové funkce (konkrétně viz Obrázek 4.8).

Nyní již máme všechny potřebné objekty formálně zadefinované. Následuje kapitola, ve které si ukážeme jisté vlastnosti ztrátové funkce.

2.2 Odvození algoritmu zpětného šíření chyby

Uvažujme nyní ztrátovou funkci $L(\theta, D)$. Je nutné si uvědomit, že dataset D zůstane v průběhu optimalizace pevný a nebude se měnit. Naopak, snažíme se najít nejlepší možnou sadu parametů $\theta \in \Theta$, která minimalizuje hodnotu $L(\theta, D)$.

Dataset D tedy chápeme pouze jako parametr ztrátové funkce L, přičemž L je pak pouze funkce proměnné θ. Analogicky i funkci $s\left(\mathbb{F}\left(\theta, x_n\right), y_n\right)$ má smysl vnímat jen jako funkci proměnné θ.

Jak jsme již dříve napověděli, bude nás zajímat gradient funkce L vzhledem k parametrům θ (tedy vektor parciálních derivací funkce L vzhledem ke všem vahám a prahům sítě).

Značení 2. *Zápisem $\nabla L\left(\theta, D\right)$ budeme rozumět vektor parciálních derivací ztrátové funkce L vzhledem ke všem vahám a prahům sítě. Tedy $\nabla L\left(\theta, D\right)$ značí gradient funkce L vzhledem k proměnné θ.*

Díky linearitě derivace si můžeme výpočet gradientu usnadnit, stačí uvažovat jeden tréninkový vzorek $(x, y) \in [0,1]^{n_0} \times \{0,1\}^{n_k}$. Navíc pro výpočet požadovaných parciálních derivací ztrátové funkce L vzhledem k vahám a prahům nám stačí vypočítat odpovídající parciální derivace funkce s. Ještě než se pustíme do samotného výpočtu gradientu, zavedeme si pár užitečných značení a promyslíme si jisté technické okolnosti.

Značení 3. *Symbolem a^k označme výstup neuronové sítě pro tréninkový vstup x; tedy $a^k = \mathbb{F}\left(\theta, x\right)$. Toto značení je v souladu se značením aktivit z dřívějších vrstev, které jsme uvedli v Definici 7.*

Následuje úvaha, díky které bude snadnější odvodit výpočet gradientu a díky které bude následně i kód algoritmu šíření zpětné chyby o něco přehlednější.

Podle dřívějších definic a podle poznámky za definicí vrstvy neuronů platí:

$$a^k = \mathbb{F}\left(\theta, x\right) = \begin{pmatrix} f_1^k\left(\theta_1^k, a^{k-1}\right) \\ f_2^k\left(\theta_2^k, a^{k-1}\right) \\ \vdots \\ f_{n_k}^k\left(\theta_{n_k}^k, a^{k-1}\right) \end{pmatrix} = \begin{pmatrix} \sigma\left(z_1^k\left(w_1^k, b_1^k, a^{k-1}\right)\right) \\ \sigma\left(z_2^k\left(w_2^k, b_2^k, a^{k-1}\right)\right) \\ \vdots \\ \sigma\left(z_{n_k}^k\left(w_{n_k}^k, b_{n_k}^k, a^{k-1}\right)\right) \end{pmatrix}.$$

Jak je vidět, a^k přímo závisí na převážených vstupech k-té vrstvy. Když bychom ale na chvíli zapomněli, odkud se čísla $z_1^k, z_2^k, \ldots, z_{n_k}^k$ vzala, mohli bychom vnímat funkci a^k jednoduše jako funkci proměnných $z_1^k, z_2^k, \ldots, z_{n_k}^k$. Formálně by nám tento obrat umožnil derivovat a_k podle z_i^k, $i \in \{1, 2, \ldots, n_k\}$. Ukažme si to na příkladu.

Příklad. Pokud bychom nyní chtěli spočítat např. $\partial a_1^k / \partial w_{1,1}^k$, můžeme a_1^k jakožto funkci vah a prahů sítě vnímat i jako složenou funkci následujících dvou funkcí: za prvé - funkce a_1^k závislé na jedné vektorové proměnné z^k - a za druhé - funkce z^k závislé na vahách a prazích sítě. Potom:

$$\frac{\partial a_1^k}{\partial w_{1,1}^k} = \frac{\partial a_1^k}{\partial z_1^k} \frac{\partial z_1^k}{\partial w_{1,1}^k} = \sigma'\left(z_1^k\right) a_1^{k-1}.$$

Díky tomu, že z^k chápeme jako zobrazení a zároveň jako proměnnou, dává prostřední výraz smysl.

△

Analogicky a^k (a také jiná zobrazení v síti) závisí i na převážených vstupech z dřívějších vrstev. Budeme tedy také zaměňovat převážené vstupy $z^1, z^2, \ldots, z^{k-1}$ s proměnnými funkcí, do kterých je dosazujeme.

V literatuře, která pojednává o neuronových sítích je také zvykem pro parciální derivace ztrátové funkce vzhledem k proměnným $z^1, z^2, \ldots, z^k$ používat následující značení. Používá se nejen při odvození výpočtu gradientu, ale také jako proměnná přímo v kódu algoritmu zpětného šíření chyby - jedná se o hodnotu, která se ve výpočtu použije na více místech.

Značení 4. *Nechť $s, z^1, z^2, \ldots, z^k$ jsou jako výše. Pro každé $l \in \{1, 2, \ldots, k\}$ označme:*

$$\delta^l(a, y) = \begin{pmatrix} \delta_1^l(a, y) \\ \delta_2^l(a, y) \\ \vdots \\ \delta_{n_l}^l(a, y) \end{pmatrix},$$

kde

$$\delta_i^l(a, y) = \frac{\partial s}{\partial z_i^l}(a, y), \quad i \in \{1, 2, \ldots, n_l\}.$$

Pro jednoduchost zápisu nám opět poslouží následující značení. Řekneme si, co je tzv. Hadamardův součin; jedná se o násobení vektorů po složkách.

Definice 10. *Nechť $a, b \in \mathbb{R}^n, n \in \mathbb{N}, a = (a_1, a_2, \ldots, a_n)^T, b = (b_1, b_2, \ldots, b_n)^T$. Definujeme* Hadamardův součin vektorů a a b *následujícím předpisem:*

$$a \odot b = (a_1 b_1, a_2 b_2, \ldots, a_n b_n)^T.$$

Následující lemma hovoří o tom, jak vypočítat hodnoty $\delta^l\left(a^k, y\right)$, pro všechna $l \in \{1, 2, \ldots, k\}$. Všimněme si, že díky zavedeným značením pro důkaz nepotřebujeme o moc víc než řetízkové pravidlo. Podrobnější intuici o tom, co lemma říká a jakými různými způsoby lze uměle zavedené veličiny δ^l chápat, lze najít v [2]. Důkaz lze také najít v [4] (Lemma 1).

Lemma 1. *Pro všechna $l \in \{1, 2, \ldots, k-1\}$ platí:*

$$\delta^l\left(a^k, y\right) = \sigma'\left(z^l\right) \odot \left(W^{l+1}\right)^T \delta^{l+1}\left(a^k, y\right). \tag{2.1}$$

Navíc všechny tyto hodnoty lze vypočítat iteračně, neboť platí:

$$\delta^k\left(a^k, y\right) = \sigma'\left(z^k\right) \odot \left(a^k - y\right). \tag{2.2}$$

Důkaz. Nejprve dokážeme rovnost (2.2). Nechť $j \in \{1, 2, \ldots, n_k\}$ je dáno libovolné. Platí:

$$\begin{aligned} \delta_j^k\left(a^k, y\right) &= \frac{\partial s}{\partial z_j^k}\left(a^k, y\right) = \frac{\partial}{\partial z_j^k}\left(\frac{1}{2}||a^k - y||^2\right) = \frac{\partial}{\partial z_j^k}\left(\frac{1}{2}\sum_{i=1}^{n_k}\left(a_i^k - y_i\right)^2\right) = \\ &= \frac{\partial}{\partial z_j^k}\left(\frac{1}{2}\sum_{i=1}^{n_k}\left(\sigma\left(z_i^k\right) - y_i\right)^2\right) = \frac{1}{2}\cdot 2\left(\sigma\left(z_j^k\right) - y_j\right)^1 \sigma'\left(z_j^k\right) = \\ &= \sigma'\left(z_j^k\right)\left(a_j^k - y_j\right). \end{aligned}$$

Protože j bylo voleno libovolně, rovnost platí pro všechna $j \in \{1, 2, \ldots, n_k\}$. Celkem tedy:

$$\delta^k\left(a^k, y\right) = \begin{pmatrix} \delta_1^k\left(a^k, y\right) \\ \delta_2^k\left(a^k, y\right) \\ \vdots \\ \delta_{n_k}^k\left(a^k, y\right) \end{pmatrix} = \begin{pmatrix} \sigma'\left(z_1^k\right)\left(a_1^k - y_1\right) \\ \sigma'\left(z_2^k\right)\left(a_2^k - y_2\right) \\ \vdots \\ \sigma'\left(z_{n_k}^k\right)\left(a_{n_k}^k - y_{n_k}\right) \end{pmatrix} =$$

$$= \begin{pmatrix} \sigma'\left(z_1^k\right) \\ \sigma'\left(z_2^k\right) \\ \vdots \\ \sigma'\left(z_{n_k}^k\right) \end{pmatrix} \odot \begin{pmatrix} \left(a_1^k - y_1\right) \\ \left(a_2^k - y_2\right) \\ \vdots \\ \left(a_{n_k}^k - y_{n_k}\right) \end{pmatrix} = \sigma'\left(z^k\right) \odot \left(a^k - y\right),$$

což jsme chtěli.

Nyní mějme $l \in \{1, 2, \ldots, k-1\}$, dokážeme, že platí rovnost (2.1). Nechť opět $j \in \{1, 2, \ldots, n_l\}$ je dáno libovolné. Počítejme:

$$\delta_j^l\left(a^k, y\right) = \frac{\partial s}{\partial z_j^l}\left(a^k, y\right) = \sum_{i=1}^{n_{l+1}} \frac{\partial s}{\partial z_i^{l+1}}\left(a^k, y\right) \frac{\partial z_i^{l+1}}{\partial z_j^l}\left(w_i^{l+1}, b_i^{l+1}, a^l\right) =$$

$$= \sum_{i=1}^{n_{l+1}} \delta_i^{l+1}\left(a^k, y\right) \frac{\partial z_i^{l+1}}{\partial z_j^l}\left(w_i^{l+1}, b_i^{l+1}, a^l\right) \tag{2.3}$$

Dále pro $i \in \{1, 2, \ldots, n_{l+1}\}$ platí:

$$\frac{\partial z_i^{l+1}}{\partial z_j^l}\left(w_i^{l+1}, b_i^{l+1}, a^l\right) = \frac{\partial}{\partial z_j^l}\left(\sum_{\gamma=1}^{n_l} w_{i,\gamma}^{l+1} a_\gamma^l + b_i^{l+1}\right) =$$

$$= \frac{\partial}{\partial z_j^l}\left(\sum_{\gamma=1}^{n_l} w_{i,\gamma}^{l+1} \sigma\left(z_\gamma^l\right) + b_i^{l+1}\right) = w_{i,j}^{l+1} \sigma'\left(z_j^l\right).$$

Pokračujme ve výpočtu (2.3):

$$\sum_{i=1}^{n_{l+1}} \delta_i^{l+1}\left(a^k, y\right) w_{i,j}^{l+1} \sigma'\left(z_j^l\right) = \sigma'\left(z_j^l\right)\left(\left(W^{l+1}\right)^T \delta^{l+1}\left(a^k, y\right)\right)_j.$$

Protože j bylo voleno libovolně, rovnost platí pro všechna $j \in \{1, 2, \ldots, n_l\}$.

Celkem tedy:

$$\delta^l\left(a^k,y\right)=\begin{pmatrix}\delta_1^l\left(a^k,y\right)\\ \delta_2^l\left(a^k,y\right)\\ \vdots\\ \delta_{n_l}^l\left(a^k,y\right)\end{pmatrix}=\begin{pmatrix}\sigma'\left(z_1^l\right)\left(\left(W^{l+1}\right)^T\delta^{l+1}\left(a^k,y\right)\right)_1\\ \sigma'\left(z_2^l\right)\left(\left(W^{l+1}\right)^T\delta^{l+1}\left(a^k,y\right)\right)_2\\ \vdots\\ \sigma'\left(z_{n_l}^l\right)\left(\left(W^{l+1}\right)^T\delta^{l+1}\left(a^k,y\right)\right)_{n_l}\end{pmatrix}=$$

$$=\begin{pmatrix}\sigma'\left(z_1^l\right)\\ \sigma'\left(z_2^l\right)\\ \vdots\\ \sigma'\left(z_{n_l}^l\right)\end{pmatrix}\odot\begin{pmatrix}\left(\left(W^{l+1}\right)^T\delta^{l+1}\left(a^k,y\right)\right)_1\\ \left(\left(W^{l+1}\right)^T\delta^{l+1}\left(a^k,y\right)\right)_2\\ \vdots\\ \left(\left(W^{l+1}\right)^T\delta^{l+1}\left(a^k,y\right)\right)_{n_l}\end{pmatrix}=$$

$$=\sigma'\left(z^l\right)\odot\left(W^{l+1}\right)^T\delta^{l+1}\left(a^k,y\right),$$

což jsme chtěli.

Při přímém chodu vstupního vektoru $a_0\in[0,1]^{n_0}$ neuronovou sítí si zapamatujeme všechny evaluace převážených vstupů l-té vrstvy z^l pro všechna $l\in\{1,2,\ldots,k\}$. Pak díky znalosti z^k a a^k můžeme podle vztahu (2.2) dopočítat $\delta^k\left(a^k,y\right)$. Díky znalosti $\delta^k\left(a^k,y\right)$ a z^{k-1} dále můžeme podle vztahu (2.1) dopočítat $\delta^{k-1}\left(a^k,y\right)$. Obecně, známe-li $\delta^{l+1}\left(a^k,y\right)$ a z^l pro $l\in\{1,2,\ldots,k-2\}$, můžeme dopočítat hodnotu $\delta^l\left(a^k,y\right)$. Sestupným chodem tedy můžeme spočítat všechny požadované hodnoty.

□

V dalším tvrzení pohovoříme o tom, jak využít znalost hodnot $\delta^l\left(a^k,y\right)$ k výpočtu parciálních derivací ztrátové funkce sítě vzhledem k vahám a prahům sítě. Opět odkážeme na knihu [2], ve které najdeme intuitivnější vysvětlení následujících výpočtů. Rovněž důkaz tohoto tvrzení lze najít v [4]. Všimněme si, jak jednoduché vztahy následující tvrzení dává, totiž jak zavedení značení δ^l zpřehledňuje situaci. Opět využijeme převážně řetízkové pravidlo.

Tvrzení 2. *Pro všechna* $l\in\{1,2,\ldots,k\},i\in\{1,2,\ldots,n_l\},j\in\{1,2,\ldots,n_{l-1}\}$ *platí:*

$$\frac{\partial s}{\partial b_i^l}\left(a^k,y\right)=\delta_i^l\left(a^k,y\right), \tag{2.4}$$

$$\frac{\partial s}{\partial w_{i,j}^l}\left(a^k,y\right)=\delta_i^l\left(a^k,y\right)a_j^{l-1}. \tag{2.5}$$

Důkaz. Nechť $l\in\{1,2,\ldots,k\},i\in\{1,2,\ldots,n_l\}$ dány libovolné, začneme důka-

zem rovnosti (2.4). Platí:

$$\begin{aligned}\frac{\partial s}{\partial b_i^l}\left(a^k,y\right) &= \frac{\partial s}{\partial z_i^l}\left(a^k,y\right)\frac{\partial z_i^l}{\partial b_i^l}\left(w_i^l,b_i^l,a^{l-1}\right) = \delta_i^l\left(a^k,y\right)\frac{\partial z_i^l}{\partial b_i^l}\left(w_i^l,b_i^l,a^{l-1}\right) = \\ &= \delta_i^l\left(a^k,y\right)\frac{\partial}{\partial b_i^l}\left(w_{i,1}^l a_1^{l-1} + w_{i,2}^l a_2^{l-1} + \cdots + w_{i,n_{l-1}}^l a_{n_{l-1}}^{l-1} + b_i^l\right) = \\ &= \delta_i^l\left(a^k,y\right)\cdot 1 = \delta_i^l\left(a^k,y\right).\end{aligned}$$

Nechť navíc $j \in \{1,2,\ldots,n_{l-1}\}$ je dáno libovolné, dokážeme rovnost (2.5):

$$\begin{aligned}\frac{\partial s}{\partial w_{i,j}^l}\left(a^k,y\right) &= \frac{\partial s}{\partial z_i^l}\left(a^k,y\right)\frac{\partial z_i^l}{\partial w_{i,j}^l}\left(w_i^l,b_i^l,a^{l-1}\right) = \delta_i^l\left(a^k,y\right)\frac{\partial z_i^l}{\partial w_{i,j}^l}\left(w_i^l,b_i^l,a^{l-1}\right) = \\ &= \delta_i^l\left(a^k,y\right)\frac{\partial}{\partial w_{i,j}^l}\left(w_{i,1}^l a_1^{l-1} + w_{i,2}^l a_2^{l-1} + \cdots + w_{i,n_{l-1}}^l a_{n_{l-1}}^{l-1} + b_i^l\right) = \\ &= \delta_i^l\left(a^k,y\right)a_j^{l-1}.\end{aligned}$$

Protože l,i,j byly voleny libovolně, dostáváme platnost rovností pro všechna $l \in \{1,2,\ldots,k\}, i \in \{1,2,\ldots,n_l\}, j \in \{1,2,\ldots,n_{l-1}\}$, což jsme chtěli.

□

Odvodili jsme vzorečky pro výpočet parciálních derivací ztrátové funkce neuronové sítě podle všech jejich parametrů; máme tedy vše potřebné pro výpočet $\nabla L(\theta, D)$. Při programování neuronových sítí se dnes běžně používají knihovny, které implementují základní lineární algebru (zpravidla obsahují datové struktury reprezentující matice a vektory, a také implementují funkce pro násobení matic, násobení skaláru s vektorem, Hadamardův součin atd.). Použití těchto knihoven výrazným způsobem zpřehledňuje kód. Níže si ukážeme pseudokód algoritmu zpětného šíření chyby, který předpokládá možnost použití funkcí z výše zmíněných knihoven. Zaveďme si proto následující značení.

Značení 5. *Pro $l \in \{1,2,\ldots,k\}$ označme:*

$$\frac{\partial s}{\partial b^l}\left(a^k,y\right) = \begin{pmatrix}\frac{\partial s}{\partial b_1^l}\left(a^k,y\right)\\ \frac{\partial s}{\partial b_2^l}\left(a^k,y\right)\\ \vdots \\ \frac{\partial s}{\partial b_{n_l}^l}\left(a^k,y\right)\end{pmatrix},$$

$$\frac{\partial s}{\partial w^l}\left(a^k,y\right) = \begin{pmatrix}\frac{\partial s}{\partial w_1^l}\left(a^k,y\right)\\ \frac{\partial s}{\partial w_2^l}\left(a^k,y\right)\\ \vdots \\ \frac{\partial s}{\partial w_{n_l}^l}\left(a^k,y\right)\end{pmatrix} = \begin{pmatrix}\frac{\partial s}{\partial w_{1,1}^l} & \frac{\partial s}{\partial w_{1,2}^l} & \cdots & \frac{\partial s}{\partial w_{1,n_{l-1}}^l}\\ \frac{\partial s}{\partial w_{2,1}^l} & \frac{\partial s}{\partial w_{2,2}^l} & \cdots & \frac{\partial s}{\partial w_{2,n_{l-1}}^l}\\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial s}{\partial w_{n_l,1}^l} & \frac{\partial s}{\partial w_{n_l,2}^l} & \cdots & \frac{\partial s}{\partial w_{n_l,n_{l-1}}^l}\end{pmatrix}.$$

Analogicky zavedeme symboly $\frac{\partial L}{\partial b^l}(\theta, D)$ a $\frac{\partial L}{\partial w^l}(\theta, D)$.

Následující tvrzení je přímým důsledkem (dokázat bychom jej mohli rozepsáním do složek) Tvrzení 2.

Tvrzení 3. *Pro všechna* $l \in \{1, 2, \ldots, k\}$ *platí:*

$$\frac{\partial s}{\partial b^l}\left(a^k, y\right) = \delta^l\left(a^k, y\right)$$
$$\frac{\partial s}{\partial w^l}\left(a^k, y\right) = \delta^l\left(a^k, y\right)\left(a^{l-1}\right)^T.$$

Následuje pseudokód algoritmu zpětného šíření chyby, který pro danou neuronovou síť a tréninkový dataset vypočítá $\nabla L\left(\theta, D\right)$ (viz Algoritmus 1).

Algoritmus 1 Algoritmus zpětného šíření chyby (angl. backpropagation)

Vstup: $\theta, D = \{(x_n, y_n)\}_{n=1}^N$
Výstup: $\nabla L\left(\theta, D\right)$
$\nabla L\left(\theta, D\right) \leftarrow 0$
for $n = 1, 2, \ldots, N$ **do**
  $a_0 \leftarrow x_n$
  for $l = 1, 2, \ldots k$ **do**
    $z^l \leftarrow W^l a^{l-1} + b^l$
    $a^l \leftarrow \sigma\left(z^l\right)$
  end for
  $\delta^k \leftarrow \sigma'\left(z^k\right) \odot \left(a^k - y_n\right)$
  $\frac{\partial L}{\partial b^k} \leftarrow \frac{\partial L}{\partial b^k} + \delta^k$
  $\frac{\partial L}{\partial w^k} \leftarrow \frac{\partial L}{\partial w^k} + \delta^k \cdot \left(a^k\right)^T$
  for $l = k-1, k-2, \ldots, 1$ **do**
    $\delta^l \leftarrow \sigma'\left(z^l\right) \odot \left(W^{l+1}\right)^T \delta^{l+1}$
    $\frac{\partial L}{\partial b^l} \leftarrow \frac{\partial L}{\partial b^l} + \delta^l$
    $\frac{\partial L}{\partial w^l} \leftarrow \frac{\partial L}{\partial w^l} + \delta^l \cdot \left(a^l\right)^T$
  end for
end for
$\nabla L\left(\theta, D\right) \leftarrow \frac{1}{N} \nabla L\left(\theta, D\right)$

Všimněme si, že algoritmus zpětného šíření chyby počítá hodnoty δ^l, $\partial L/\partial b^l$, $\partial L/\partial w^l$, $l \in \{1, 2, \ldots, k\}$ efektivně - žádnou hodnotu nepočítá dvakrát. Je to díky výběru „směru“, ve kterém zmíněné hodnoty počítá. Jelikož hodnoty příslušící l-té vrstvě lze spočítat z hodnot příslušících $(l+1)$-ní vrstvě, je výhodné iterovat od poslední vrstvy k první, pamatovat si již spočtené výsledky a použít je při dalších výpočtech. Lze tedy říct, že algoritmus zpětného šíření chyby využívá principu dynamického programování.

3. Optimalizační algoritmy

V této kapitole popíšeme algoritmy, které se k tréninku neuronových sítí používají. Nejprve představíme klasické verze těchto algoritmů, které jsou obecně známé jakožto optimalizační algoritmy. U neuronových sítí se pak běžně používají stochastické verze těchto algoritmů - o tom ale až později.

Téměř všechny zde popsané algoritmy jsou úpravami metody největšího spádu a ve většině případů mají pouze lineární řád konvergence. Tyto metody mají společné, že nevyužívají jinou informaci o neuronové síti (resp. o ztrátové funkci) než gradient.

Nebyla by rychlejší Newtonova metoda, která za jistých podmínek zaručuje kvadratický řád konvergence? V jiných případech vhodnější Newtonova metoda je bohužel v případě neuronových sítí nepoužitelná. Kdybychom chtěli k tréninku neuronové sítě použít Newtonovu metodu, potřebovali bychom znát matici druhých derivací ztrátové funkce. Předpokládejme, že by se nám úspěšně podařilo odvodit vzorce a algoritmus pro výpočet této matice. Tak či tak bychom narazili na problém: Velikost této matice odpovídá počtu druhých derivací ztrátové funkce, přičemž tento počet roste kvadraticky s počtem proměnných ztrátové funkce - tedy s počtem parametrů sítě.

Přitom neuronová síť, kterou v Kapitole 4.4 použijeme pro řešení jednoduchého kanonického problému pro neuronové sítě, která je na dnešní poměry velmi malá, bude obsahovat okolo dvaceti tisíc parametrů. Matice druhých derivací by tedy obsahovala zhruba čtyři sta miliónů prvků. Zprůměrovat tyto hodnoty přes tréninkový dataset, který bude obsahovat padesát tisíc tréninkových vzorků, by tedy mohlo být výpočetně tak náročné, že bychom sotva udělali jeden krok. Pro neuronové sítě, které se dnes používají v praxi, jejichž počet parametrů je řádově mnohem větší, bychom tuto matici jistě spočítat nedokázali.

Výjimkou této kapitoly bude Nelder-Meadova simplexová metoda, jejíž způsob fungování není založen na znalosti gradientu účelové funkce, ale hledá minimum funkce na základě hodnot funkce samotné. Tato metoda si dokáže poradit s akademickými protipříklady na metody spádového typu, a pro zajímavost ji v Kapitole 4.3 vyzkoušíme i na úloze trénování neuronové sítě.

3.1 Představení jednotlivých metod

Než ale začneme metody testovat, postupně si je všechny představíme. V každé následující sekci popíšeme, popř. odvodíme, jednu z metod, která hledá lokální minimum funkce $f : \mathbb{R}^d \to \mathbb{R}$, $f \in \mathcal{C}^1\left(\mathbb{R}^d\right)$, kde $d \in \mathbb{N}$. Ve většině případů bude optimalizace fungovat následovně: Zvolíme počáteční aproximaci $x_0 \in \mathbb{R}^d$ (aproximace lokálního minima), v prvním kroku najdeme vektor $u_0 \in \mathbb{R}^d$ (daný definicí optimalizační metody), zvolíme koeficient $\eta_0 \in (0, \infty)$ a položíme $x_1 = x_0 + \eta_0 u_0$. Obecně, máme-li již k dispozici aproximaci $x_n \in \mathbb{R}^d$, pak podle formule z definice optimalizační metody najdeme vektor $u_n \in \mathbb{R}^d$, zvolíme koeficient $\eta_n \in (0, \infty)$ a položíme $x_{n+1} = x_n + \eta_n u_n$. Následně očekáváme konvergenci $x_n \xrightarrow{n\to\infty} x_*$, kde $x_* \in \mathbb{R}^d$ je lokální minimum funkce f.

Téměř všechny metody, které zde představíme, lze najít v [5], kde je popsáno i mnoho dalších metod, které lze pro trénování neuronových sítí použít.

3.1.1 Metoda největšího spádu (GD)

Začneme metodou, která je základním kamenem pro odvození ostatních metod spádového typu - *metodou největšího spádu*, angl. *gradient descent* (odtud zkratka *GD*). Metodu zde odvodíme; další informace a teorii týkající se metody největšího spádu lze případně najít ve většině učebnic základů numerické optimalizace (např. [7]).

Jelikož chceme konvergovat k nějakému lokálnímu minimu, je přirozené požadovat, aby každá aproximace $x_n, n \in \mathbb{N}_0 \left(:= \mathbb{N} \cup \{0\}\right)$ splňovala

$$f\left(x_{n+1}\right) < f\left(x_n\right).$$

Rozepsáním x_{n+1} dostaneme:

$$f\left(x_n + \eta_n u_n\right) < f\left(x_n\right).$$

Geometrická interpretace požadavku: posuneme-li se z bodu x_n o malý násobek vektoru u_n (číslo η_n z praktických důvodů nebývá velké), chceme, abychom přišli do místa, kde je hodnota funkce f nižší. To motivuje následující definici.

Definice 11 (Spádový směr)**.** *Nechť $f : \mathbb{R}^d \to \mathbb{R}$ je funkce, $x \in \mathbb{R}^d$ je libovolné, $u \in \mathbb{R}^d$ je libovolný směr. Řekneme, že směr u je* spádový*, pokud existuje $\eta_0 \in (0, \infty)$ takové, že pro všechna $\eta \in (0, \eta_0)$ platí:*

$$f\left(x + \eta u\right) < f\left(x\right).$$

Poznámka. V následujícím budeme symbolem „$\cdot$“ značit standardní skalární součin v $\mathbb{R}^d$.

Pro funkce s hladkostí $\mathcal{C}^1\left(\mathbb{R}^d\right)$ máme dokonce lemma, které hovoří o tom, kdy je směr spádový.

Lemma 4 (Charakterizace spádových směrů)**.** *Nechť $f \in \mathcal{C}^1\left(\mathbb{R}^d\right)$, $x \in \mathbb{R}^d$ je libovolné, $u \in \mathbb{R}^d$ je libovolný směr splňující $\nabla f\left(x\right) \cdot u \neq 0$. Pak $\nabla f\left(x\right) \cdot u < 0$ právě tehdy, když u je spádový směr.*

Důkaz. Definujme funkci $g : [0, \infty) \to \mathbb{R}$, $g\left(\eta\right) = f\left(x + \eta u\right)$. Z existence a spojitosti parciálních derivací funkcí f a $\eta \mapsto x + \eta u, \eta \in [0, \infty)$ máme pro každé $\eta \in (0, \infty)$ vztah $g'\left(\eta\right) = \nabla f\left(x + \eta u\right) \cdot u$. Navíc ze spojitosti g v nule je $g'_+\left(0\right) = \nabla f\left(x\right) \cdot u$.
$\Longrightarrow$: Je-li $\nabla f\left(x\right) \cdot u < 0$, pak

$$0 > \nabla f\left(x\right) \cdot u = g'_+\left(0\right),$$

a tedy ze spojitosti g' existuje $\delta > 0$ takové, že $g'\left(\eta\right) < 0$ pro všechna $\eta \in (0, \delta)$, tedy g je na $[0, \delta)$ ryze klesající; volme $\eta_0 = \delta$, pak speciálně pro každé $\eta \in (0, \eta_0)$ je

$$f\left(x + \eta u\right) = g\left(\eta\right) < g\left(0\right) = f\left(x\right),$$

což znamená, že u je spádový směr.

$\Longleftarrow$: Naopak, je-li u spádový směr, pak existuje $\eta_0 \in \mathbb{R}$, $\eta_0 > 0$ takové, že pro všechna $\eta \in (0, \eta_0)$ platí:

$$g(\eta) = f(x + \eta u) < f(x) = g(0). \tag{3.1}$$

Chceme ukázat $g'_+(0) < 0$. Předpokládejme pro spor $g'_+(0) > 0$ (podle předpokladů $g'_+(0) = \nabla f(x) \cdot u \neq 0$). Potom ze spojitosti g' existuje $\delta \in \mathbb{R}$, $\delta > 0$ takové, že $g'(\eta) > 0$ pro všechna $\eta \in (0, \delta)$, tedy g je na $[0, \delta)$ ryze rostoucí; nechť $\xi \in \mathbb{R}, 0 < \xi < \min\{\delta, \eta_0\}$ je libovolné. Potom díky ryzí monotonii na příslušném intervalu máme:

$$g(\xi) > g(0),$$

což dává spor s (3.1). Tedy vskutku

$$0 > g'_+(0) = \nabla f(x) \cdot u.$$

Tím je důkaz dokončen.

□

Jak název metody napovídá, bude využívat spádového směru, ve kterém funkce klesá nejvíce.

Definice 12 (Směr největšího spádu)**.** *Nechť $f \in \mathcal{C}^1\left(\mathbb{R}^d\right)$, $x \in \mathbb{R}^d$ je libovolné. Směr $u \in \mathbb{R}^d$ se nazývá* směr největšího spádu*, pokud u splňuje:*

1. *Směr u je spádový,*
2. $\|u\| = 1$,
3. $\nabla f(x) \cdot u = \min\left\{\nabla f(x) \cdot v, v \in \mathbb{R}^d, \|v\| = 1\right\}$.

Poznámka. Minimum ze třetího bodu předešlé definice vždy existuje. Uzavřená jednotková koule $B_{\mathbb{R}^d} = \left\{v \in \mathbb{R}^d, \|v\| \leq 1\right\}$ je kompaktní v $\mathbb{R}^d$ ($\mathbb{R}^d$ s eukleidovskou normou tvoří normovaný lineární prostor nad $\mathbb{R}$, a tedy kompaktost plyne z faktu, že $\dim\left(\mathbb{R}^d\right) = d < \infty$), a jednotková sféra $S_{R^d} = \left\{v \in \mathbb{R}^d, \|v\| = 1\right\}$ je uzavřenou podmnožinou $B_{\mathbb{R}^d}$ (je vzorem uzavřené množiny $\{1\}$ při spojitém zobrazení $v \mapsto \|v\|$, $v \in B_{\mathbb{R}^d}$). Spojitá funkce $v \mapsto \nabla f(x) \cdot v, v \in S_{\mathbb{R}^d}$ zde tedy svého minima nabývá.

O tom, jak nalézt směr největšího spádu hovoří následující lemma.

Lemma 5 (O směru největšího spádu)**.** *Nechť $f \in \mathcal{C}^1\left(\mathbb{R}^d\right)$, $x \in \mathbb{R}^d$ je libovolné. Je-li $\nabla f(x) \neq 0$, pak směr $-\nabla f(x) / \|\nabla f(x)\|$ je směr největšího spádu.*

Důkaz. Pro jednoduchost zápisu označme $u = -\nabla f(x) / \|\nabla f(x)\|$. Platí:

$$\begin{aligned} \nabla f(x) \cdot u = \nabla f(x) \cdot \frac{-\nabla f(x)}{\|\nabla f(x)\|} = -\frac{1}{\|\nabla f(x)\|} \left(\nabla f(x) \cdot \nabla f(x)\right) = \\ = -\frac{1}{\|\nabla f(x)\|} \|\nabla f(x)\|^2 = -\|\nabla f(x)\| < 0, \end{aligned} \tag{3.2}$$

a tedy podle Lemmatu 4 je u spádový směr. Platnost podmínky $\|u\| = 1$ je zřejmá, a tedy zbývá dokázat, že v u se nabývá minima funkce $v \mapsto \nabla f(x) \cdot v$, $v \in S_{\mathbb{R}^d}$. Nechť $v \in S_{\mathbb{R}^d}$ je dáno libovolné. Cauchy-Schwarzova nerovnost dává:

$$|\nabla f(x) \cdot v| \leq \|\nabla f(x)\| \|v\| = \|\nabla f(x)\|,$$

a tedy speciálně platí:

$$\nabla f(x) \cdot v \geq -\|\nabla f(x)\|.$$

Tedy také minimum na $S_{\mathbb{R}^d}$ musí být větší nebo rovno $-\|\nabla f(x)\|$. Podle výpočtu 3.2 se ale tohoto minima v u nabývá. Tedy vskutku u je směr největšího spádu.

□

Existuje-li tedy směr největšího spádu, umíme jej jednoduše najít - máme pro něj explicitní vzoreček. Navíc podle Lemmatu 5 pro existenci směru největšího spádu stačí, aby byl gradient účelové funkce nenulový. Není problém, že v případě nulového gradientu směr největšího spádu nemusí existovat? Je-li gradient účelové funkce nulový, je splněna nutná podmínka lokálního extrému, a tedy metoda možná našla požadované lokální minimum - neexistence směru největšího spádu nám tedy nevadí, naopak - máme radost.

Jedná se ovšem pouze o nutnou podmínku lokálního extrému, nikoliv o podmínku postačující; metoda tedy mohla najít pouze stacionární bod. V tomto bodě, jak uvidíme v následující definici, se metoda zastaví. Problém uváznutí ve stacionárním bodě, který není lokálním minimem, metoda nijak neřeší, a tak se tento problém stává jedním z nedostatků metody největšího spádu.

Přejděme nyní k samotné definici metody největšího spádu.

Definice 13 (Metoda největšího spádu)**.** *Nechť $f \in \mathcal{C}^1\left(\mathbb{R}^d\right)$ je účelová funkce, $x_0 \in \mathbb{R}^d$ je libovolná počáteční aproximace. O posloupnosti $\{x_n\}_{n=0}^{\infty}$ řekneme, že vznikla* metodou největšího spádu*, pokud pro každé $n \in \mathbb{N}_0$ platí:*

$$x_{n+1} = x_n + u_n,$$

kde

$$u_n = -\eta_n \nabla f(x_n),$$
$$\eta_n \in (0, \infty).$$

Poznámka. V praxi vždy vypočítáme pouze konečnou část posloupnosti z předchozí definice. Rozhodnutí, kdy výpočet ukončíme přitom činíme na základě předem daných rozhodovacích kritérií.

Poznámka. Abychom rozlišili parametry optimalizační metody (jako například η_n) od parametrů sítě, budeme parametrům metody říkat *hyperparametry.*

Poznámka. Z praktických důvodů je značení v předchozí definici mírně nekonzistentní s předchozím textem - formule pro x_{n+1} již neobsahuje krok η_n, ten je totiž skrytý v samotném u_n. Pro zavedení metod, které zobecňují metodu největšího spádu, které jsou popř. nadstavbou nad metodu největšího spádu, totiž budeme potřebovat více hyperparametrů - samotný krok pak hraje jinou roli než zde. Pro odvození, které jsme provedli výše se však lépe hodí dříve použité značení.

Nabízí se otázka, jak volit krok η_n aby metoda fungovala rozumně.

Poznámka. Krok $\eta_n \in (0,\infty), n \in \mathbb{N}_0$ lze volit několika způsoby:

1. *Přesný line-search:* směr kroku nám metoda předepsala nejlepší možný - v tomto směru účelová funkce klesá nejvíce; podobně můžeme volit i krok - najdeme krok splňující:

$$f\left(x_n - \eta_n \nabla f\left(x_n\right)\right) = \min_{\eta \in (0,\infty)} f\left(x_n - \eta \nabla f\left(x_n\right)\right).$$

   K tomu můžeme používat jiné optimalizační metody - narozdíl od úlohy výše řešíme pouze jedno-dimenzionální problém. Tento způsob volby kroku se může zdát příliš rafinovaný, a dalo by se čekat, že bude velmi účinný. Nicméně za chvíli uvidíme (viz Lemma 6), že v případě metody největšího spádu s přesným line-search dochází k tzv. *zig-zag efektu* - po sobě jdoucí směry $-\nabla f\left(x_n\right)$ a $-\nabla f\left(x_{n+1}\right)$ jsou na sebe kolmé, a to obvykle konvergenci naopak zpomaluje.

2. *Přibližný line-search:* zig-zag efektu se částečně můžeme vyhnout znepřesněním optimalizace kroku. Například místo přísného zastavovacího kritéria 1D optimalizace můžeme vždy provést jen pár kroků příslušné metody.

3. *Klesající posloupnost velikostí kroku:* často se může hodit po čase „zjemnit krok“. Jsme-li blíž hledaného stacionárního bodu, velký krok může zapříčinit, že stacionární bod „přeskočíme“ - časem se nám hodí jemnější krok. Můžeme tedy volit $(\eta_n)_{n=0}^{\infty} \subseteq (0,\infty)$, $\eta_n \xrightarrow{n\to\infty} 0$. Posloupnost $(2^{-n})_{n=0}^{\infty}$ splňuje požadované podmínky, ale platí $\sum_{n=0}^{\infty} 2^{-n} = 2$ - kdyby pro každé $n \in \mathbb{N}_0$ platilo $\| -\nabla f\left(x_n\right) \| = 1$ - ke stacionárnímu bodu, který je vzdálený více než dvě jednotky od počáteční aproximace, bychom opět nedošli. Abychom se tomuto jevu vyhnuli, přidáváme podmínku $\sum_{n=0}^{\infty} \eta_n = \infty$.

4. *Konstantní krok:* nejjednodušší možností je zvolit $\eta_0 \in (0,\infty)$ a pro $n \in \mathbb{N}$ pak volit $\eta_n = \eta_0$. Zvolíme-li dost malý krok η_0, může metoda fungovat dost dobře. Při testování metod a při aplikaci na neuronových sítích se v této práci budeme uchylovat právě k této volbě. Při trénování neuronových sítí obecně je volba konstantního kroku nejběžnější - pro konstantu η_0 se používá anglický termín *learning-rate*.

Následuje slíbené lemma.

Lemma 6 (Zig-zag effect)**.** *Nechť $f \in \mathcal{C}^1\left(\mathbb{R}^d\right)$ je účelová funkce a posloupnost $\{x_n\}_{n=0}^{\infty}$ vznikla metodou největšího spádu s optimální délkou kroku, tj. vznikla metodou největšího spádu a pro každé $n \in \mathbb{N}_0$ velikost kroku η_n splňuje:*

$$f\left(x_n - \eta_n \nabla f\left(x_n\right)\right) = \min_{\eta \in (0,\infty)} f\left(x_n - \eta \nabla f\left(x_n\right)\right).$$

Potom každé dva po sobě jdoucí směry jsou na sebe kolmé, neboli pro každé $n \in \mathbb{N}$ platí $u_{n+1} \perp u_n$.

Důkaz. Nechť $n \in \mathbb{N}_0$ je dáno libovolné; z definice u_n je zřejmé, že stačí ukázat ekvivalentní podmínku:

$$\nabla f\left(x_{n+1}\right) \cdot \nabla f\left(x_n\right) = 0. \tag{3.3}$$

Zavedeme si pomocnou funkci jako v důkazu Lemmatu 4; definujme

$$g : (0, \infty) \to \mathbb{R}, \quad g(\eta) = f(x_n - \eta \nabla f(x_n)).$$

Z existence a spojitosti parciálních derivací příslušných funkcí máme opět pro každé $\eta \in (0, \infty)$ vztah $g'(\eta) = -\nabla f(x_n - \eta \nabla f(x_n)) \cdot \nabla f(x_n)$.

Podle předpokladů funkce g v bodě η_n nabývá svého minima, a tak z věty o nutné podmínce pro (lokální) extrém je zde derivace funkce g rovna nule; tedy:

$$0 = g'(\eta_n) = -\nabla f(x_n - \eta_n \nabla f(x_n)) \cdot \nabla f(x_n) = -\nabla f(x_{n+1}) \cdot \nabla f(x_n),$$

což je ekvivalentní rovnosti (3.3).

□

Pozitivní teoretický výsledek pro volbu optimálního kroku dává následující věta (pro důkaz viz [7], Věta 22).

Věta 7. *Nechť $f \in \mathcal{C}^2\left(\mathbb{R}^d\right)$ je funkce a nechť $\{x_n\}_{n=0}^{\infty}$ je posloupnost, která vznikla metodou největšího spádu s optimálním krokem (přesný line-search). Pak pro každý hromadný bod x posloupnosti $\{x_n\}_{n=0}^{\infty}$ platí $\nabla f(x) = 0$.*

Všimněme si nakonec, že již máme vše potřebné pro trénování sítě metodou největšího spádu; vesměs nám stačí znát pouze gradient ztrátové funkce, který ale spočítat umíme, díky Algoritmu 1. V praxi se sice na trénování neuronových sítí používají stochastické verze zde popsaných metod, ale o tom až později v Sekci 3.2.

3.1.2 Metoda největšího spádu s hybností (GDM)

Nadstavbou nad metodou největšího spádu je *metoda největšího spádu s hybností*, původně anglicky *gradient descent with momentum* (odtud zkratka *GDM*), v angličtině také známá jako *heavy-ball method*. Obsáhlejším zdrojem k GDM je například [8], Kapitola 4.

Alternativní název heavy-ball method přímo navádí na fyzikální analogii popisující chování této metody - představíme-li si graf účelové funkce jako „terén“ v $\mathbb{R}^{d+1}$ (d-plochu v $\mathbb{R}^{d+1}$), pak trajektorie balónku s nenulovou hmotností, který spustíme po terénu z bodu odpovídajícího počáteční aproximaci $x_0 \in \mathbb{R}^d$, odpovídá (alespoň přibližně) interpolaci posloupnosti nalezené metodou GDM.

Tedy na rozdíl od metody největšího spádu má na konstruovanou posloupnost vliv jakási „setrvačnost“ způsobená předchozími polohami, resp. předchozími pohyby, pomyslného balónku.

Přejděme k definici metody.

Definice 14 (Metoda největšího spádu s hybností)**.** *Nechť $f \in \mathcal{C}^1\left(\mathbb{R}^d\right)$ je účelová funkce, $x_0 \in \mathbb{R}^d$ je libovolná počáteční aproximace, $m \in [0, 1)$ je koeficient hybnosti. O posloupnosti $\{x_n\}_{n=0}^{\infty}$ řekneme, že vznikla* metodou největšího spádu s hybností*, pokud pro každé $n \in \mathbb{N}_0$ platí:*

$$x_{n+1} = x_n + u_n,$$

kde

$$\begin{aligned} u_n &= m u_{n-1} - (1 - m)\, \eta_n \nabla f(x_n), \\ u_{-1} &= 0, \\ \eta_n &\in (0, \infty). \end{aligned}$$

Poznámka. V předchozí definici je vektor u_n konvexní kombinace vektorů u_{n-1} a $-\eta_n \nabla f(x_n)$ pro každé $n \in \mathbb{N}_0$.

Poznámka. Metodu GDM lze chápat jako zobecnění metody GD, neboť volbou koeficientu hybnosti $m = 0$ dostáváme přesně metodu GD. Článek [9] doporučuje volbu $m = 0.9$. Některé definice metody GDM povolují hodnotu $m = 1$, nicméně je zřejmé, že při této volbě by výsledná posloupnost byla konstantní - fyzikální analogie odpovídá nekonečnému tření, které brání pohybu.

Uvažujme pevné $n \in \mathbb{N}$, konstantní krok $\eta_n = \eta \in (0, \infty)$ a pracujme s předpisem pro u_n z předchozí definice:

$$u_n = m u_{n-1} - (1 - m)\, \eta_n \nabla f(x_n).$$

Od obou stran rovnic odečtěme u_{n-1} a následně rovnici vydělme číslem $1 - m$. Dostáváme:

$$\frac{u_n - u_{n-1}}{1 - m} = -u_{n-1} - \eta \nabla f(x_n),$$

což je aproximace rovnice:

$$\frac{du}{dt} = -u - \eta \nabla f(x)$$

metodou konečných diferencí. Tato rovnice popisuje pohyb částice se třením (u je zde rychlost částice, která se v čase t nachází v bodě $x(t)$, tedy $u = dx/dt$). Podrobnější popis fyzikálních analogií pro metody GD a GDM lze najít v [8]. Na základě fyzikální analogie lze také odvodit Nesterovovu metodu, o které pojednává následující kapitola.

3.1.3 Nesterovova metoda (NAG)

Podobně jako metoda GDM principu hybnosti využívá také *Nesterovova metoda* (zkratka *NAG* pochází z anglického *Nesterov accelerated gradient*). Více o Nesterovově metodě lze najít opět v [8]. Přejděme nyní k definici metody.

Definice 15 (Nesterovova metoda)**.** *Nechť $f \in \mathcal{C}^1\left(\mathbb{R}^d\right)$ je účelová funkce, $x_0 \in \mathbb{R}^d$ je libovolná počáteční aproximace, $m \in [0, 1)$ je koeficient hybnosti. O posloupnosti $\{x_n\}_{n=0}^{\infty}$ řekneme, že vznikla* Nesterovovou metodou*, pokud pro každé $n \in \mathbb{N}_0$ platí:*

$$x_{n+1} = x_n + u_n,$$

kde

$$\begin{aligned} u_n &= m u_{n-1} - (1 - m)\, \eta_n \nabla f(x_n + m u_{n-1}), \\ u_{-1} &= 0, \\ \eta_n &\in (0, \infty). \end{aligned}$$

Poznámka. Všimněme si, že od metody GDM se Nesterovova metoda liší pouze v jednom - gradient účelové funkce počítá „v bodě, kam by balónek dojel, kdyby na něj měla vliv pouze předchozí setrvačnost“.

I přes to, že se definice Nesterovovy metody moc neliší od definice metody GDM, má obvykle v porovnání s metodou GDM daleko lepší výsledky. Podle [5] má za jistých předpokladů Nesterovova metoda kvadratický řád konvergence. Pro stochastickou verzi Nesterovovy metody, která se používá k trénování neuronových sítí, toto tvrzení neplatí, stochastická verze má opět pouze lineární řád konvergence, opět viz [5]. Společně s metodou Adam (viz Sekce 3.1.5) je dnes v kontextu neuronových sítí považována za „state of the art“ metodu.

3.1.4 AdaGrad (Adaptivní gradient)

Algoritmus *AdaGrad* (zkratka pro *adaptive gradient algorithm*) byl vyvinut jako nadstavba nad metodu největšího spádu speciálně proto, aby řešil problém výběru velikosti kroku, resp. problém výběru learning-rate. Jak jsme již dříve zmínili, budeme v případě metody největšího spádu volit konstantní velikost kroku. Pro různé účelové funkce, resp. pro různé struktury neuronových sítí, mohou být vhodné jiné velikosti kroku, resp. jiné hodnoty learning-rate; přitom nalezení hodnot, které budou fungovat správně, nemusí být jednoduché.

AdaGrad v průběhu výpočtu upravuje velikost kroku, a tedy tuto úlohu řeší za nás. Bere přitom ohled na velikosti gradientů účelové funkce v bodech, kde probíhá výpočet. Na začátku optimalizace je velikost kroku největší a postupem času, když přibývá bodů, ve kterých proběhl výpočet, když se akumulují velikosti předchozích gradientů, algoritmus krok zmenšuje. Proč může být výhodné postupem času zmenšovat velikost kroku jsme již naznačili v jedné z poznámek za Definicí 13 - hledaný stacionární bod bychom v případě volby příliš velkého kroku mohli „přeskočit“.

AdaGrad byl původně představen v [10]; my ale budeme pracovat s formulací uvedenou v [5].

Definice 16 (AdaGrad)**.** *Nechť $f \in \mathcal{C}^1\left(\mathbb{R}^d\right)$ je účelová funkce, $x_0 \in \mathbb{R}^d$ je libovolná počáteční aproximace, $\eta \in (0,\infty)$, $\varepsilon \in (0,\infty)$. O posloupnosti $\{x_n\}_{n=0}^{\infty}$ řekneme, že vznikla* metodou AdaGrad*, pokud pro každé $n \in \mathbb{N}_0$ platí:*

$$x_{n+1} = x_n + u_n,$$

kde

$$u_n = -\eta \frac{\nabla f\left(x_n\right)}{\sqrt{\varepsilon + \sum_{k=0}^{n} \left(\nabla f\left(x_k\right)\right)^2}},$$

kde dělení vektorů provádíme po složkách, aplikaci funkce na vektor chápeme po složkách a přičtení konstanty k vektoru znamená přičtení dané konstanty ke každé složce vektoru.

Poznámka. Konstanta ε z předchozí definice slouží pouze jako prevence případného dělení nulou.

Poznámka. I přes to, že AdaGrad za nás řeší volbu velikosti kroku, máme k dispozici hyperparametr η (viz předchozí definice), kterým můžeme algoritmus popohnat nebo naopak zkrotit. Podle [5] většina implementací standardně volí $\eta = 0.01$.

Jako každá metoda i AdaGrad má své problémy. Jako první problém můžeme chápat potřebu určit konstantu η, přestože tomuto jsme se chtěli prvotně vyhnout. Zásadním problémem této metody je rychlé klesání velikosti kroku k nule - v reakci na tento problém byly vyvinuty optimalizační algoritmy AdaDelta a RMSProp (představeny postupně v [11] a [12]); ty místo započítání všech předchozích gradientů počítají s jejich exponenciálně klouzavým průměrem. Algoritmy AdaDelta a RMSProp přeskočíme a podíváme se na metodu, která je rozšiřuje.

3.1.5 Adaptive moment estimation (Adam)

Dostáváme se k metodě *Adam*, která je dnes společně s Nesterovovou metodou považována za „state of the art" metodu pro trénování neuronových sítí. Představena byla v roce 2014 v [13] a podobně jako AdaGrad je i metoda Adam založena na „úpravě hyperparametrů za chodu". Adam přizpůsobuje velikost kroku na základě exponenciálního klouzavého průměru předchozích gradientů. Exponenciální klouzavý průměr metoda také používá pro volbu gradientu, s čímž jsme se již setkali například u metody GDM.

Definice 17 (Adam). *Nechť $f \in \mathcal{C}^1\left(\mathbb{R}^d\right)$ je účelová funkce, $x_0 \in \mathbb{R}^d$ je libovolná počáteční aproximace, $\eta \in (0,\infty)$, $\varepsilon \in (0,\infty)$, $\beta_1, \beta_2 \in [0,1)$. O posloupnosti $\{x_n\}_{n=0}^{\infty}$ řekneme, že vznikla* metodou Adam*, pokud pro každé $n \in \mathbb{N}_0$ platí:*

$$x_{n+1} = x_n + u_n,$$

kde

$$\begin{aligned} u_n &= -\eta \frac{\sqrt{1-\beta_2}}{1-\beta_1} \frac{v_n}{V_n + \varepsilon}, \\ v_n &= \beta_1 v_{n-1} + (1-\beta_1) \nabla f(x_n), \\ V_n &= \sqrt{\beta_2 V_{n-1} + (1-\beta_2) \left(\nabla f(x_n)\right)^2}, \end{aligned}$$

kde dělení vektorů provádíme po složkách, aplikaci funkce na vektor chápeme po složkách a přičtení konstanty k vektoru znamená přičtení dané konstanty ke každé složce vektoru.

Poznámka. Článek [5] uvádí, že vhodné hodnoty hyperparametrů β_1, β_2 a ε jsou postupně $0.9, 0.999$ a 10^{-8}.

3.1.6 Nelder-Meadova simplexová metoda (NM)

Na závěr si představíme *Nelder-Meadovu simplexovou metodu*, která se od všech dříve zmíněných metod velmi liší - vůbec nepoužívá informaci o gradientu účelové funkce. Místo toho pracuje pouze s hodnotami účelové funkce. Jak uvidíme v Kapitole 4.2, díky nezávislosti na gradientu neselže na jistém protipříkladu, se kterým má většina dříve zmíněných metod problém.

Jelikož zde metodu uvádíme spíše pro zajímavost, nebudeme ji korektně definovat; uvedeme pouze myšlenku, na které je založena. Přesnou definici metody lze najít v článku [14].

Metoda je založena na pohybujícím se simplexu, jehož vrcholy by měly konvergovat k bodu minima účelové funkce. Pojmem *simplex* rozumíme konvexní obal $d + 1$ afinně nezávislých bodů, kde d je dimenze prohledávaného prostoru. Pro jednoduchost pracujme v $\mathbb{R}^2$ s trojúhelníkem.

V každém kroku vyhodnotíme účelovou funkci ve vrcholech trojúhelníku a vrcholy trojúhelníku setřídíme podle příslušných funkčních hodnot. Pokud nejmenší z těchto funkčních hodnot splňuje podmínky zastavovacího kritéria, algoritmus končí. V opačném případě spočítáme polohu ještě jednoho bodu v trojúhelníku a v závislosti na tomto bodě a poloze vrcholu s nejnižší hodnotou účelové funkce také ještě dva body mimo trojúhelník a jeden bod uvnitř trojúhelníku. Na základě vyhodnocení účelové funkce v těchto nových bodech se pak zkonstruuje nový trojúhelník, se kterým pokračujeme do dalšího kroku. Detailní popis metody přesahuje rámec tohoto textu.

Tato metoda se pro trénování neuronových sítí běžně nepoužívá, nicméně pro zajímavost Nelder-Meadovu simplexovou metodu v Kapitole 4.3 otestujeme i v neuronové síti. Nutno však zmínit, že s metodou se v kontextu neuronových sítí lze setkat. Nepoužívá se však pro trénink sítě, ale lze ji použít pro nalezení vhodných hyperparametrů (learning-rate, koeficient hybnosti); v této práci ji však pro tento účel nepoužijeme.

3.2 Stochastické verze spádových metod

Jak jsme již dříve zmínili, pro trénování neuronových sítí se používají tzv. *stochastické* verze výše představených optimalizačních metod. Proces, jak z obyčejné metody odvodit její stochastickou verzi, je velmi jednoduchý (viz níže), nicméně je třeba podotknout, že teoretické výsledky, které máme pro obyčejné metody (např. pro GD - gradient descent), nelze aplikovat na jejich stochastické verze (např. *SGD - stochastic gradient descent*); jedná se o dva naprosto odlišné matematické objekty.

Jak víme z Definice 9, pro vyhodnocení ztrátové funkce L musíme iterovat celým tréninkovým datasetem; taktéž musíme učinit pro výpočet gradientu ztrátové funkce L (viz Algoritmus 1). V praxi však chceme používat velké a rozsáhlé datasety; jeden krok spádové metody pak může být drahý na výpočetní čas. Ukazuje se, že daleko efektivnějším přístupem je gradient na celém datasetu aproximovat gradientem ztrátové funkce na mnohem menší části datasetu a pracovat s touto aproximací. I když pak pracujeme pouze s aproximací gradientu, výpočetní čas této aproximace je velmi malý, a tedy místo jednoho lepšího kroku můžeme za stejný čas vykonat mnohem více méně přesných kroků. Praxe ukazuje, že tento postup je mnohem lepší, než počítat gradient přesně.

Postupovat tedy budeme následovně. Zahájíme tzv. *epochu*, což je pojem pro cyklus tréninku na celém datasetu. Celý dataset náhodně zpřeházíme, rozdělíme na menší datasety konstantní velikosti (říkejme jim *mini-batche*, v anglické literatuře se používá termín *mini-batch*, neexistuje ustálený český překlad) a síť trénujeme některou ze stochastických verzí spádových metod. Stochastická verze spádové metody bude v každém kroku místo gradientu ztrátové funkce uvažovat jeho aproximaci spočítanou na některé mini-batchi, přičemž v průběhu jedné epochy gradient aproximuje na každé mini-batchi právě jednou. Jakmile jsou všechny mini-batche vyčerpány, epocha končí. Postup opakujeme pro pevný počet epoch.

Důležitou částí je náhodné zpřeházení celého datasetu. Odtud mj. přívlastek *stochastický* (největší spád), popř. *stochastic* (gradient descent). Tento krok je důležitý zejména proto, aby jednotlivé aproximace gradientů na mini-batchích dobře odpovídaly gradientu na celém datasetu. Matematicky se konvergencí stochastických metod dále zabývat nebudeme, neboť tato otázka je nad rámec tohoto textu. Nicméně článek [15] poskytuje alespoň intuici (viz další odstavce), proč by stochastické zjednodušení mělo fungovat.

Mějme dataset S a dále uvažujme dataset S^*, který vznikne spojením deseti kopií datasetu S. Uvažujme tréning neuronové sítě spádovými metodami na těchto datasetech (uvažujme klasické verze optimalizačních metod, nikoliv stochastické). Gradient ztrátové funkce vyjde na obou datasetech stejně, neboť oba obsahují stejnou informaci (S^* vznikl spojením kopií S), nicméně výpočet gradientu na S^* bude trvat desetkrát déle. V tomto případě se nám zřejmě vyplatí trénovat síť na datasetu S, neboť tréning na S^* s sebou nenese žádné výhody. Při používání stochastických spádových metod pak spoléháme na to, že mini-batche s sebou nesou vesměs stejnou informaci („jsou kopiemi“ S), a dobře reprezentují celý dataset (v analogii výše S^*). Tomu, aby mini-batche nesly vesměs stejnou informaci, pomáhá právě zmíněné náhodné rozprostření tréninkových vzorků napříč mini-batchemi.

V dalším budeme pro stochastickou verzi metody největšího spádu používat zkratku *SDG* (z angl. *stochastic gradient descent*) a pro stochastickou verzi metody největšího spádu s hybností budeme používat zkratku *SGDM* (z angl. *stochastic gradient descent with momentum*). Pro stochastické verze metod NAG, AdaGrad a Adam budeme používat zkratky totožné s těmi, které používáme pro klasické verze těchto metod; vždy ale bude z kontextu jasné, zda hovoříme o klasické, nebo stochastické verzi metody.

V další kapitole představíme úlohy, na kterých jsme jednotlivé metody (jak klasické verze, tak i stochastické varianty) testovali, a také uvedeme výsledky testování.

4. Implementace a testování

V této kapitole potkáme celkem tři příklady, na kterých dříve zmíněné optimalizační metody otestujeme. Konkrétně se bude jednat o testování na akademických úlohách (jednoduché funkce z $\mathbb{R}^2$ do $\mathbb{R}$), testování na úloze klasifikace ropných vrtů, přičemž pro řešení této úlohy již použijeme miniaturní neuronovou síť, a na závěr úlohu otestujeme na kanonické úloze pro neuronové sítě - budeme trénovat rozpoznávání rukou psaných číslic.

Nejdříve ale uvedeme detailnější informace o tom, v jakém prostředí jsme úlohy řešili.

4.1 Popis prostředí a použitých technologií

K implementaci všech úloh i optimalizačních metod jsme použili programovací jazyk Python (verze 3.10.12). Python je vysokoúrovňový, všestranný, snadno čitelný programovací jazyk, pro který je k dispozici spousta knihoven užitečných pro různé účely, mezi které patří také analýza dat, nebo právě strojové učení. Podle [16] byl Python v únoru 2024 nejpopulárnějším programovacím jazykem vůbec, a to už od poloviny roku 2018, kdy v oblíbenosti mezi programátory překonal jazyk Java.

Jak bylo zmíněno, pro Python existují rozsáhlé knihovny zaměřené na strojové učení, jako jsou například TensorFlow (vyvinut společností Google pro interní použití ve výzkumu a ve výrobě) nebo třeba PyTorch (původně vyvinut MetaAI, dnes patří pod Linux Foundation). Tyto knihovny implementují jak neuronové sítě, tak i prostředí pro trénink sítí, přitom při programování si můžeme vybrat mj. z metod uvedené v Kapitole 3, které se dnes běžně v praxi používají.

Pro testování metod na tréninku neuronových sítí jsme však žádnou z těchto knihoven nepoužili a pracovali jsme výhradně s vlastní implementací, která vychází z implementace, kterou detailně prochází Michael A. Nielsen ve své knize (viz [2]). Tato implementace používá pouze standardní knihovnu jazyka Python a knihovnu NumPy (verze 1.24.1), která implementuje základní lineární algebru.

Poznámka. Pro implementaci méně podstatných funkcí, metod nebo podprogramů mezi které patří např.: metoda pro uložení, resp. načtení, parametrů neuronové sítě do, resp. ze, souboru; funkce pro vypsání statistik o trénování sítě, popř. statistik o optimalizaci; program umožnující uživateli kreslit číslice; jsme používali jazykový model ChatGPT (verze 3.5), který už dnes celkem obstojně dokáže vygenerovat daný kód i v jazyce Python. Jazykový model kód často vygeneroval tak dobře, že byly potřeba jen malé úpravy týkající se způsobu implementace zbytku projektu.

4.2 Testování na akademických úlohách

Prozatím necháme neuronové sítě stranou a otestujeme klasické verze spádových metod (a Nelder-Meadovu simplexovou metodu) na běžných funkcích z $\mathbb{R}^2$ do $\mathbb{R}$.

4.2.1 Představení úlohy

Každou z metod, které jsme představili v Kapitole 3, otestujeme na jedné ze tří funkcí, které představíme v následující definici.

Definice 18. *Definujeme*

1. kvadratickou funkci *jako funkci:*

$$p : \mathbb{R}^2 \to \mathbb{R}, \quad p\,(x, y) = x^2 + 2y^2,$$

2. hyperbolický paraboloid *jako funkci:*

$$h : \mathbb{R}^2 \to \mathbb{R}, \quad h\,(x, y) = \frac{1}{10}\left(x^2 - y^2\right)$$

3. *a* Rosenbrockovu funkci *jako funkci:*

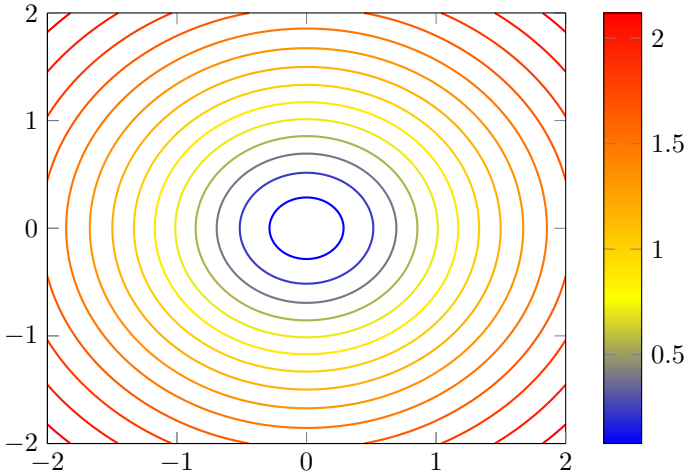
$$r : \mathbb{R}^2 \to \mathbb{R}, \quad r\,(x, y) = (1 - x)^2 + 100\left(y - x^2\right)^2.$$

Poznámka. Snadno lze odvodit, že pro všechna $(x,y) \in \mathbb{R}^2$ platí:

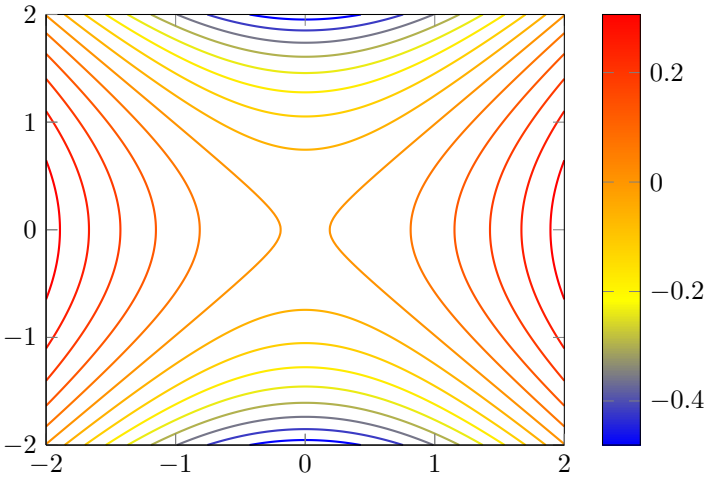
$$\begin{aligned}
\nabla p\,(x,y) &= (2x, 4y)^T, \\
\nabla h\,(x,y) &= \frac{1}{5}\,(x, -y)^T, \\
\nabla r\,(x,y) &= 2\left((x-1) - 200x\left(y - x^2\right), 100\left(y - x^2\right)\right)^T,
\end{aligned}$$

a také, že funkce p a r mají na $\mathbb{R}^2$ globální minima postupně v bodech $(0,0)^T$ a $(1,1)^T$, a že funkce h nemá na $\mathbb{R}^2$ globální minimum, má pouze stacionární (v tomto případě sedlový) bod v $(0,0)^T$. Při testování optimalizačních metod požadujeme, aby metody našly (s jistou tolerancí) právě tyto body.

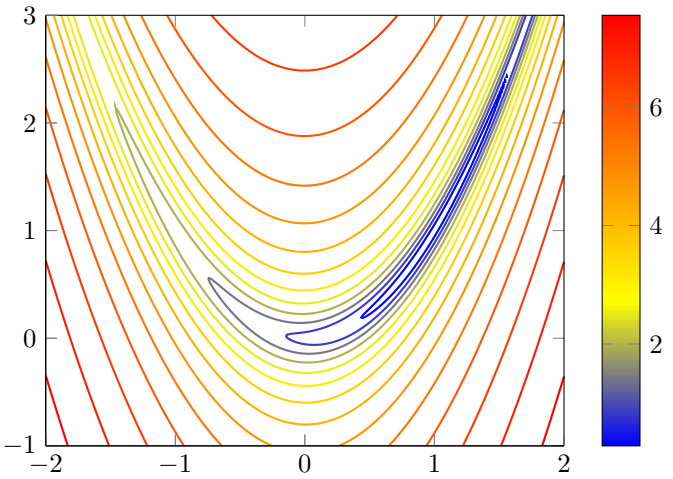
Vrstevnice funkcí p, h, r z minulé definice můžeme vidět postupně na Obrázcích 4.1, 4.2 a 4.3.



Obrázek 4.1: Vrstevnice kvadratické funkce.



Obrázek 4.2: Hyperbolický paraboloid.



Obrázek 4.3: Vrstevnice Rosenbrockovy funkce.

Při testování metod na kvadratické funkci lze očekávat velmi příznivé výsledky, neboť tato funkce je velmi jednoduchá; lze na ní pozorovat, jak se jednotlivé metody chovají za dobrých podmínek.

V případě hyperbolického paraboloidu volíme jako počáteční aproximaci bod $(1, 0)^T$; snadno lze nahlédnout, že v případě spádových metod dojde díky této volbě k uvíznutí v sedlovém bodě. Druhá složka gradientu je totiž vždy nulová. Výsledky tedy ilustrují jeden z problémů spádových metod.

Rosenbrockova funkce je typickým příkladem funkce, na které spádové metody selhávají. Na okolí globálního minima Rosenbrockovy funkce je totiž norma gradientu velmi malá, a tak metody konvergují velmi pomalu. Naopak dále od globálního minima dosahuje norma gradientu velkých čísel a spádové metody se tak odtud na okolí globálního minima vůbec nedostanou.

4.2.2 Způsob testování

Pro každou testovací funkci z minulé sekce použijeme všechny optimalizační metody představené v Kapitole 3. Hledaný stacionární bod je nám známý, a tak

volme následující zastavovací kritérium. Optimalizaci ukončíme ve chvíli, když bude velikost přímé chyby menší než daná tolerance: 10^{-8}. Jinými slovy, když bude hledaný stacionární bod vzdálený od aktuální aproximace méně než 10^{-8}, optimalizaci ukončíme. Metody porovnáme na základě toho, kolik kroků bude potřeba k nalezení takto dobré aproximace.

Aby bylo testování metod objektivní, budeme každou metodu testovat na stejné sadě hyperparametrů, které odpovídají velikosti kroku, resp. „learning-rate“, a následně pro porovnání použijeme nejlepší dosažený výsledek (nejnižší potřebný počet kroků, aby metoda našla dost dobrou aproximaci výsledku). Jednotlivé hodnoty velikosti kroku, na kterých budeme metody testovat, volme $\eta \in \{10, 5, 2, 1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001\}$. Při metodách, které jsou charakterizovány i jinými hyperparametry než je learning-rate, tyto parametry nastavíme na články doporučenou hodnotu (viz příslušné definice metod, Kapitola 3). Nelder-Meadova simplexová metoda jako jediná žádný learning-rate nepoužívá, a tedy ji budeme testovat pouze jednou, a to s doporučenými hodnotami ostatních hyperparametrů.

V případě kvadratické funkce na počáteční aproximaci moc nezáleží, a tak zvolme $x_0 = (2, 1.5)^T$. Jak jsme již dříve zmínili, na hyperbolickém paraboloidu chceme ilustrovat jeden z problémů spádových metod - uvíznutí v sedlovém bodě; volme proto počáteční aproximaci $x_0 = (1, 0)^T$. Na Rosenbrockově funkci budeme metody testovat dvakrát; v prvním kole volme příznivější počáteční aproximaci $x_0 = (2, 1.5)^T$, která se nachází na místě „s rozumnějšími podmínkami“ - norma gradientu zde není extrémní (ani moc malá, ani příliš velká); v druhém kole volme počáteční aproximaci $x_0 = (0, 0)^T$, která leží v oblasti, kde má gradient funkce velmi malou normu.

4.2.3 Výsledky testování

Následují tabulky s výsledky testování. Každá tabulka uvádí výsledky testování na jedné testovací funkci. V každé tabulce najdeme setříděné metody podle toho, jak byly úspěšné (tedy setříděné podle počtu kroků, které potřebovaly pro nalezení dost dobré aproximace), při jaké hodnotě learning-rate tohoto výsledku metoda dosáhla a nakonec vykonaný počet kroků. Navíc uvádíme také celkové pořadí mezi všemi metodami a všemi hodnotami learning-rate; běžně jsme totiž obdrželi výsledek takový, že metoda A porazila metodu B pro tři různé hodnoty learning-rate; v takovém případě tedy uvádíme pouze nejlepší výsledek metody A a neúspěch metody B se projeví na celkovém pořadí, které bude kvůli úspěchu metody A o tři pozice nižší.

Kvadratická funkce

Nejprve jsme metody testovali na nejjednodušší z testovacích funkcí, na jednoduché kvadratické funkci. Výsledky na této testovací funkci ukazuje Tabulka 4.1.

Velmi úspěšně se v žebříčku umístila metoda AdaGrad, která s hodnotami $\eta \in \{1, 2, 5, 10\}$ obsadila první čtyři pozice. Jak jsme již dříve zmínili, AdaGrad byl vyvinut, aby řešil problém nalezení správného learning-rate. Jak můžeme pozorovat, AdaGrad našel minimum rychle a přesně navzdory různým volbám learning-rate. Úspěšně se ale umístily také Nesterovova metoda, metoda největšího spádu a Nelder-Meadova simplexová metoda. Uvědomme si však, že proti

Pořadí	Metoda	η, learning rate	Počet kroků
1.	AdaGrad	2	14
5.	NAG	2	53
6.	GD	0,1	86
7.	NM	—	96
14.	Adam	10	330
15.	GDM	1	331

Tabulka 4.1: Výsledky testování optimalizačních metod na kvadratické funkci.

Newtonově metodě všechny metody uvedené výše prohrávají, neboť Newtonově metodě k nalezení globálního minima stačí pouze jedna iterace.

Hyperbolický paraboloid

Vzhledem k tomu, že na této funkci chceme pouze ilustrovat uváznutí v sedlovém bodě, nemá smysl počítat během kolika kroků se metoda dostane ke stacionárnímu bodu. Necháváme tedy proběhnout deset tisíc iterací každé metody pro každou hodnotu learning-rate, ověříme pouze, že metody opavdu uvízly.

Poznamenejme, že Nelder-Meadovu simplexovou metodu na této funkci nemá smysl testovat, neboť nemá problém s uvíznutím v sedlovém bodě. Tomuto bodu se vyhne a konstruuje posloupnost bodů, ve kterých se funkční hodnoty blíží k mínus nekonečnu.

Následují výsledky testování. Všechny spádové metody uvízly v sedlovém bodě $(0, 0)^T$, a to i pro několik různých hodnot learning-rate. Pro zajímavost zde také můžeme pozorovat, že v několika případech kvůli špatnému nastavení learning-rate (kvůli příliš nízké hodnotě tohoto hyperparametru) metoda nedošla daleko od počáteční aproximace, tedy že na volbě learning-rate opravdu záleží.

Rosenbrockova funkce, první kolo

Výsledky testování na této funkci jsou k vidění v Tabulce 4.2. Připomínáme, že v tomto kole volíme počáteční aproximaci $x_0 = (2, 1.5)^T$.

Pořadí	Metoda	η, learning rate	Počet kroků
1.	NM	—	349
2.	Adam	0,05	5551
3.	GDM	0,005	9028
4.	NAG	0,005	9266
5.	AdaGrad	2	26921
10.	GD	0,001	45119

Tabulka 4.2: Výsledky testování optimalizačních metod na Rosenbrockově funkci s počáteční aproximací $x_0 = (2, 1.5)^T$.

Jak jsme již dříve zmínili, Nelder-Meadova simplexová metoda na tomto protipříkladu funguje daleko lépe, než spádové metody. Divoké chování gradientu totiž Nelder-Meadovu metodu narozdíl od spádových metod přímo neovlivní.

Rosenbrockova funkce, druhé kolo

Další výsledky získané na Rosenbrockově funkci vidíme v Tabulce 4.3. Připomínáme, že v tomto kole volíme počáteční aproximaci $x_0 = (0, 0)^T$.

Pořadí	Metoda	η, learning rate	Počet kroků
1.	NM	—	103
2.	Adam	0,1	2657
3.	GDM	0,01	4324
4.	NAG	0,01	4346
8.	AdaGrad	0,05	19085
16.	GD	0,001	45132

Tabulka 4.3: Výsledky testování optimalizačních metod na Rosenbrockově funkci s počáteční aproximací $x_0 = (0, 0)^T$.

Opět můžeme pozorovat úspěch Nelder-Meadovy simplexové metody. Jak ale uvidíme v následující kapitole, zde úspěšná Nelder-Meadova metoda není zase tak praktická pro použití k trénování neuronové sítě.

4.3 Testování na úloze klasifikace ropných vrtů

Nyní konečně přejdeme k testování na neuronových sítích. Zatím však zůstaneme v malém měřítku a budeme řešit problém klasifikace ropných vrtů pomocí malé neuronové sítě. Pro příklad jsme brali inspiraci v druhé kapitole článku [4].

4.3.1 Představení úlohy

Jsme na území, v jehož podzemí se nachází ropa. Nepřekvapí nás tedy, že na tomto území najdeme spoustu ropných vrtů. Našim úkolem je vybrat pozici pro nový ropný vrt tak, aby měl „dobré výsledky“, tedy vybrat pozici, na které se v podzemí nachází „hodně“ ropy.

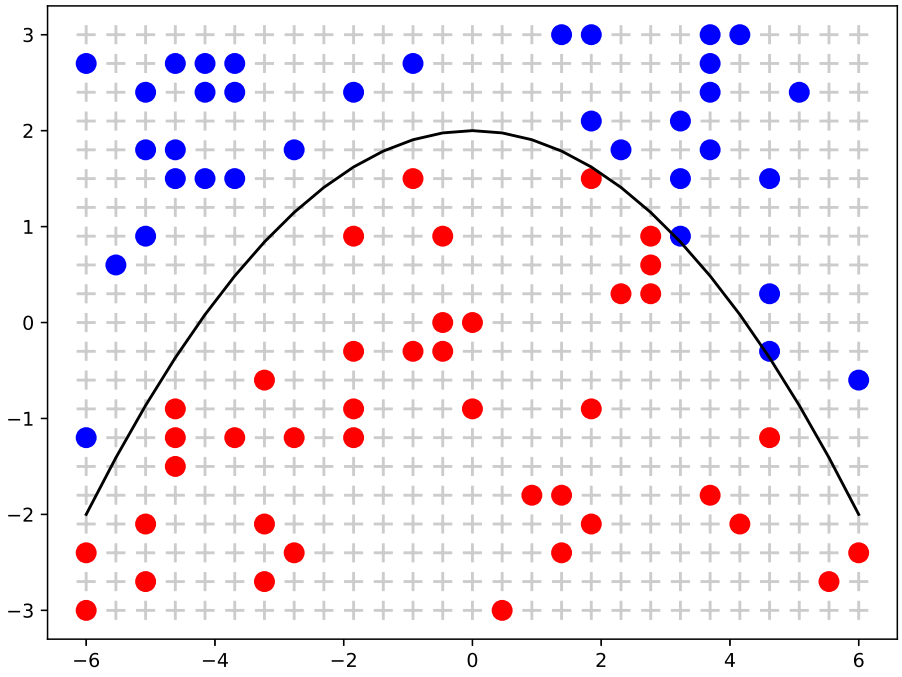
Území, na kterém se nacházíme, je čtvercová mřížka a *pozicí* na tomto území rozumíme jeden konkrétní mřížový bod. Na každé pozici se přitom nachází buď *hodně*, nebo *málo* ropy; je-li na dané pozici postavený ropný vrt, pak tento ropný vrt má *dobré výsledky*, nebo má *špatné výsledky* právě podle toho, zda se na této pozici nachází hodně, nebo málo ropy.

Při rozhodování, kde postavit nový ropný vrt, budeme používat informace z ostatních ropných vrtů - tedy zda již existující ropný vrt na nějaké pozici má dobré výsledky, nebo má špatné výsledky. Předpokládáme pak, že jednotlivá území, kde se nachází hodně, resp. málo ropy, jsou souvislé oblasti oddělené hladkou křivkou.

Vzhledem k tématu této práce, se rozhodneme problém řešit pomocí neuronových sítí. Na datech z ropných vrtů naučíme neuronovou síť provádět následující. Síť na vstupu dostane jistou pozici na ropném poli (tj. souřadnice této pozice, tj. dvě čísla) a na výstupu nám dá svůj tip, zda se na dané pozici nachází hodně, nebo málo ropy. Síť můžeme tento úkon učit na datech z ropných vrtů - pro danou

pozici totiž víme, jestli se pod ní nachází hodně, nebo málo ropy - víme totiž, zda má ropný vrt dobré, nebo špatné výsledky.

Nyní popíšeme zadání úlohy korektněji. Výše zmíněným územím bude množina $\Omega = [-6, 6] \times [-3, 3] \subseteq \mathbb{R}^2$, pozicemi na tomto území bude $21 \cdot 27 = 567$ bodů v Ω, ekvidistantně rozmístěných v obou složkách, celkem ve 21 řádcích a 27 sloupcích. Všechny pozice, které se budou nacházet nad křivkou $y = 2 - x^2/9$, budou obsahovat málo ropy a všechny pozice, které se budou nacházet pod touto křivkou, budou obsahovat hodně ropy. Právě 75 z celkem 567 pozic budou obsazeny nějakým ropným vrtem. Popsanou situaci ukazuje Obrázek 4.4.



Obrázek 4.4: Ropné pole; černá křivka znázorňuje hranici mezi oblastmi, kde je hodně, resp. málo ropy; znaménka + znázorňují pozice v poli; puntíky znázorňují jednotlivé ropné vrty; modré vrty mají špatné výsledky (oblast nad křivkou má málo ropy), červené vrty mají dobré výsledky (oblast pod křivkou má hodně ropy).

Neuronová síť, kterou budeme trénovat, **uvidí pouze již existující ropné vrty, a jaké výsledky tyto vrty mají**; tedy neuronová síť během tréninku (ani po něm) **nemá informaci o křivce rozdělující jednotlivé oblasti**. Jde nám tedy o to, jak dobře bude neuronová síť zobecňovat informaci získanou z již existujících ropných vrtů.

4.3.2 Způsob testování

Zvolme strukturu sítě $(2, 8, 2)$; dvěma vstupním neuronům předáme souřadnice pozice, která nás zajímá (tuto pozici přeškálujeme do intervalu $(0, 1)$); odpověď neuronové sítě budeme interpretovat jako pořadí neuronu ve výstupní vrstvě, který bude mít pro daný vstup větší aktivitu. Bude-li aktivnější první neuron výstupní vrstvy, budeme tuto odpověď chápat tak, že na pozici, kterou jsme neuro-

nové síti předali, je hodně ropy, a naopak situaci vyhodnotíme, bude-li aktivnější druhý neuron výstupní vrstvy.

Tréninkovým datasetem bude posloupnost dvojic - souřadnice ropného vrtu a očekávaná odpověď (výše zmíněná interpretace toho, zda má tento vrt dobré výsledky). Jelikož má tento dataset pouze 75 prvků, není problém používat klasické verze optimalizačních metod a není třeba pracovat se stochastickými variantami; s těmi se potkáme v Sekci 4.4.

Opět zvolíme doporučené hodnoty hyperparametrů (viz příslušné definice metod, Kapitola 3), kromě learning-rate, který budeme volit

$$\eta \in \{10, 5, 2, 1, 0.5, 0.1, 0.05, 0.01, 0.005, 0.001\}$$

jako v Sekci 4.2.

Každou metodou budeme trénovat pro každou volbu learning-rate do té doby, dokud hodnota ztrátové funkce na celém tréninkovém datasetu nebude menší, než jistá tolerance. Metody srovnáme podle počtu kroků potřebných pro to, aby dostaly celkovou ztrátu pod tuto toleranci.

4.3.3 Výsledky testování

V Tabulce 4.4 uvádíme naměřená data při trénování neuronové sítě pomocí spádových metod. Pořadí uvedené v prvním sloupci tabulky má stejný význam jako v Sekci 4.2.

Pořadí	**Metoda**	**η, learning rate**	**Počet kroků**
1.	AdaGrad	0,5	21
2.	NAG	5	25
5.	GDM	2	44
12.	GD	1	181
14.	Adam	5	248

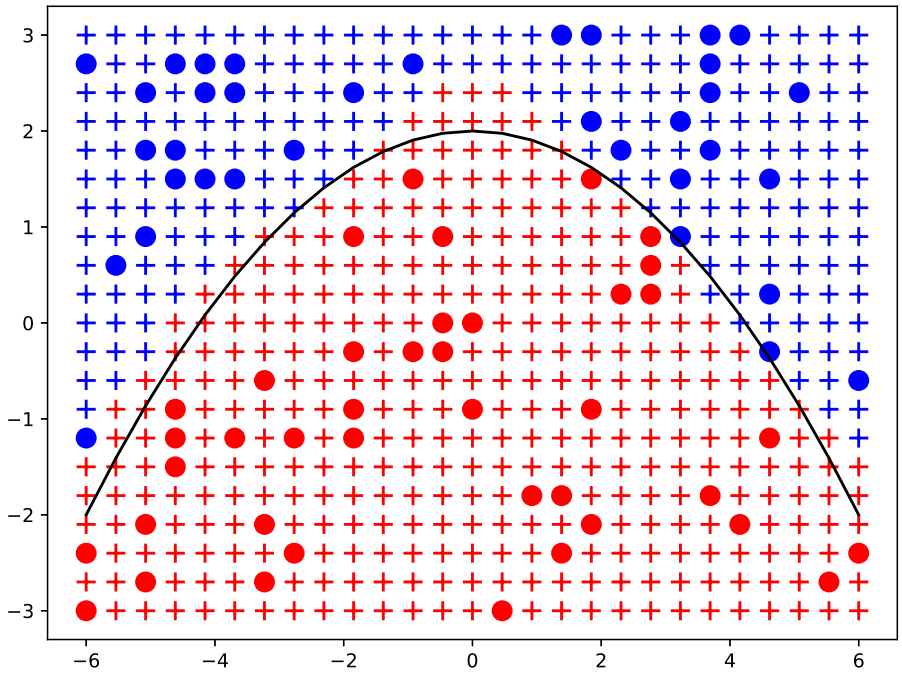
Tabulka 4.4: Výsledky testování spádových metod na neuronové síti se strukturou $(2, 8, 2)$.

Všechny metody se úspěšně dostaly pod toleranci, přičemž nejúspěšnější byly v tomto případě metoda AdaGrad a Nesterovova metoda, které zvládly stlačit celkovou ztrátu pod danou toleranci za méně než třicet kroků.

Odhad oblasti, ve které se nachází hodně ropy můžeme nyní získat tak, že vytrénovanou neuronovou síť necháme postupně zpracovat vstupy odpovídající všem 567 pozicím. Takto získaný odhad neuronové sítě, kterou jsme vytrénovali metodou AdaGrad (první pozice v tabulce), je znázorněn na Obrázku 4.5.

Jak jsme již dříve slíbili, na tomto příkladu jsme otestovali také Nelder-Meadovu simplexovou metodu. Vzhledem k počtu parametrů (celkem 42 vah a prahů) se nejedná o trojúhelník pohybující se v $\mathbb{R}^2$, ale o simplex složený z celkem 43 vrcholů pohybující se v $\mathbb{R}^{42}$.

Pod stejnou toleranci, kterou jsme nastavili spádovým metodám, se Nelder-Meadova simplexová metoda dostala během 2135 kroků, a tak by v Tabulce 4.4 zaujala poslední pozici. Ačkoliv Nelder-Meadova simplexová metoda dosahovala



Obrázek 4.5: Odhad neuronové sítě, kterou jsme vytrénovali metodou AdaGrad, viz legenda pod Obrázkem 4.4. Znaménka + jsou zbarvená podle odhadu neuronové sítě, zda se na dané pozici nachází málo ropy (modrá barva), nebo hodně ropy (červená barva).

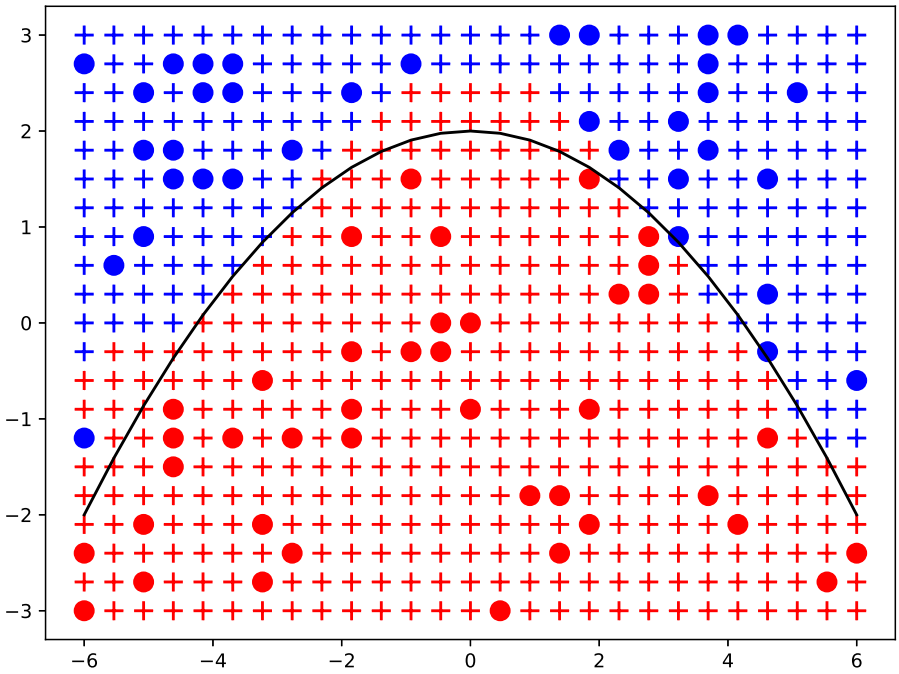
na akademických funkcích výborných výsledků, zde se jí takového úspěchu nedostává.

Výpočetní čas byl navíc u Nelder-Meadovy simplexové metody mnohonásobně větší, nicméně tento fakt může být závislý na implementaci. Ve velké neuronové síti, kterou budeme trénovat v Kapitole 4.4, Nelder-Meadova simplexová metoda nedokončila (v rozumném čase) ani jednu epochu, a proto ji ani nebudeme dále testovat.

Odhad, kde se nachází hodně ropy, který dává neuronová síť trénovaná Nelder-Meadovou simplexovou metodou, můžeme vidět na Obrázku 4.6.

4.4 Testování na úloze rozpoznávání rukou psaných číslic

V této sekci budeme testovat stochastické verze metod, neboť již budeme pracovat s větší neuronovou sítí, pomocí které budeme řešit problém rozpoznávání rukou psaných číslic. Tato úloha je dnes považovaná za základní kanonickou testovací úlohu, na podobném problému testují neuronové sítě např. i autoři v [2] nebo v [3].



Obrázek 4.6: Odhad neuronové sítě, kterou jsme vytrénovali Nelder-Meadovou simplexovou metodou, viz legenda pod Obrázkem 4.4. Znaménka + jsou zbarvená podle odhadu neuronové sítě, zda se na dané pozici nachází málo ropy (modrá barva), nebo hodně ropy (červená barva).

4.4.1 Představení úlohy

Každý člověk má odlišný rukopis, a tak konstrukce algoritmu, který dokáže klasifikovat obrázek rukou psané číslice, není triviální. Velmi efektivním nástrojem pro řešení tohoto problému jsou právě neuronové sítě; to ale jen díky tomu, že máme k dispozici velký tréninkový dataset obsahující rukou psané číslice společně s příslušnými správnými odpověďmi. Mluvíme zde o MNIST databázi rukou psaných číslic (viz [6]).

Tato databáze obsahuje celkem 70 000 obrázků rukou psaných číslic (viz Obrázek 4.7) společně s označením, jaká číslice se na obrázku nachází. Číslice, které v databázi najdeme přitom byly psány pěti sty lidmi, tedy databáze obsahuje rozmanitou škálu různých rukopisů.

Obrázek 4.7: Ukázka pěti číslic z celkem 70 000 podobných, které lze nalézt v MNIST databázi rukou psaných číslic (viz [6]). Ke každému z uvedených obrázků databáze poskytuje i správnou odpověď, tedy jaké číslo se nachází na daném obrázku.

Standardně se tato sada obrázků dělí na 60 000 tréninkových vzorků a 10 000 testovacích vzorků; příslušný model se pak učí pouze na tréninkových obrázcích a testuje se na testovacích obrázcích, které modelu do doby testování nebyly k dispozici. Můžeme se takto vyhnout *přeučení* (angl. *overfitting*), tj. jevu, kdy se neuronová síť příliš přizpůsobuje tréninkovému datasetu, ale již se neučí rozpoznávat číslice - poznáme, že na testovacím datasetu úspěšnost klasifikace neroste.

V následujícím budeme taktéž uvažovat rozdělení na tréninkové a testovací obrázky, nicméně ze skupiny tréninkových obrázků ještě vybereme 10 000 obrázků, které budeme nazývat *validační*. Na validačních obrázcích budeme síť trénovat (pořád patří mezi 60 000 tréninkových vzorků), ale také na nich budeme síť testovat. Ve výsledku tedy dostaneme dvě čísla charakterizující, jak dobře si síť vedla - úspěšnost na validačních datech (na takových, na kterých síť rovněž trénovala) a úspěšnost na testovacích datech (na takových, které síť během tréninku neviděla). Právě toto rozdělení nám také může pomoci s detekcí přeučení; stoupá-li totiž úspěšnost na validačních datech, ale na testovacích nikoliv, síť se už neučí rozpoznávat rukou psané číslice, ale memoruje tréninkový dataset nazpaměť.

4.4.2 Způsob testování

Každý obrázek z MNIST databáze je černobílý a má rozměry 28×28 pixelů. Přirozeně je pak reprezentován polem délky $784 = 28^2$, které obsahuje příslušné hodnoty na stupni šedi (angl. grayscale value) pro každý pixel obrázku. Přirozeně se tedy nabízí volit strukturu neuronové sítě takovou, aby vstupní vrstva obsahovala 784 neuronů - jeden neuron pro každý pixel obrázku. Co se správných odpovědí týče, v úvahu přichází pouze hodnoty z množiny $\{0, 1, \ldots, 9\}$, a tak mějme pro každou z těchto možností právě jeden neuron ve výstupní vrstvě sítě. Odvodili jsme volbu struktury sítě ve tvaru $(784, \ldots, 10)$.

Zbývá zvolit počet a jednotlivé velikosti skrytých vrstev. Více skrytých vrstev a větší počet neuronů může znamenat lepší výsledky, ale nese s sebou také větší výpočetní nároky. Spokojíme se proto pouze s jednou skrytou vrstvou, která bude obsahovat 30 neuronů. Tato volba je zlatým středem mezi nízkými výpočetními nároky a robustností sítě; se stejnou volbou struktury sítě se lze setkat i v první kapitole [2].

Na této úloze budeme opět testovat ve dvou kolech.

Jak už jsme dříve zmínili, vzhledem k velikosti datasetu budeme používat stochastické verze optimalizačních metod. V prvním kole tedy otestujeme, zda je tato modifikace opravdu potřeba; zvolíme jednu konkrétní metodu - metodu největšího spádu - a budeme neuronovou síť trénovat pro různé velikosti mini-batchí. Pro každou velikost mini-batche z množiny

$$\{1, 2, 4, 8, 16, 32, 64, 128, 512, 2048, 16384, 60000\}$$

budeme neuronovou síť trénovat po dobu třiceti epoch (tedy neuronová síť uvidí kompletní dataset celkem třicetkrát, viz Kapitola 3.2) a nakonec porovnáme úspěšnost neuronové sítě na testovacích datech. Přitom trénink provedeme pro každou velikost mini-batche třikrát a pro porovnání použijeme nejlepší výsledek.

V druhém kole již budeme pracovat s pevnou velikostí mini-batche a síť budeme trénovat postupně všemi optimalizačními metodami (s výjimkou Nelder-Meadovy simplexové metody), tedy jejich stochastickými verzemi pro pevnou

velikost mini-batche. Každou optimalizační metodu budeme navíc uvažovat pro několik sad hyperparametrů. Pro porovnání metod pak opět použijeme úspěšnost neuronové sítě na testovacích datech po trénování danou optimalizační metodou po pevnou dobu třiceti epoch. Přitom pro porovnání opět použijeme nejlepší výsledek ze tří testování.

4.4.3 Výsledky testování

Následují výsledky testování optimalizačních metod pro trénování neuronové sítě se strukturou $(784, 30, 10)$ na MNIST datasetu.

První kolo - ideální velikost mini-batchí pro SGD

V Tabulce 4.5 můžeme vidět výsledky testování stochastichých metod. Tabulka ukazuje nejlepší (vzhledem k úspěšnosti sítě klasifikovat obrázky z testovací sady) ze tří výsledků trénování sítě stochastickou verzí metody největšího spádu (SGD) pro různé velikosti mini-batchí, které jsou uvedeny v prvním sloupci tabulky. Příslušná procentuální úspěšnost na testovacích datech je pak uvedena v druhém sloupci. Dále v tabulce uvádíme čas v sekundách, za který metoda stihla všech třicet epoch tréninku. Pro představu, jak moc daná velikost mini-batche sníží celkový počet kroků, uvádíme v posledním sloupci také přibližný počet kroků vykonaný metodou během třiceti epoch tréninku.

Velikost mini-batche	**Úspěšnost na test. datech [%]**	**Celkový čas tréninku [s]**	**Přibližný počet kroků**
1	95,61	232,01	1 800 000
2	95,28	226,96	900 000
4	95,19	207,42	450 000
8	95,46	194,69	225 000
16	95,34	190,97	112 500
32	95,19	186,69	56 250
64	95,24	184,81	28 125
128	48,99	184,15	14 062
512	11,99	182,38	3 515
2 048	11,35	184,83	878
16 384	10,02	205,00	109
60 000	10,32	205,60	30

Tabulka 4.5: Výsledky testování stochastické verze metody největšího spádu pro různé velikosti mini-batchí. Uvedeny jsou nejlepší dosažené výsledky ze tří testování.

Jak můžeme vidět v posledním řádku tabulky, počítat gradient přesně se nevyplatilo; náročnost tohoto výpočtu jde ruku v ruce s faktem, že metoda za celých třicet epoch vykoná pouze třicet kroků; přitom výpočetní čas je srovnatelný se stochastickými verzemi, které si s problémem poradily daleko lépe.

K našemu překvapení dobře fungoval opačný extrém - mini-batch o velikosti 1; výsledek ilustruje úspěšnost stochastického vylepšení - méně přesné aproximace gradientů jsou kompenzovány větším počtem kroků.

Rovněž si ale můžeme všimnout, že celkový čas tréninku pro velikost mini-batche 1 je ale nejvyšší ze všech. Pro další testování zvolme velikost mini-batche 16; celkový čas tréninku se oproti nejhoršímu času sníží cca o čtyřicet sekund (jelikož budeme testovat hodně metod pro hodně sad hyperparametrů, je tento rozdíl zásadní) a přitom neztrácíme mnoho úspěšnosti při klasifikaci testovacích dat natrénovanou sítí.

Poznámka. Než se pustíme do druhého kola, dovolíme si ještě malou odbočku. Výsledky z Tabulky 4.5 by již mohly svědčit o tom, že jsme schopní celkem dobře najít/aproximovat nějaké lokální minimum ztrátové funkce sítě. Chápejme dále ztrátovou funkci sítě pouze jako funkci parametrů sítě. Znalost přibližné polohy lokálního minima nám nyní umožňuje alespoň trochu nahlédnout, jak terén ztrátové funkce sítě kolem tohoto lokálního minima vypadá.

Jak jsme již zmínili za Definicí 9, dimenze parametrického prostoru sítě (definiční obor ztrátové funkce) je v našem případě (sítě se strukturou $(784, 30, 10)$) nemalá. Rozhodně není dost malá na to, abychom si dokázali představit graf ztrátové funkce. Nicméně můžeme se podívat alespoň na řez tímto grafem.

Zafixujeme-li všechny až na dva parametry sítě, můžeme ztrátovou funkci sítě chápat jako funkci těchto dvou parametrů. Graf této funkce již dokážeme vykreslit. Výsledek takového činění můžeme vidět na Obrázku 4.8 - ztrátovou funkci jsme vyhodnotili v jistých síťových bodech okolo lokálního minima ztrátové funkce a interpolovali.

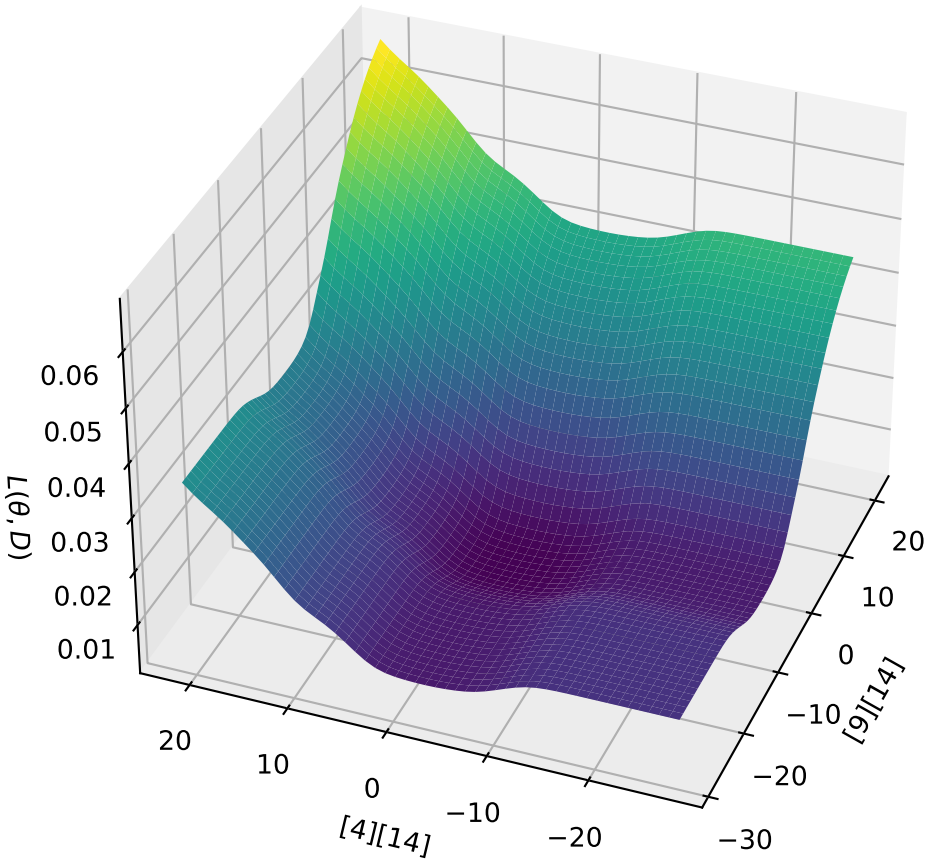
Druhé kolo - porovnání stochastických metod

Každou z metod jsme testovali pro několik různých sad hyperparametrů, neboť apriori není jasné, pro kterou sadu by měla metoda fungovat nejlépe. V Kapitole 3.1 jsme při představení některých z metod uváděli doporučené nastavení hyperparametrů; tyto doporučené sady hyperparametrů jsme rovněž zařadili do testování.

Testování na všech sadách hyperparametrů proběhlo celkem třikrát, přičemž nakonec jsme pro každou sadu vybrali nejlepší ze tří výsledků - tento (nejlepší) výsledek jsme pak použili k porovnání. Tabulka 4.6 pak uvádí pro každou stochastickou spádovou metodu tu sadu hyperparametrů, se kterou byla tréninková metoda nejúspěšnější. Zároveň se v Tabulce 4.6 dočteme procentuální úspěšnost příslušné vytrénované sítě na testovacích datech a také celkový čas, který metoda potřebovala pro trénování po dobu třiceti epoch.

Metoda	η	$m\,(\approx \beta_1)$	β_2	ε	**Úspěšnost na test. datech** [%]	**Celkový čas tréninku** [s]
SGDM	0,5	0,7	—	—	95,57	192,54
NAG	0,5	0,7	—	—	95,34	197,35
SGD	0,3	—	—	—	95,23	181,63
Adam	1,0	0,7	0,9	0,1	95,14	211,72
AdaGrad	0,1	—	—	0,1	94,67	195,61

Tabulka 4.6: Nejlepší dosažené výsledky testování stochastických verzí spádových metod pro velikost mini-batche 16, pro různé sady hyperparametrů. Uvedeny jsou nejlepší dosažené výsledky ze tří testování.



Obrázek 4.8: Řez grafem ztrátové funkce. Téměř všechny parametry sítě jsou voleny pevně „blízko“ lokálního minima ztrátové funkce sítě. V grafu vidíme, jak se ztrátová funkce mění, když posouváme váhu mezi čtrnáctým neuronem předposlední vrstvy sítě a čtvrtým neuronem výstupní vrstvy sítě (osa „[4][14]“) a váhu mezi čtrnáctým neuronem předposlední vrstvy sítě a devátým neuronem výstupní vrstvy sítě (osa „[9][14]“).

Můžeme vidět, že všechny metody síť vytrénovaly na přibližně stejnou úroveň - na testovacím datasetu neuronová síť dokázala správně klasifikovat přibližně 95 % obrázků. Pro představu, dnes se různým typům neuronových sítí na MNIST testovacím datasetu daří dosáhnout úspěšnosti i přes 99.5 % (viz např. [17]).

V Tabulce 4.6 si lze všimnout, že se složitostí metod přichází i mírný nárůst celkové doby tréninku; nejjednodušší metoda - SGD - trénink dokončila za 181.63 s a všechny ostatní metody, které SGD rozšiřují, potřebovaly času více - nejsložitější Adam potřeboval přibližně o půl minuty více, než SGD.

Dále si můžeme všimnout, že nejlepší výsledek v Tabulce 4.5 je lepší než nejlepší výsledek v Tabulce 4.6. Pozorujeme opět, že i na volbě velikosti mini-batche záleží. Další odlišné výsledky bychom mohli obdržet testováním ostatních stochastických metod pro jiné velikosti mini-batchí.

Nutno podotknout, že volba hyperparametrů nemusí být triviální proces. Úspěšnost metody může při stejné sadě hyperparametrů ovlivnit také struktura neuronové sítě (při jiné struktuře sítě má ztrátová funkce jiný tvar), nebo velikost mini-batche (např. podle [18] může mít dokonce snížení hodnoty η za jistých okolností stejný efekt jako zvýšení velikosti mini-batchí). Metodám s vícero hyperparametry - jako je například Adam - pak může být úkol naladit správné hyperparametry výzva. Větší počet hyperparametrů se tak stává dvousečnou zbraní - máme sice prostor přizpůsobit si vlastnosti metody, ale jedná se o netriviální práci navíc.

Závěr

Stěžejními třemi tématy práce byly popis neuronových sítí a s nimi souvisejících pojmů, představení numerických optimalizačních algoritmů, které se pro trénování neuronových sítí používají, a testování těchto algoritmů na různých příkladech.

V Kapitole 1 jsme uvedli laický popis neuronových sítí, jejich jednotlivých komponent, vysvětlili jsme, jak sítě fungují, a motivovali tak Kapitolu 2, kde jsme neuronové sítě popsali rigorózně matematicky. V Kapitole 2.1 jsme zavedli pojmy potřebné pro porozumění dané problematice, zavedli ztrátovou funkci sítě, a tím převedli problém trénování neuronové sítě na problém optimalizace; v Kapitole 2.2 jsme odvodili algoritmus zpětného šíření chyby, který umožňuje efektivně počítat gradient ztrátové funkce sítě potřebný pro optimalizaci spádovými metodami.

Právě metody spádového typu jsou dnes běžně používané k trénování neuronových sítí, některé jsme si představili v Kapitole 3.1. Odvodili jsme metodu největšího spádu, která je základním kamenem, na kterém staví ostatní představené metody, s výjimkou Nelder-Meadovy simplexové metody, kterou jsme uvedli spíše jako zajímavost. V Sekci 3.2 jsme dále představili stochastické verze spádových algoritmů, které v trénování neuronových sítí hrají stěžejní roli; tyto metody používají pouze aproximace gradientu ztrátové funkce, a tak urychlují celý proces optimalizace.

Na třech různých příkladech jsme pak metody testovali v Kapitole 4. Nejprve jsme v Kapitole 4.2 metody otestovali na jednoduchých akademických příkladech. Spuštěním metod na hyperbolickém paraboloidu jsme ilustrovali problém spádových metod, tj. uvíznutí v sedlovém bodě; při optimalizaci na Rosenbrockově funkci jsme pak viděli, že metody spádových směrů nemusí být vhodné pro každý příklad. Avšak v dalších dvou příkladech jsme se přesvědčili, že spádové metody jsou velmi vhodné pro numerickou optimalizaci v neuronových sítích. Na problému klasifikace ropných vrtů v Kapitole 4.3 jsme v neuronové síti vyzkoušeli i Nelder-Meadovu metodu, abychom ilustrovali, že trénování malé sítě se obejde i bez znalosti gradientu, a že se nejedná o nic jiného, než o matematickou optimalizaci. S větší neuronovou sítí jsme pracovali v Kapitole 4.4, kde jsme porovnali stochastické verze spádových metod; taky jsme zde udělali malou odbočku a ukázali si řez grafem ztrátové funkce sítě.

V práci by šlo dále pokračovat, popřípadě na ni navázat, několika různými směry. V průběhu práce jsme narazili na otázky, které jsme zodpověděli jen částečně. Mohli bychom dále zkoumat jevy, jako jsou uvíznutí spádových metod v sedlových bodech a jak tomuto jevu předcházet; divoké chování spádových metod na funkcích podobných Rosenbrockově protipříkladu; zajímat se, jak spolu souvisí velikost mini-batchí, struktura neuronové sítě a hodnoty hyperparametrů stochastických spádových metod; jak efektivně ladit hyperparametry některých metod; jak může trénink neuronové sítě ovlivnit volba ztrátové funkce atd. Rovněž by se dalo navázat představením dalších optimalizačních metod (i jiných než spádových!), které se dnes k tréninku neuronových sítí používají.

Zajímají-li čtenáře neuronové sítě více či méně, věříme, že pozoruhodných matematických problémů s nimi spojených se dá najít spousta, a to z různých odvětví matematiky, nejen z pohledu numerické analýzy.

Seznam použité literatury

[1] Juergen Schmidhuber. Annotated history of modern AI and deep learning. *arXiv preprint arXiv:2212.11279*, 2022.

[2] Michael A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. `http://neuralnetworksanddeeplearning.com/`, [cit. 2024-03-24].

[3] Grant Sanderson (3Blue1Brown). Neural Networks lessons & YouTube playlist, 2017. `https://www.3blue1brown.com/topics/neural-networks`, [cit. 2024-03-28].

[4] Catherine F Higham and Desmond J Higham. Deep learning: An introduction for applied mathematicians. *Siam review*, 61(4):860–891, 2019.

[5] Shiliang Sun, Zehui Cao, Han Zhu, and Jing Zhao. A survey of optimization methods from a machine learning perspective. *IEEE transactions on cybernetics*, 50(8):3668–3681, 2019.

[6] Yann LeCun, Chris Burges, and Corinna Cortes. The MNIST database. `http://yann.lecun.com/exdb/mnist/`, [cit. 2024-03-24].

[7] Miloslav Feistauer and Václav Kučera. *Základy numerické matematiky*. Matfyzpress, Praha, 2014.

[8] Benjamin Recht and Stephen J. Wright. *Optimization for Modern Data Analysis*. 2019. Preprint available at `http://eecs.berkeley.edu/~brecht/opt4mlbook`, [cit. 2024-03-24].

[9] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[10] John C. Duchi, Elad Hazan, and Yoram Singer. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.*, 12:2121–2159, 2011.

[11] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[12] Tijmen Tieleman and Geoffrey Hinton. Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning, 2012.

[13] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[14] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, 01 1965.

[15] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM review*, 60(2):223–311, 2018.

[16] Pierre Carbonnelle. PYPL Popularity of Programming Language, 2023. `https://pypl.github.io/PYPL.html`, [cit. 2024-03-24].

[17] Dan Ciregan, Ueli Meier, and Jürgen Schmidhuber. Multi-column deep neural networks for image classification. In *2012 IEEE conference on computer vision and pattern recognition*, pages 3642–3649. IEEE, 2012.

[18] Samuel L Smith, Pieter-Jan Kindermans, Chris Ying, and Quoc V Le. Don't decay the learning rate, increase the batch size. *arXiv preprint arXiv:1711.00489*, 2017.

Seznam obrázků

Seznam tabulek