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

$$f(x_1, ..., x_n) = \sigma(\sum_{i=1}^{n} w_i x_i + b), \tag{1.1}$$

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

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

$$s(x_1, ..., x_m)_i = \frac{e^{x_i}}{\sum_{j=1}^m e^{x_j}},$$

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 $s(x_1, ...x_m)_i = \sigma(x_i)$ 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 x následovně:

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

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 $g, h : \mathbb{R}^d \to \mathbb{R}$:

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

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

Předchozí kapitola představuje neuronové sítě jakožto složené zobrazení s mnoha parametry. Aby takováto neuronová síť byla k něčemu užitečná, např. ke klasifikaci obrázků, musí dané zkonstruované zobrazení vracet smysluplné výsledky k daným vstupům. Toho se v praxi dociluje vhodným nastavením hyper-parametrů sítě a následným nalezením hodnot parametrů daného složeného zobrazení, které odpovídají funkční neuronové síti. Toto hledání parametrů se též nazývá jako *učení neuronové sítě* a provádí se metodami numerické optimalizace jistého vhodně zvoleného kritéria, které se označuje jako *účelová* či *ztrátová funkce*.

Nutným předpokladem ke konstrukci vhodné účelové funkce je tzv. *trénovací sada* vzorků a k nim příslušné *značky*. Jedná se o množinu možných vstupů, které jsou vybaveny správným výstupem. Je-li dána trénovací sada a značky, lze definovat účelovou funkci jako jakési měřidlo ukazující, jak moc se trénovaná neuronová síť mýlí, je-li vpuštěna na vzorky trénovací sady. Tomuto přístupu k učení neuronové sítě se také říká učení s učitelem.

Pro úplnost poznamenejme, že standardní sada dat pro neuronové sítě též obsahuje kromě trénovací sady další sadu, a to tzv. *testovací sadu* - opět spolu se správnými značkami. Tato sada slouží jako míra určující úspěšnost natrénované sítě. Konkrétně tak lze vzít podíl počtu vzorků testovací sady, pro které neuronová síť provedla správnou predikci, a celkového počtu vzorků v testovcí sadě.

2.1 Účelové funkce

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

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{m} (F_\theta(x^{(i)})_j - y_j^{(i)})^2, \tag{2.1}$$

nebo

$$L(\theta) = \frac{1}{N} \sum_{i=1}^{N} \|F_\theta(x^{(i)}) - y^{(i)}\|_2^2 \tag{2.2}$$

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

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

$$L(\theta) = -\frac{1}{N} \sum_{i=1}^{N} H(y^{(i)}, F_\theta(x^{(i)})), \tag{2.3}$$

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 (2.3) lze spočíst následovně:

$$L(\theta) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{j=1}^{m} y_j^{(i)} * \ln(F_\theta(x^{(i)})_j), \tag{2.4}$$

přičemž $x^{(i)}$ je i-tý vektor trénovací sady, $y^{(i)}$ je i-tý vektor trénovacích značek, 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 $\nabla_\theta L(\theta)$, 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 $h : \mathbb{R}^m \to \mathbb{R}^p$, $x \in \mathbb{R}^n$, potom:

$$D(h \circ g)(f(x)) = Dh(g(x)) \cdot Dg(x), \tag{2.5}$$

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

$$\nabla(h \circ g)(x) = \nabla h(g(x)) \cdot Dg(x) \tag{2.6}$$

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

$$\partial_i(h \circ g)(x) = \sum_{j=1}^{m} \partial_j h(g(x)) \cdot \partial_i g_j(x), \tag{2.7}$$

kde g_j značí j-tou komponentu vektorové funkce g.

Tedy jak lze vidět v (2.6), 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 $L(\theta)$, kde θ jsou parametry neuronové sítě, má smysl tyto parametry aktualizovat proti směru gradientu funkce L následujícím způsobem:

$$\theta \leftarrow \theta - \epsilon \cdot \nabla_\theta L(\theta), \tag{2.8}$$

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.

Gradientní sestup má ovšem nevýhodu v tom, že v každém kroku počítá gradient účelové funkce přes celou trénovací sadu. Ta mnohdy sestává z tolika vzorků, že opakovaný výpočet gradientu učelové funkce pro účely učení je časově velmi náročný a pro reálné aplikace nevhodný. Z těchto důvodů je vhodnější použít *stochastický gradientní sestup* (angl. *stochastic gradient descent*). Ten narozdíl od obyčejného gradientního sestupu nepočítá gradient účelové funkce přesně, nýbrž jej odhaduje výpočtem gradientu modifikované účelové funkce, kde modifikace účelové funkce spočívá v jejím vyhodnocování pouze na znatelně menší podmnožině trénovacích vzorků. Tato podmnožina trénovacích vzorků je ideálně náhodně vybraná a v každém kroku stochastického gradientního sestupu jiná a může obsahovat od jednotek po stovky vzorků. Pro úplnost lze poznamenat, že se této podmnožině trénovacích vzorků říká *mini-dávka* (angl. *mini-batch*).

Modifikací stochastického gradientního sestupu je tzv. *metoda hybnosti*. To 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:

$$v \leftarrow \alpha \cdot v - \epsilon \cdot \nabla_\theta L(\theta) \tag{2.9}$$

$$\theta \leftarrow \theta + v \tag{2.10}$$

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 stochastického gradientního sestupu, která je obdobou hybnosti, je *Něstěrovova hybnost*. Ta má následující předpis:

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

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

Pro srovnání stochastického gradientního sestupu, metody hybnosti a metody Něstěrovovy hybnosti se lze opřít o výsledky experimentů Hubert a Ivan (viz Obr. 2.3 a Obr. 2.3). Experiment Hubert naznačuje, že standardní stochastický gradientní sestup je v porovnání se zbylými dvěma metodami úspěšnější, byť jen o málo. Totéž potvrzuje experiment Ivan. Co se týče srovnání metody hybnosti a metody

Něstěrovovy hybnosti, ty jsou dle experimentů stejně úspěšné. Důležité je však poznamenat, že v těchto experimentech řád učení pro stochastický gradientní sestup byl desetkrát větší než u metod hybností (10^{-2} oproti 10^{-3}). Byl-li by býval stejný, buď by se neuronové sítě pomocí metod hybností vůbec nic nenaučili (pro společný řád učení 10^{-2}), nebo by stochastický gradientní sestup nedosahoval takovývh výkonů (pro společný řád učení 10^{-3}).

Krok stranou ke stochastickému gradientnímu sestupu od klasického gradientního sestupu se dle [1] nevhodně projeví nestabilitou algoritmu v pozdější fázi učení, kdy ačkoli se účelová funkce pohybuje kolem lokálního minima, odhad gradientu se neblíží nulovému vektoru, a proto dochází k oscilacím účelové funkce. Tomuto jevu lze předejít právě pomocí *proměnného řádu učení*, a to v tom smyslu, že s postupem učení řád vhodným způsobem klesá.

Existují ale i sofistikovanější algoritmy, které pracují s proměnným řádem učení. Jedná se o *algoritmy s přizpůsobivým řádem učení*: Jmenovitě *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:

$$g \leftarrow \nabla_\theta L(\theta), \tag{2.13}$$

$$r \leftarrow r + g \odot g, \tag{2.14}$$

$$\theta \leftarrow \theta - \frac{\epsilon}{\delta + \sqrt{r}} \odot g, \tag{2.15}$$

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:

$$g \leftarrow \nabla_\theta L(\theta), \tag{2.16}$$

$$r \leftarrow \rho \cdot r + (1 - \rho) \cdot g \odot g, \tag{2.17}$$

$$\theta \leftarrow \theta - \frac{\epsilon}{\delta + \sqrt{r}} \odot g, \tag{2.18}$$

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:

$$g \leftarrow \nabla_\theta L(\theta), \tag{2.19}$$

$$s \leftarrow \rho_1 \cdot s + (1 - \rho_1) \cdot g, \tag{2.20}$$

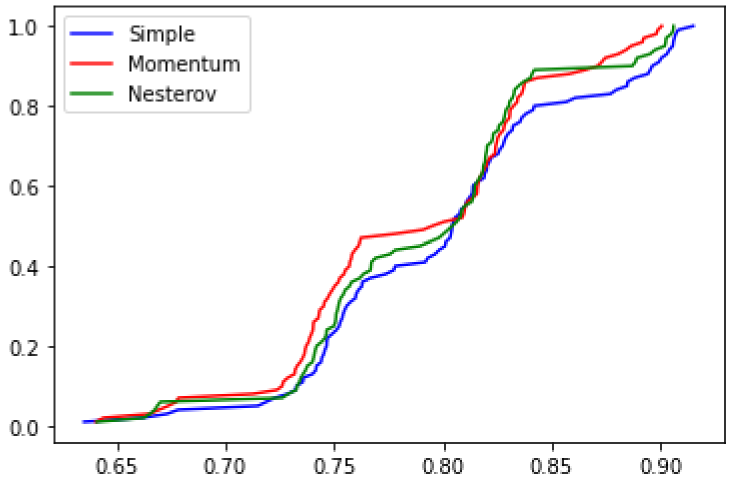
$$r \leftarrow \rho_2 \cdot r + (1 - \rho_2) \cdot g \odot g, \tag{2.21}$$

$$\hat{s} \leftarrow \frac{s}{1 - \rho_1^t} \tag{2.22}$$

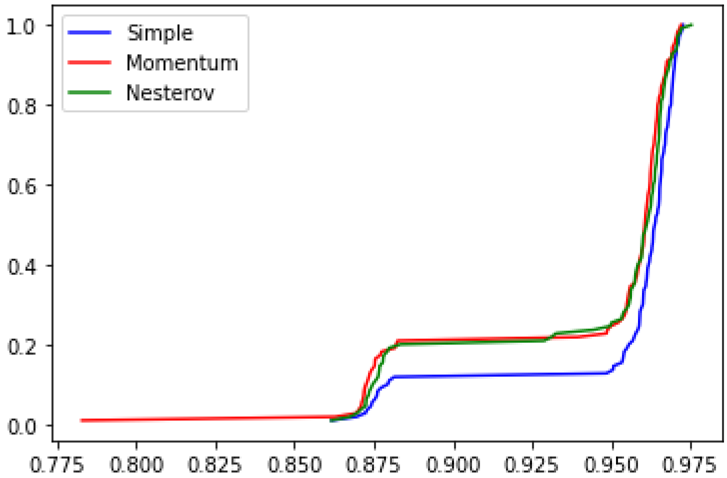
$$\hat{r} \leftarrow \frac{r}{1 - \rho_2^t} \tag{2.23}$$

$$\theta \leftarrow \theta - \frac{\epsilon}{\delta + \sqrt{\hat{r}}} \odot \hat{s}, \tag{2.24}$$

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*.



Obrázek 2.1: Srovnání algoritmů učení - experiment Hubert
Obrázek znázorňuje experimentálně zjištěné distribuční funkce tří pseudo-náhodných veličin, a to úspěšností neuronové sítě na testovací sadě natrénované vybraným algoritmem. Uvážíme-li že se nenatrénovaná neuronová síť objeví kdesi v parametrickém prostoru, a toto její objevení se je pseudo-náhodné, pak to, kam vybraný algoritmus dovede danou neuronovou síť v onom parametrickém prostoru, lze brát jako náhodný jev, a tedy úspěšnost neuronové sítě na testovací sadě jako náhodnou veličinu. Potom lze algoritmy učení porovnávat na základě distribučních funkcí těchto náhodných veličin. Prakticky takové porovnání pak proběhne následovně: Je-li graf dané distribuční funkce více vpravo, pak daná distribuce je více štědrá a naděluje lépe natrénované modely - tedy jí odpovídající algoritmus je v daném nastavení lepší. Nyní k tomuto konkrétnímu obrázku: Graf označený jako *Simple* znázorňuje výsledky *standardního stochastického gradientního sestupu*, kde bylo provedeno 5001 iterací, gradient aproximován výpočtem gradientu na dávce o velikosti 30 vzorků, řád učení byl 10^{-2}; graf označený jako *Momentum* znázorňuje výsledky *stochastického gradientního sestupu za použití hybnosti*, kde bylo provedeno 5001 iterací, gradient aproximován výpočtem gradientu na dávce o velikosti 30 vzorků, řád učení byl 10^{-3} a koeficient $\alpha = 0.9$; graf označený jako *Nesterov* znázorňuje výsledky stochastického gradientního sestupu za použití *Něstěrovovy hybnosti*, kde bylo provedeno 5001 iterací, gradient aproximován výpočtem gradientu na dávce o velikosti 30 vzorků, řád učení byl 10^{-3} a koeficient $\alpha = 0.9$. Všechny tyto výsledky byly dosaženy učením stejného modelu hluboké dopředné neuronové sítě, a to na sadě dat MNIST. Každá distribuční funkce byla sestavena na základě 100 pozorování.



Obrázek 2.2: Srovnání algoritmů učení - experiment Ivan
Obdobně jako Obr. 2.3 tento obrázek znázorňuje tři experimentálně zjištěné distribuční funkce. Graf označený jako *Simple* znázorňuje výsledky *standardního stochastického gradientního sestupu*, kde bylo provedeno 5001 iterací, gradient aproximován výpočtem gradientu na dávce o velikosti 30 vzorků, řád učení byl 10^{-2}; graf označený jako *Momentum* znázorňuje výsledky *stochastického gradientního sestupu za použití hybnosti*, kde bylo provedeno 5001 iterací, gradient aproximován výpočtem gradientu na dávce o velikosti 30 vzorků, řád učení byl 10^{-3} a koeficient $\alpha = 0.9$; graf označený jako *Nesterov* znázorňuje výsledky stochastického gradientního sestupu za použití *Něstěrovovy hybnosti*, kde bylo provedeno 5001 iterací, gradient aproximován výpočtem gradientu na dávce o velikosti 30 vzorků, řád učení byl 10^{-3} a koeficient $\alpha = 0.9$. Všechny tyto výsledky byly dosaženy učením stejného modelu konvoluční neuronové sítě, a to na sadě dat MNIST. Každá distribuční funkce byla sestavena na základě 110 pozorování.

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.