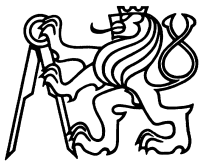
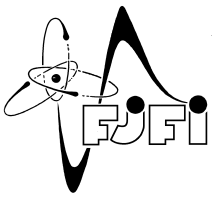


České vysoké učení technické v Praze
Fakulta jaderná a fyzikálně inženýrská



Robustní strojové učení a adversariální vzorky

Robust machine learning and adversarial examples

Bakalářská práce

Autor: **Pavel Jakš**

Vedoucí práce: **Mgr. Lukáš Adam, Ph.D.**

Akademický rok: 2021/2022

- Zadání práce -

- Zadání práce (zadní strana) -

Poděkování:
Chtěl bych zde poděkovat především svému školiteli - panu doktoru Adamovi - za pečlivost, ochotu, vstřícnost a odborné i lidské zázemí při vedení mé bakalářské práce.

Čestné prohlášení:
Prohlašuji, že jsem tuto práci vypracoval samostatně a uvedl jsem všechnu použitou literaturu.

V Praze dne 7. července 2022 Pavel Jakš

Název práce:

Robustní strojové učení a adversariální vzorky

Autor: Pavel Jakš

Obor: Matematická informatika

Druh práce: Bakalářská práce

Vedoucí práce: Mgr. Lukáš Adam, Ph.D., Katedra počítačů, Fakulta elektrotechnická, České vysoké učení technické v Praze, Karlovo náměstí 13, 121 35, Praha 2

Abstrakt: Abstrakt max. na 10 řádků.

Klíčová slova: klíčová slova (nebo výrazy) seřazená podle abecedy a oddělená čárkou

Title:

Robust machine learning and adversarial examples

Author: Pavel Jakš

Abstract: Max. 10 lines of English abstract text.

Key words: keywords in alphabetical order separated by commas

Obsah

Úvod

Pojem neuronové sítě představuje výpočetní jednotku, která svou univerzálností nachází uplatnění v mnoha disciplínách.

Kapitola 1

Neuronové sítě

Princip fungování neuronové sítě spočívá v poskládání celku z dílčích výpočetních jednotek - umělých neuronů. Takovýto neuron je standardně funkcí více proměnných, jehož výstup je proměnná jediná. Typickým modelem umělého neuronu je funkce $f : \mathbb{R}^n \to \mathbb{R}$ definovaná předpisem

$$f(a_1, ..., a_n) = \sigma(\sum_{i=1}^{n} w_i a_i + b), \tag{1.1}$$

kde n je počet vstupujících proměnných, w_i jsou tzv. váhy (w z anglického slova weight), b je práh (b z anglického slova bias), σ označuje tzv. aktivační funkci.

Roli vstupujících proměnných mohou hrát např. hodnoty RGB pixelů barevných obrázků, je-li aplikací klasifikace obrázků, nebo výstupy jiných neuronů. Pod pojmem váha se skrývá míra ovlivnění výstupu neuronu daným vstupem. Je-li váha u nějakého vstupu vysoká, pak je výstup citlivější na daný vstup. Práh pro změnu určuje posunutí citlivosti neuronu na všechny vstupy jako celku.

Poslední, avšak velmi důležitou charakteristikou tohoto modelu neuronu je aktivační funkce. Za aktivační funkci lze vzít libovolnou funkci $\sigma : \mathbb{R} \to \mathbb{R}$, existuje však základní sada:

- Sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$
- ReLU: $\sigma(z) = max(0, z)$
- LeakyReLU: $\sigma(z) = max(0, z) + \alpha * min(z, 0)$, kde $\alpha \in \mathbb{R}^+$
- Tanh: $\sigma(z) = tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$

Tyto funkce lze doplnit o jejich mírné modifikace. Moderní doporučenou praxí je užívat ReLU jako aktivační funkci a (1.1) jako model neuronu dle [1].

1.1 Hluboká dopředná neuronová síť

Je-li pojem umělého neuronu objasněn, lze se přesunout k jeho užití v neuronových sítích. Základní myšlenkou těchto sítí je vhodné poskládání umělých neuronů do vrstev, které dohromady tvoří síť neuronů. Taková vrstva je potom trojího druhu - vstupní, výstupní a skrytá. *Vstupní vrstva* je množina umělých neuronů, které mají za vstup výstupy problému, jehož je neuronová síť řešením. Za vstup si lze představit matici černobílých pixelů, které představují obrázek číslice, kterou je cíl klasifikovat. *Výstupní* vrstva sestává z neuronů, které mají za vstup výstupy neuronů předchozí vrstvy. Výstupem této vrstvy pak bude řešení daného problému - například klasifikace číslice. Posledním druhem vrstvy je *vrstva*

skrytá. Takováto vrstva má za vstupy výstupy vrstvy předcházející a její výstupy slouží jako vstupy pro vrstvu nadcházející. Má-li neuronová síť tuto architekturu, hovoří se o *dopředné neuronové síti*. Má-li navíc alespoň jednu skrytou vrstvu, lze mluvit o *hluboké dopředné neuronové síti*.

Se znalostí pojmu vrstvy neuronů lze přistoupit k poznámce o tzv. *softmax funkci*. Jedná se o vektorovou funkci $s : \mathbb{R}^m \to \mathbb{R}^m$, kde

$$s(a_1, ..., a_m)_i = \frac{e^{a_i}}{\sum_{j=1}^{m} e^{a_j}},$$

kde $i \in \hat{m}$. Její užití je nasnadě: Výstup této funkce lze totiž interpretovat jako diskrétní pravděpodobnostní distribuci, a proto ji lze užít jako aktivační funkci výstupní vrstvy, je-li cílem dané neuronové sítě klasifikace vstupu do kategorií.

Další poznámka se bude věnovat zjednodušení zápisu akce vrstvy na vstup. Podle modelu neuronu v (1.1) se akce jednodnoho neuronu na vstup sestává z násobení, následného sčítání, přičtení prahu a aplikací aktivační funkce. Tato procedura nastává pro každý neuron ve vrstvě. Tak lze sestavit z jednotlivých vah $w_i^{(j)}$ (i-tá váha j-tého neuronu ve vrstvě) matici $\mathbb{A}$, jejímiž prvky jsou právě ony váhy $(\mathbb{A})_{j,i} = w_i^{(j)}$, z prahů pak vektor b, jehož j-tá složka je rovna prahu j-tého neuronu. Dále zaveďme vektorovou funkci $s : \mathbb{R}^m \to \mathbb{R}^m$ - ať už jako výše zmíněnou softmax funkci, nebo jako po složkách aplikovanou libovolnou aktivační funkci σ ve smyslu $s(a_1, ...a_m)_i = \sigma(a_i)$ pro $i \in \hat{m}$. Pak lze psát, že aplikace vrstvy neuronů je zobrazení $\phi : \mathbb{R}^n \to \mathbb{R}^m$ působící na vektor a následovně:

$$\phi(a) = s(\mathbb{A}a + b) \tag{1.2}$$

Tedy stěžejní operací se stává maticové násobení, respektive násebení vektoru maticí zprava.

Při tomto si lze povšimnout, že takováto neuronová síť má řadu parametrů, o kterých není jasné jak je správně nastavit. Některé parametry (například váhy a prahy) se nastavují během učení neuronové sítě, čemuž je věnována samostatná kapitola. Potom tu jsou parametry, jejichž charakter je poněkud odlišný. Jedná se o ty parametry, které zůstávají během života neuronové sítě netknuté. Jako příklad lze uvést počet neuronů ve skryté vrstvě, který se promítne v rozměrech matice vah či dimenzionalitě výstupu vrstvy. Takovýmto prametrům je přisuzován název hyper-parametry.

1.2 Konvoluční sítě

Konvoluční sítě nebo též *konvoluční neuronové sítě* přinášejí svou architekturou nové možnosti zpracování dat se specifickou strukturou, do které patří například časové řady, obrázky nebo videa. Středobodem konvolučních sítí je, jak již název napovídá, operace *konvoluce*. Ta nahrazuje maticové násobení, kterým lze reprezentovat operace ve výše popsaném modelu hluboké dopředné sítě.

Operace *konvoluce* je ve vší obecnosti operace mezi dvěma číselnými funkcemi g a h se stejným definičním oborem, jejíž výstupem je nová číselná funkce standardně označovaná jako $g * h$. Uveďme zde definici konvoluce pro reálné funkce definované na $\mathbb{R}^d$, tedy $g, h : \mathbb{R}^d \to \mathbb{R}$:

$$(g * h)(t) = \int_{\mathbb{R}^d} g(\tau)h(t - \tau)d\tau$$

Důležitým předpokladem pro možnost konvoluce je samozřejmě konvergence integrálu na pravé straně.

Ačkoliv je konvoluce komutativní operací, v kontextu strojového učení se mezi oběma funkcemi vstupujícími do konvoluce rozlišuje. Funkce vstupující jako první se nazývá vstup a druhá funkce se nazývá jádrem. Dále se v kontextu konvolučních sítí standardně objevují diskrétní funkce, které nabývají nenulových hodnot pouze v konečně mnoha bodech. Potom integrál přes $\mathbb{R}^d$ přechází v konečnou sumu:

$$(g * h)(i_1, ..., i_d) = \sum_{j_1} ... \sum_{j_d} g(j_1, ..., j_d)h(i_1 - j_1, ..., i_d - j_d) \tag{1.3}$$

Díky komutativitě konvoluce lze též psát:

$$(g * h)(i_1, ..., i_d) = \sum_{j_1} ... \sum_{j_d} g(i_1 - j_1, ..., i_d - j_d)h(j_1, ..., j_d) \tag{1.4}$$

Při aplikaci komutativity došlo k tzv. *překlopení jádra* (termín pochází z anglického kernel flipping). Za vynechání překlopení jádra lze dojít ke *křížové korelaci*:

$$(g * h)(i_1, ..., i_d) = \sum_{j_1} ... \sum_{j_d} g(i_1 + j_1, ..., i_d + j_d)h(j_1, ..., j_d) \tag{1.5}$$

Mnoho knihoven zabývajících se neuronovými sítěmi dle [1] implementují křížovou korelaci namísto konvoluce, ačkoliv tuto svou implementaci nazývají konvolucí.

Další nedílnou součástí konvolučních sítí je tzv. *pooling*. Spolu s konvolucí tvoří mocný nástroj, který ve formě konvolučních a pooling vrstev hlubokých neuronových sítí přináší například invarianci sítě vůči malému posunutí vstupu (dle [1]).

Pooling je funkce, která nahrazuje hodnoty v bodech nějakou souhrnou statistikou určitého okolí daného bodu. Např. *max pooling* aplikovaný na matici se podívá na obdélníkové okolí předem definovaných rozměrů daného bodu a jako svůj výstup vybere maximální hodnotu nalezenou v onom okolí. Jiné oblíbené pooling funkce zahrnují funkce reportující průměr či L^2 normu daného obdelníkového okolí.

Standardní konvoluční vrstva neuronové sítě pak sestává ze tří fází. První fáze provádí paralelně několik konvolucí, které produkují sadu aktivcí. Druhá fáze, někdy označovaná jako *detekční fáze*, aplikuje na výstupy první fáze aktivační funkci. Třetí fáze potom provádí *pooling*.

Kapitola 2

Učení neuronové sítě

Standardní přístup k *učení neuronové sítě*, což je termín, kterým se označuje vhodné nalezení parametrů neuronové sítě, je paradigma učení s učitelem. Tento pohled na učení neuronové sítě předpokládá existenci tzv. *trénovací sady dat* $\mathbb{T}$ (angl. *training dataset*), což je uspořádaná dvojice obsahující množinu *vzorků* $\mathbb{X} = \{x^{(i)} | i \in \hat{N}\}$ a k nim příslušné *značky* $\mathbb{Y} = \{y^{(i)} | i \in \hat{N}\}$, kde pojem vzorek představuje vstup neuronové sítě jakožto zobrazení a pojem značka představuje správný výstup neuronové sítě; N je potom velikost trénovací sady $\mathbb{T}$. Trénovací sada pak hraje roli učitele.

2.1 Účelové funkce

Je-li pojem trenovací sady objasněn, lze přistoupit k termínu *účelové funkce* nebo též *ztrátové funkce*. Jedná se o reálnou funkci, která měří, jak moc se trénovaná neuronová síť mýlí ve svých predikcích na vzorcích trénovací sady. Úloha učení je potom převedena na úlohu optimalizace tohoto vhodně zvoleného kritéria.

Standardní účelová funkce je sestavena jako součet nebo průměr dílčích ztrát, které neuronová síť dosahuje na vzorcích trénovací sady:

$$J(\theta) = \sum_{i=1}^{N} L(F_\theta(x^{(i)}), y^{(i)}), \tag{2.1}$$

případně:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} L(F_\theta(x^{(i)}), y^{(i)}), \tag{2.2}$$

kde $x^{(i)}$ je i-tý vektor trénovací sady, $y^{(i)}$ je i-tý vektor trénovacích značek, N je velikost trénovací sady, F_θ neuronová síť jakožto funkce $F_\theta : \mathbb{R}^n \to \mathbb{R}^m$ parametrizovaná parametry θ, L značí konkrétní ztrátu pro daný vzorek a J je celková účelová funkce. V tomto textu se držme tvaru v (2.2).

Jedna z klasických účelových funkcí je funkce střední kvadratické chyby. Je dána přepisem:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{m} (F_\theta(x^{(i)})_j - y_j^{(i)})^2, \tag{2.3}$$

nebo

$$J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \|F_\theta(x^{(i)}) - y^{(i)}\|_2^2 \tag{2.4}$$

kde $x^{(i)}$ je i-tý vektor trénovací sady, $y^{(i)}$ je i-tý vektor trénovacích značek, N je velikost trénovací sady, F_θ neuronová síť jakožto funkce $F_\theta : \mathbb{R}^n \to \mathbb{R}^m$ parametrizovaná parametry θ a $\|\cdot\|_2$ je L^2 norma.

Další účelová funkce, která nachází uplatnění v klasifikačních problémech, se vypočte pomocí křížové entropie:

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N} H(y^{(i)}, F_\theta(x^{(i)})), \tag{2.5}$$

kde H označuje právě onu křížovou entropii mezi pravděpodobnostními distribucemi. Připomeňme, že klasifikační neuronová síť produkuje diskrétní pravděpodobnostní distribuce, a proto lze na výstup takovéto neuronové sítě a její značky (také pravděpodobnostní distribuce) aplikovat křížovou entropii. Onen výraz v (2.5) lze spočíst následovně:

$$J(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{m} y_j^{(i)} \cdot \ln(F_\theta(x^{(i)})_j), \tag{2.6}$$

přičemž $x^{(i)}$ je i-tý vektor trénovací sady, $y^{(i)}$ je i-tý vektor trénovacích značek, N je velikost trénovací sady, F_θ neuronová síť jakožto funkce $F_\theta : \mathbb{R}^n \to \mathbb{R}^m$ parametrizovaná parametry θ.

2.2 Algoritmus zpětného šíření chyby

Nejčastější metody učení neuronové sítě ve svém chodu pracují s gradientem účelové funkce podle parametrů neuronové sítě $\nabla_\theta J(\theta)$, který lze spočíst pomocí *algoritmu zpětného šíření chyby* (angl. *backpropagation*). Tento algoritmus však lze použít nejen v takto úzce specializovaném prostředí strojového učení, nýbrž i pro výpočet Jacobiho matice libovolné funkce (dle [1]).

Pro celkový popis algoritmu zaveďme pojem *výpočetního grafu*. Nechť vrcholy grafu představují proměnné, a to libovolných rozměrů, hrany grafu nechť jsou barevné a orientované, kde barva značí jednu z prováděných operací a orientace značí, jaká proměnná vznikla ze které pomocí dané operace.

Pojem výpočetního grafu lze ilustrovat následujícím příkladem: Nechť proměnná u je číslo a proměnné v a w vektory stejných rozměrů a platí, že proměnnou u lze získat jako $u = v \cdot w$. Potom tomuto příkladu náleží výpočetní graf o třech vrcholech, a to vrcholech proměnných v, w a u, a dvou hranách - první z v do u o barvě odpovídající tomu býti prvním argumentem skalárního součinu a druhá z w do u o barvě odpovídající tomu býti druhým argumentem skalárního součinu.

Dále je zapotřebí uvést *řetězové pravidlo* pro výpočet derivace složené funkce, o které se algoritmus opírá. Nechť $g : \mathbb{R}^n \to \mathbb{R}^m$ a $h : \mathbb{R}^m \to \mathbb{R}^p$, $a \in \mathbb{R}^n$, potom:

$$D(h \circ g)(a) = Dh(g(a)) \cdot Dg(a), \tag{2.7}$$

kde D značí totální diferenciál. Zúžíme-li se na $p = 1$, dostáváme:

$$\nabla(h \circ g)(a) = \nabla h(g(a)) \cdot Dg(a) \tag{2.8}$$

a podíváme-li se na i-tou komponentu gradientu $h \circ g$:

$$\partial_i(h \circ g)(a) = \sum_{j=1}^{m} \partial_j h(g(a)) \cdot \partial_i g_j(a), \tag{2.9}$$

kde g_j značí j-tou komponentu vektorové funkce g.

Tedy jak lze vidět v (2.8), pro algoritmus bude stěžejní násobení vektoru gradientu s maticí totálního diferenciálu. Vrcholy výpočetního grafu jsou ovšem libovolných rozměrů. Potom lze dané proměnné

urovnat do vektorů a spočíst gradient opět násobením vektoru gradientu s maticí totálního diferenciálu a následně převést vypočtený gradient zpět do příslušného tvaru.

Nyní lze nahlédnout na výpočet funkce jejíž gradient je žádoucí spočíst, například účelové funkce neuronové sítě, pomocí výpočetního grafu. Potom algoritmus zpětného šíření chyby postupuje po výpočetním grafu od výsledné proměnné k listovým vrcholům a aplikuje řetězové pravidlo.

V praxi je ovšem snadné natrefit na velmi složité výpočetní grafy, které vedou k vyhodnocování mnoha podvýrazů. Navíc mnoho takovýchto podvýrazů může být stejných. Při implementaci je tedy namístě otázka, zda již vyhodnocené výrazy uložit do paměti či je pokaždé vyhodnotit znovu. Je-li žádoucí co nejkratší doba běhu, pak je odpovědí vyhodnocené výrazy ukládat. Opačný přístup lze uplatnit při nedostatku paměti stroje.

2.3 Algoritmy učení

Základním algoritmem pro učení neuronové sítě je *gradientní sestup* (angl. *gradient descent*). Opírá se o fakt, že gradient reálné funkce určuje směr největšího spádu dané funkce v daném bodě. Proto, máme-li účelovou funkci $J(\theta)$, kde θ jsou parametry neuronové sítě, má smysl tyto parametry aktualizovat proti směru gradientu funkce J následujícím způsobem:

$$\theta \leftarrow \theta - \epsilon \cdot \nabla_\theta J(\theta), \tag{2.10}$$

kde ϵ je tzv. *řád učení* (angl. *learning rate*) - kladné číslo, které určuje velikost jednoho kroku; jedná se o další hyper-parametr neuronové sítě. Takovouto aktualizaci parametrů neuronové sítě lze provést několikrát, a to například tolikrát, dokud účelová funkce nedosáhne přijatelné hodnoty. Ideální by bylo, kdybychom gradientním sestupem dosáhli globálního minima účelové funkce, to ovšem není v žádném případě zaručeno, že se stane, gradientní sestup totiž dokáže nalézt pouze lokální minimum - ale to je pro reálné aplikace mnohdy dostačující.

Modifikací gradientního sestupu je tzv. *metoda hybnosti*. Ta uvádí na scénu novou proměnnou - *rychlost* v (z angl. *velocity*), která je stejných rozměrů jako gradient účelové funkce a nese v sobě informaci o předchozích odhadech gradientu účelové funkce. Její role v algoritmu učení je následující:

$$v \leftarrow \alpha \cdot v - \epsilon \cdot \nabla_\theta J(\theta) \tag{2.11}$$

$$\theta \leftarrow \theta + v \tag{2.12}$$

Užití hybnosti vede tedy k představení dalšího hyper-parametru, a to parametru $\alpha \in [0, 1)$, který určuje míru ovlivnění dalšího kroku předchozími odhady gradientu. Dle [1] jsou za hodnoty tohoto parametru nejčastěji volena čísla 0.5, 0.9 a 0.99.

Jinou modifikací gradientního sestupu, která je obdobou hybnosti, je *metoda Něstěrovovy hybnosti*. Ta má následující předpis iterace:

$$v \leftarrow \alpha \cdot v - \epsilon \cdot \nabla_\theta J(\theta + \alpha \cdot v) \tag{2.13}$$

$$\theta \leftarrow \theta + v \tag{2.14}$$

Existují další algoritmy, které pracují s proměnným řádem učení. Jedná se o *algoritmy s přizpůsobivým řádem učení*: *AdaGrad*, *RMSProp* a *Adam*. Tyto algoritmy přizpůsobují řád učení jednotlivým parametrům zvlášť.

Algoritmus *AdaGrad* dle [1] přizpůsobuje řád učení každému parametru jednotlivě, a to jeho škálováním nepřímo úměrně druhé odmocnině součtu všech hodnot gradientu, jež danému parametru v průběhu učení příslušel. To vede k tomu, že parametry, kterým přísluší velké hodnoty parciálních derivací účelové

funkce, mají úměrně tomu rychlý úbytek v řádu učení, zatímco parametry, kterým přísluší malé hodnoty parciálních derivací učelové funkce, mají úměrně tomu pomalý úbytek v řádu učení. Celkový efekt tedy je, že se síť pohybuje rychleji ve směrech menšího spádu. Jedna iterace by potom mohla vypadat následovně:

$$g \leftarrow \nabla_\theta J(\theta), \tag{2.15}$$
$$r \leftarrow r + g \odot g, \tag{2.16}$$
$$\theta \leftarrow \theta - \frac{\epsilon}{\delta + \sqrt{r}} \odot g, \tag{2.17}$$

kde δ je malé číslo (např. 10^{-7}) pro numerickou stabilitu, $\odot$ značí Hadamardův součin a výraz zlomku a odmocniny na třetím řádku je myšlen po složkách.

Nevýhoda tohoto algoritmu ovšem je jeho paměť - v proměnné r si pamatuje velmi vzdálené hodnoty gradientu, což dle [1] mnohdy vede k předčasnému poklesu řádu učení. Proto je namístě uvést další algoritmus - *RMSProp*. Tento algoritmus nahrazuje součet přes všechny hodnoty gradientu exponenciálně tlumeným váženým průměrem, a to způsobem, kde jedna iterace vypadá následovně:

$$g \leftarrow \nabla_\theta J(\theta), \tag{2.18}$$
$$r \leftarrow \rho \cdot r + (1 - \rho) \cdot g \odot g, \tag{2.19}$$
$$\theta \leftarrow \theta - \frac{\epsilon}{\delta + \sqrt{r}} \odot g, \tag{2.20}$$

kde δ je malé číslo (např. 10^{-7}) pro numerickou stabilitu, $\odot$ značí Hadamardův součin a výraz zlomku a odmocniny na třetím řádku je myšlen po složkách. Objevil se tu však nový hyper-parametr $\rho \in [0, 1)$ - *řád úpadku* (angl. *decay rate*).

Posledním představeným algoritmem je algoritmus *Adam*, který nese název z anglického *adaptive moments*, což přeloženo do češtiny zní jako přizpůsobivé momenty. V prvním přiblížení se jedná o kombinaci algoritmu RMSProp a metody hybnosti. Ve skutečnosti však je hybnost zakomponována již v následujícím, a to sice v odhadu prvního obecného momentu gradientu. Druhým aspektem, ve kterém se algoritmus liší od prostého RMSProp s hybností, jsou korekce pomocí prahu prováděné na odhadech prvního a druhého obecného momentu gradientu. Jedna iterace algoritmu vypadá:

$$g \leftarrow \nabla_\theta J(\theta), \tag{2.21}$$
$$s \leftarrow \rho_1 \cdot s + (1 - \rho_1) \cdot g, \tag{2.22}$$
$$r \leftarrow \rho_2 \cdot r + (1 - \rho_2) \cdot g \odot g, \tag{2.23}$$
$$\hat{s} \leftarrow \frac{s}{1 - \rho_1^t} \tag{2.24}$$
$$\hat{r} \leftarrow \frac{r}{1 - \rho_2^t} \tag{2.25}$$
$$\theta \leftarrow \theta - \frac{\epsilon}{\delta + \sqrt{\hat{r}}} \odot \hat{s}, \tag{2.26}$$

kde δ je malé číslo (např. 10^{-7}) pro numerickou stabilitu, $\odot$ značí Hadamardův součin, výraz zlomku a odmocniny na šestém řádku je myšlen po složkách, t je pořadí iterace a $\rho_1, \rho_2 \in [0, 1)$ jsou *řády úpadku*.

2.4 Stochastické algoritmy učení

Výše zmíněné metody, jak je patrné z jejich předpisů, počítají gradient účelové funkce $\nabla_\theta J(\theta)$. Tento krok je ovšem velmi časově náročný, protože standardní trénovací sady mívají velmi mnoho vzorků. Při připomenutí (2.2) se výpočet sestává z N výpočtů dílčích gradientů:

$$\nabla_\theta J(\theta) = \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta L(F_\theta(x^{(i)}), y^{(i)}), \tag{2.27}$$

kde $x^{(i)}$ je i-tý vektor trénovací sady, $y^{(i)}$ je i-tý vektor trénovacích značek, N je velikost trénovací sady, F_θ neuronová síť jakožto funkce $F_\theta : \mathbb{R}^n \to \mathbb{R}^m$ parametrizovaná parametry θ, L značí konkrétní ztrátu pro daný vzorek a J je celková účelová funkce.

Proto je doporučenou praxí dle [1] aproximovat gradient účelové funkce $\nabla_\theta J(\theta)$ pomocí výpočtu na tzv. *mini-dávce* (z angl. *mini-batch*). Jedná se v každém kroku gradientního sestupu nebo jeho modifikací o to, že se z trénovací sady rovnoměrně vybere $M \ll N$ vzorků gradient se odhadne pomocí výpočtu na těchto M vzorcích:

$$\nabla_\theta J(\theta) \approx \sum_{j=1}^{M} \nabla_\theta L(F_\theta(x^{(i_j)}), y^{(i_j)}), \tag{2.28}$$

kde $x^{(i)}$ je i-tý vektor trénovací sady, $y^{(i)}$ je i-tý vektor trénovacích značek, M je velikost mini-dávky, $i_j \sim U\{1, N\}$ jsou indexy vzorků vybraných do mini-dávky, F_θ je neuronová síť jakožto funkce $F_\theta : \mathbb{R}^n \to \mathbb{R}^m$ parametrizovaná parametry θ, L značí konkrétní ztrátu pro daný vzorek a J je celková účelová funkce.

Číslo M lze vybírat dle [1] v řádu jednotek až stovek. Při aplikaci této apriximace během standardního gradientního sestupu se algoritmu říká *stochastický gradientní sestup* (angl. *stochastic gradient descent*), ovšem tento úkrok stranou lze provést i v případě ostatních představených algoritmech, ty však pro svou stochastickou variantu nemají speciální název.

2.5 Srovnání algoritmů učení

Pro účely srovnávání algoritmů učení neuronové sítě lze zvolit mnoho kritérií. Jedním z nich by mohl být samotný průběh účelové funkce v závislosti na počtu provedených iterací vybraného algoritmu, když všechny představené algoritmy mají iterativní charakter.

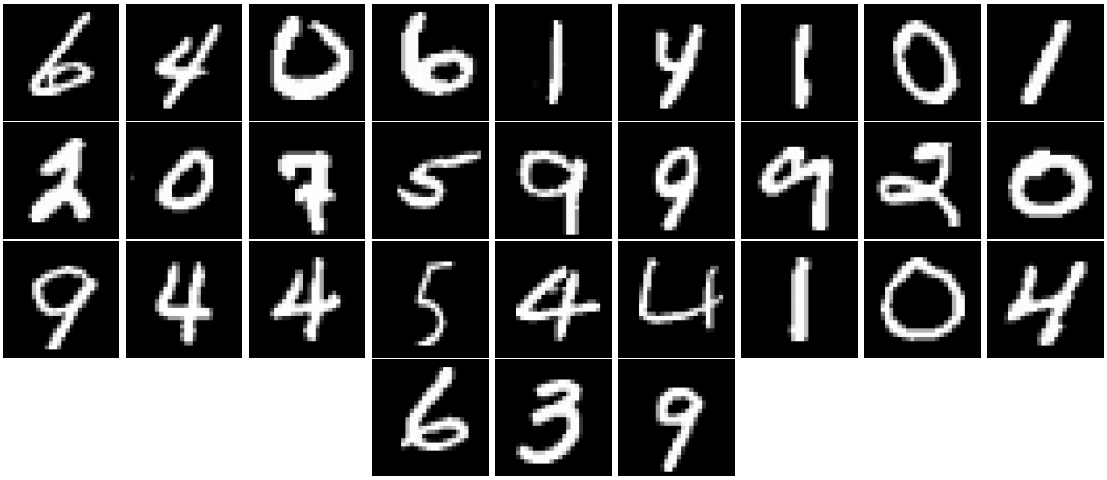
Jiným přístupem je užití tzv. *testovací sady* $\mathbb{S}$ (angl. *test dataset*). Svou strukturou testovací sada kopíruje sadu trénovací, jedná se tedy o uspořádanou dvojici množin vzorků $\mathbb{X} = \{x^{(i)} | i \in \hat{S}\}$ a značek $\mathbb{Y} = \{y^{(i)} | i \in \hat{S}\}$, kde S je velikost testovací sady.

Je-li neuronová síť svým charakterem síť klasifikační, pak lze sledovat podíl správných predikcí na testovacím datasetu vůči celkovému počtu vzorků. Výhodou tohoto přístupu je fakt, že při svém učení neuronová síť na vzorky testovacího datasetu nenarazila, což má za důsledek to, že lze očekávat stejnou úspěšnost sítě při její aplikaci. Tento přístup je využit v tomto textu.

Nyní je namístě vyslovit poznámku o inicializaci parametrů neuronové sítě před samotným učením. Dle [1] je standardním postupem pro inicializaci vybírat hodnoty parametrů náhodně, a to z rovnoměrného rozdělení na rozumném intervalu. Konkrétní experimenty v tomto textu pracují s následujícím rozdělením vah a prahů:

$$(W)_{i,j}, b_i \sim U\left(-\frac{1}{\sqrt{n}}, +\frac{1}{\sqrt{n}}\right), \tag{2.29}$$

kde n je v případě vah počet sloupečků matice vah, v případě prahů velikost vektoru prahů. Závěrem této poznámky tedy je, že inicializace parametrů neuronové sítě je náhodný proces. To má za důsledek fakt,



Obrázek 2.1: Datová sada MNIST

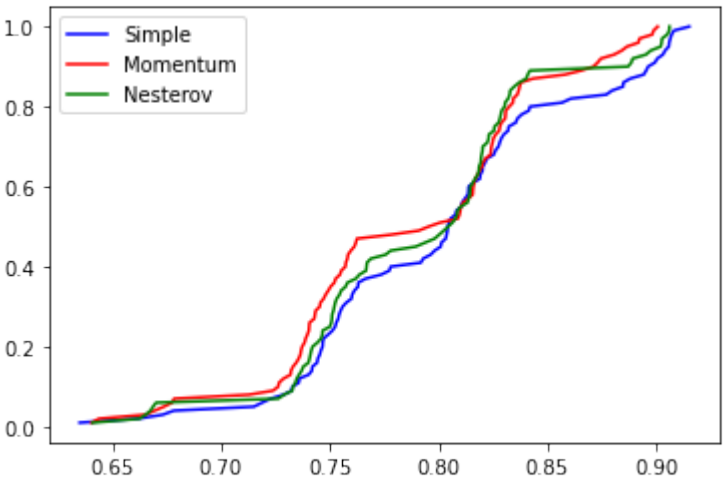
že na proces učení neuronové sítě lze nahlížet očima statistika. Tento text konkrétně nahlíží na úspěšnost neuronové sítě na testovací sadě jako na náhodnou veličinu. Potom lze totiž porovnávat jednotlivé algoritmy na základě distribuční funkce této specifické náhodné veličiny.

Nedílnou ingrediencí pro srovnání algoritmů učení je samotná sada dat a k nim příslušný úkol, zda se jedná o klasifikaci či o regresi. Tato část textu se věnuje úkolu klasifikace ručně psaných číslic z černobílého obrázku. Sada dat, která je zde použita je nazvána MNIST. Její trénovací sada $\mathbb{T}$ obsahuje 60 000 vzorků (a k nim odpovídajících značek) a testovací sada $\mathbb{S}$ obsahuje 10 000 vzorků (a k nim odpovídajících značek). Vzorky jsou ve své podstatě matice o rozměrech 28 řádků a 28 sloupečků, jejichž prvky jsou nezáporná celá čísla o hodnotě nejvýše 255. Tyto matice lze interpretovat jako obrázky.

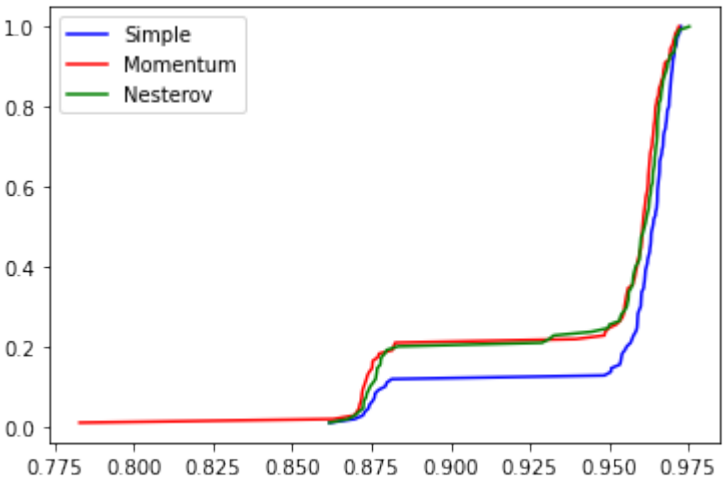
Přistupme nyní k samotnému srovnání algoritmů stochastický gradientní sestup, metoda hybnosti a metoda Něstěrovovy hybnosti (obě ve stochastické verzi). Pro srovnání těchto algoritmů byly provedeny následující dva experimenty: První se týká trénování jedné hluboké dopředné neuronové sítě těmito algoritmy pro úkol datové sady MNIST, jež je uvedena výše v textu, a to konkrétně aplikací 5 000 iterací algoritmu na nově inicializovanou síť. Pro stochastický gradientní sestup byl použit řád učení o hodnotě 10^{-2}, pro obě metody hybnosti byl použit řád učení 10^{-3} a koeficient $\alpha = 0.9$. Dále uveďme velikost mini-dávky $M = 30$ pro všechny tři algoritmy. V takovémto nastavení byly všechny tři algoritmy spuštěny stokrát. Na výsledné distribuční funkce lze nahlédnout v obrázku (2.2). Z grafu lze vyčíst takřka zanedbatelný rozdíl mezi metodou hybnosti a metodou Něstěrovovy hybnosti. Dále graf vyjadřuje nemalou větší úspěšnost obyčejného stochastického gradientního sestupu.

Druhý experiment je téměř totožný, jen je použit jiný model neuronové sítě, a to konkrétně se zakomponovanou konvolucí. Jinak je experiment totožný. Proto lze z Obr. (2.3) odezřít opět stejné výsledky, a to konkrétně, že stochastický gradientní sestup má mírně lepší výkonnost.

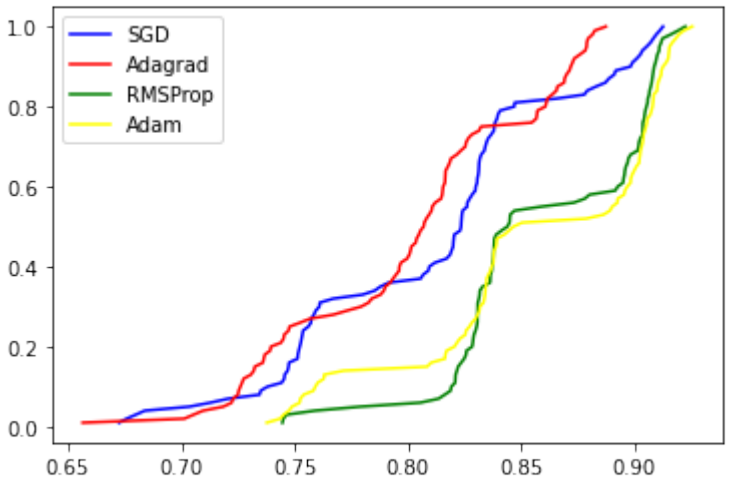
Pro srovnání algoritmů *stochastický gradientní sestup*, *AdaGrad*, *RMSProp* a *Adam* lze využít podkladů na obrázku (2.4), který zachycuje výsledky obdobných experimentů jako popsaných výše. Nastavení tohoto pokusu bylo následující: Pro stochastický gradientní sestup a algoritmus AdaGrad byl použit řád učení o hodnotě 10^{-2}, pro algoritmy RMSProp a Adam 10^{-3}. Pro AdaGrad bylo dále použito $\delta = 10^{-10}$, pro RMSProp $\delta = 10^{-8}$ a $\rho = 0.99$, pro Adam $\delta = 10^{-8}$, $\rho_1 = 0.9$ a $\rho_2 = 0.999$. Úkol byl stejný - natrénovat tentýž model dopředné neuronové sítě pro klasifikaci číslic datové sady MNIST za použití 5 000 iterací daného algoritmu. Učení sítě vždy proběhlo stokrát. Ze zmíněného obrázku vyplývá, že algoritmus AdaGrad je v tomto nastavení srovnatelný se stochastickým gradientním sestupem a že algoritmy RMSProp a Adam jsou minimálně pro toto specifické nastavení lepší.



Obrázek 2.2: Srovnání algoritmů učení I
Simple - stochastický gradientní sestup; *Momentum* - metoda hybnosti; *Nesterov* - metoda Něstěrovovy hybnosti.



Obrázek 2.3: Srovnání algoritmů učení II
Simple - stochastický gradientní sestup; *Momentum* - metoda hybnosti; *Nesterov* - metoda Něstěrovovy hybnosti.



Obrázek 2.4: Srovnání algoritmů učení III
SGD - stochastický gradientní sestup; *AdaGrad* - algoritmus AdaGrad; *RMSProp* - algoritmus RMSProp; *Adam* - algoritmus Adam

Kapitola 3

Adversariální vzorky

3.1 Metody generování adversariálních vzorků

3.1.1 FGSM

3.1.2 Iterativní FGSM

Kapitola 4

Robustní učení neuronové sítě

Závěr

Text závěru....

Literatura

[1] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*. MIT Press, 2016.

[2] I. Goodfellow, J. Shlens, C. Szegedy, *Explaining and Harnessing Adversarial Examples*. In 'International Conference on Learning Representations', ICLR 2015.

[3] J. Nocedal, S. Wright, *Numerical optimization*. Springer Science & Business Media, 2006.

[4] M. A. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2018.