



Gymnázium
České Budějovice
Jírovcova 8

STŘEDOŠKOLSKÁ ODBORNÁ ČINNOST
Obor č. 18: Informatika

Neuronové sítě

Jonáš Havelka

vedoucí práce: Dr. rer. nat. Michal Kočer

Jihočeský kraj

V Českých Budějovicích 3. dubna 2020

2019/2020

Prohlášení

Prohlašuji, že jsem svou práci SOČ vypracoval samostatně a použil jsem pouze prameny a literaturu uvedené v seznamu bibliografických záznamů.


Prohlašuji, že tištěná verze a elektronická verze soutěžní práce SOČ jsou shodné.

Nemám závažný důvod proti zpřístupňování této práce a jejích příloh v souladu se zákonem č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších předpisů, bude-li dodržena licence MIT (viz soubor LICENSE v přílohách).


V Českých Budějovicích dne podpis

Jonáš Havelka

Abstrakt





Neuronové sítě se dnes objevují všude, ať už jde o vyhledávání, překládání nebo třeba jen zpracovávání dat. Mnoho programovacích jazyků má své knihovny pro práci s umělou inteligencí, ale právě  Kotlin, který je mým oblíbeným programovacím jazykem a který lze použít skoro kdekoli (webové stránky, server, mobily), takovou knihovnu postrádá. Proto jsem se rozhodl svoji práci koncipovat jako snahu o implementování takové knihovny.

Klíčová slova

Neuronové sítě, Neuron, Umělá inteligence, Aktivační funkce,  Kotlin, Multiplatformní knihovna, Java, Javascript

Poděkování

Poděkování patří hlavně mému učiteli informatiky, který je zároveň vedoucím mé práce, za skvělou výuku na hodinách a velkou trpělivost při kontrole našich prací. Také nesmím zapomenout na Alžbětu Neubauerovou, která mě celý rok podporovala a několikrát provedla korekturu mé práce.

Dále bych rád poděkoval všem komunitám, jejichž nástroje jsem používal, tj. JetBrains, v jejichž programovacím jazyce  Kotlin programuji a jejichž prostředí IntelliJ k tomu využívám,  Gradle, který používám ke kompilaci, \LaTeX , ve kterém píšu text, a dále  git a  GitHub, jež uchovávají má data, ať už text nebo knihovnu.

Obsah

I	Teoretická část	9
1	Laický náhled na neuronové sítě	10
1.1	Neuron	10
1.2	Aktivační funkce	10
1.3	Sítě	11
1.4	Dopředná propagace a zpětná propagace	11
1.5	Využití neuronových sítí	12
2	Formální náhled	15
2.1	Definice neuronu a sítě	15
2.2	Dopředná propagace	16
2.3	Chybová funkce	16
2.4	Zpětná propagace	17
2.5	Sít	18
2.5.1	Dopředná propagace	19
2.5.2	Zpětná propagace	19
2.5.3	Zakomponování biasu	20
2.6	Aktivační funkce	21
2.7	Shrnutí	26
II	Praktická část	27
3	Struktura knihovny	29
3.1	core	29
3.1.1	IActivationFunctions	29

3.1.2	ActivationFunctions	29
3.1.3	CustomFunction	30
3.1.4	INeuralNetwork	30
3.1.5	BasicNeuralNetwork	31
3.1.6	ConvolutionalNetwork	31
3.2	mnistDatabase	32
3.2.1	Databáze MNIST	33
3.2.2	Databáze EMNIST	33
4	Používání knihovny	34
4.1	Trénování sítě	34
4.2	Používání sítě	35
4.3	Nastavení hodnot	35
	Apendix	35
	Slovníček pojmů	37
	Bibliografie	38
	Seznam obrázků	39
	Přílohy	40
	Zdrojový kód knihovny	USB
	Dokumentace	USB
	Testovací dataset	USB
	Zdrojový kód ukázkového programu	USB
	Ukázkový program	USB
	Zdrojový kód práce v L ^A T _E Xu	USB
	Přehled grafů aktivačních funkcí	41
	Zdrojový kód knihovny	42
	src/commonMain/kotlin/core/ActivationFunctions.kt	42
	src/commonMain/kotlin/core/BasicNeuralNetwork.kt	46
	src/commonMain/kotlin/core/ConvolutionalNeuralNetwork.kt	49

src/commonMain/kotlin/core/CustomFunction.kt	51
src/commonMain/kotlin/core/IActivationFunctions.kt	51
src/commonMain/kotlin/core/INeuralNetwork.kt	52
src/commonMain/kotlin/mnistDatabase/loadFile.kt	53
src/commonTest/kotlin/sample/Constants.kt	55
src/commonTest/kotlin/sample/NeuralNetworkTest.kt	56
src/jsTest/kotlin/sample/ConstantsJS.kt	58
src/jvmMain/kotlin/mnistDatabase/loadFileJVM.kt	58
src/jvmTest/kotlin/sample/ConstantsJVM.kt	58
src/jvmTest/kotlin/sample/NeuralNetworkTestJVM.kt	59
src/commonMain/kotlin/core/AssociativeMemory.kt	61
src/commonMain/kotlin/core/Neuron.kt	62

Úvod

Neuronové sítě jsou v poslední době velmi skloňované téma. Nikdo pořádně neví, jak to, že fungují tak dobře. Cílem této práce však nebude zkoumat neuronové sítě, ale implementovat je v co největším rozsahu (ať už struktury bez širšího využití jako asociativní paměť, nebo často používané konvoluční sítě na rozpoznávání obrázků).

Kotlin je ideální programovací jazyk pro vývoj knihovny, protože je interoperabilní s Javou, Javascriptem i C, a tak umožňuje tuto knihovnu používat jak pro JVM (Java Virtual Machine), tak i v prohlížeči nebo v programech kompilovaných přímo do binárního kódu.

V textu jsou použity pojmy ze stavby biologického neuronu, objektově orientovaného programování, Kotlinu, atd. Tyto pojmy jsou vysvětleny na konci práce.

Celá originální maturitní práce je k dispozici na GitHubu, text včetně zdrojového LaTeXu na adrese https://github.com/JoHavel/Maturitni-Seminarni-Prace/tree/my_work a knihovna samotná pak na <https://github.com/JoHavel/NeuralNetwork>.

Část I

Teoretická část

1 Laický náhled na neuronové sítě

1.1 Neuron

Počítačové neuronové sítě nejsou jen výmysl lidí, jejich základ nalezneme v nervových soustavách živočichů. Základní stavební jednotka takové soustavy (stejně tak i neuronové sítě) je *neuron*. Neuron funguje tak, že přes *dendrity* přijímá elektrické (přesněji iontové) signály od jiných neuronů a když součet signálů přeteče určitou danou *mez*, vyšle neuron signál přes *axony* dál do dalších neuronů.

Přenos signálu z axonu do dendritu se odehrává v malých prostorách mezi nimi zvaných *synapse*. Vodivost synapsí je ovlivněna jejich chemickým složením, a proto se domníváme, že proces učení probíhá měněním těchto chemických spojů [book:Informatika].

Náš umělý neuron tedy bude mít *seznam dendritů* (nesoucích informaci z jakého neuronu vedou signál a jak ho mění synapse), tzv. *aktivační funkci* (viz dále) a výstupní signál. Často navíc bude obsahovat základní hodnotu (angl. *bias*), která reprezentuje mez, při jejímž překročení začne neuron vysílat signál. Jinak řečeno posouvá aktivační funkci ve směru osy x .

Neuron (hlavně ten umělý) ilustruje obrázek 2.1 nacházející se v další kapitole. Podrobněji o souvislosti biologických a umělých neuronových sítích pojednává [book:BNN].

1.2 Aktivační funkce

Jak už bylo zmíněno, přírodní neuron funguje na principu toho, že když součet vstupních signálů nepřekračuje určitou mez, nevysílá neuron žádný (nebo téměř žádný) signál. Když je však tato mez překonána, neuron vyšle signál. V podstatě tedy vysílá buď 0 nebo 1. Pro účely umělého neuronu je 0 a 1 nedostačující, jelikož při procesu učení potřebujeme měnit hodnoty jemně, abychom nerozbili již naučené znalosti.

Proto se jako aktivační funkce (tedy to, co určuje jaký má být výstup v závislosti na součtu vstupů, v případě přírody tedy funkce zobrazující interval $-\infty$ až mez (bias) na 0

a zbylá čísla na 1 viz *binární krok* v sekci 2.6) používají funkce co nejvíce podobné právě tomuto binárnímu kroku, které jsou ale spojité a mají co „nejhezčí“ derivace (protože při zpětné propagaci právě podle derivace určíme, jak moc daný neuron ovlivňuje výsledek).

1.3 Sítě

Jelikož „nahodilé neurony“ by se těžko udržovaly v paměti a operace na nich by byly velmi pomalé, potřebujeme síť nějak uspořádat. Nejjednodušším uspořádáním jsou *vrstvy* (viz obrázek 1.1). Každý neuron z jedné vrstvy má dendrity ze všech neuronů z vrstvy minulé. Tak se předejde cyklům, které jsou složité na výpočty, a navíc si nemusíme u každého neuronu pamatovat, ze kterých neuronů do něj vede signál.

Velmi využívanými strukturami jsou také konvoluční neuronové sítě, kde nejdříve aplikujeme filtry¹ na části vstupních dat a teprve výstupy z těchto filtrů jsou vstupem do neuronové sítě. O konvolučních sítích se můžete dočíst v [lec:CNN] nebo v [sem:CNN].

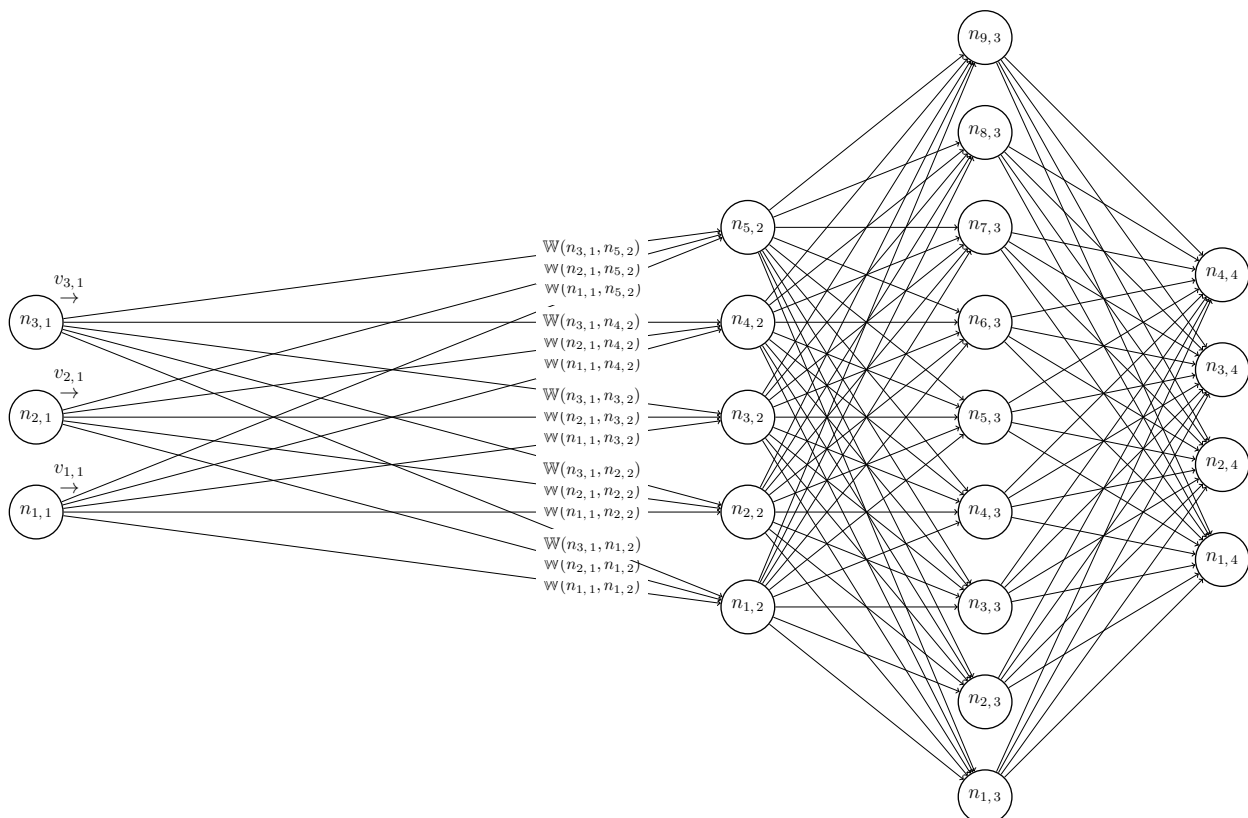
I tak se „nahodilé neurony“ občas používají, jelikož při malém množství neuronů a hlavně při malém množství synapsí je přepočítání samotných neuronů efektivnější než počítání celých vrstev. Ukázkou takové malé sítě je asociativní paměť, kde neuronům přiřadíme objekty, které si tato síť má „pamatovat“. Když chceme zjistit, co je v paměti asociováno s daným objektem, vybudíme (v umělé síti to znamená nastavíme výstupní signál na 1) neuron odpovídající tomuto objektu a následně sledujeme, které další neurony jsou vybudeny (viz obrázek 1.2). Takto funguje i lidská paměť, pamatujeme si právě asociace. Umělou asociativní paměť zmiňuje [book:Informatika].

1.4 Dopředná propagace a zpětná propagace

Dopředná propagace (častěji se používá anglický výraz *forward propagation*) je jednoduše spočítání signálů ve všech neuronech. Tedy u každého neuronu se sečtou vstupní signály (popř. přičte *bias*) a spočítá se funkční hodnota aktivační funkce v tomto bodě.

Naopak zpětná propagace (častěji se používá anglický výraz *backward propagation* či *backpropagation*) je na základě chyby, kterou spočítáme z výstupu neuronové sítě a předpokládaného výstupu, určení, které proměnné hodnoty (synapse a *biasy*) se na ní nejvíce

¹Často malé neuronové sítě, které sami vytvoříme. Síť používané jako filtry se nemusí učit (učení filtrů není vzhledem k obtížnosti zatím implementováno). Další možný filtr je třeba Fourierova transformace viz [art:FCNN], kterou se však dále zabývat nebudeme (tato možnost není ani implementována v knihovně).



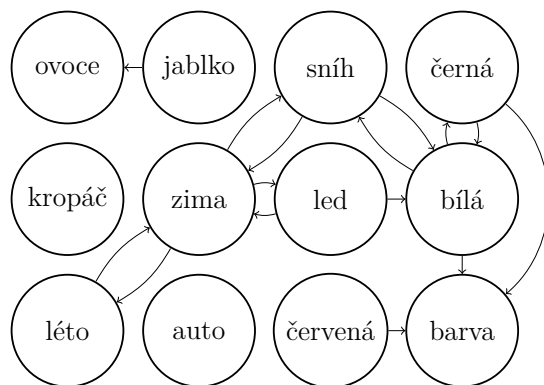
Obr. 1.1: Běžná neuronová síť (\mathbb{W} jsou váhy, n neurony a v je výstupní signál, viz kapitola 2, konkrétně sekce 2.5)

podílejí. Potom tyto hodnoty posuneme odpovídajícím způsobem (stejně jako příroda mění chemické vlastnosti synapse). Z matematického pohledu se hodnoty posunou proti směru gradientu chyby, jelikož právě gradient udává, kterým směrem máme souřadnice (tj. váhy a biasy) posunout, aby funkce (tj. chyba) vzrostla.

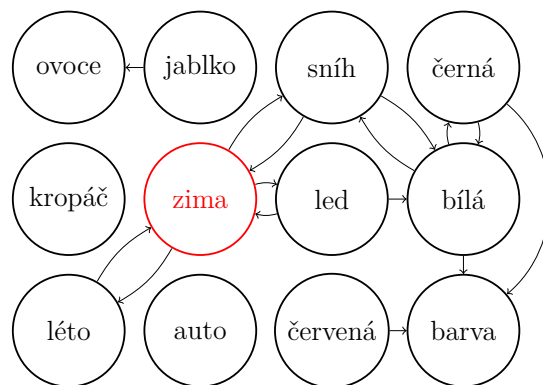
1.5 Využití neuronových sítí

Než se pustíme do matematiky, která stojí za fungováním neuronových sítí, ještě si řekneme, kde a jaké neuronové sítě využíváme. Jedno z nejviditelnějších využití je rozpoznávání obrázků, protože takovou úlohu jen stěží zvládnou běžné algoritmy. Mezi rozpoznávání obrázků patří jak strojové čtení textů, tak třeba rozpoznávání tváře nebo klasifikace, zda je na obrázku morče, nebo slon. K tomu se používají hlavně konvoluční sítě, jelikož filtr rozezná hrany a různé útvary a neuronová síť podle toho určí dané rozřazení (znak, člověka, zvíře...).

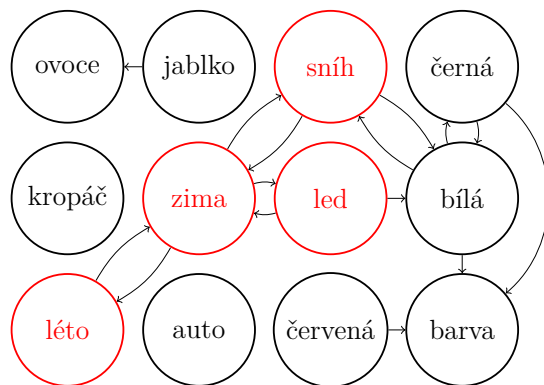
Další oblastí je překlad. Překládat slova zvládneme jednoduše podle slovníků, ale aby věta dávala smysl a slovo bylo přeloženo v kontextu věty, potřebujeme něco více. Pro to se používá vektorový prostor slov, tedy všem slovům přiřadíme určitý vektor (to musíme udělat



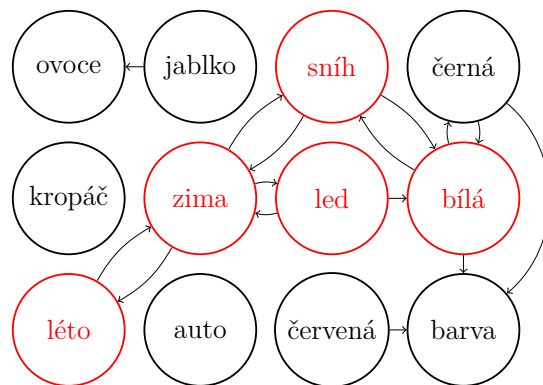
(a) Vybuzení 1. neuronu (zimy)



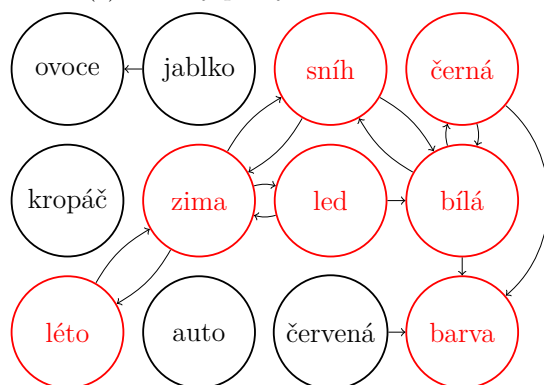
(b) 1 krok po vybuzení 1. neuronu



(c) 2 kroky po vybuzení 1. neuronu



(d) 3 kroky po vybuzení 1. neuronu



(e) 4 kroky po vybuzení 1. neuronu

Obr. 1.2: Asociativní paměť, červeně jsou vybuzené neurony



Obr. 1.3: Generovaná tvář [online:Face]

vždy, protože neuronová síť nemá jiný vstup) a poté na vzorovém textu učíme neuronovou síť odhadovat slovo podle několika okolních slov. Při tom ale neupravujeme jen hodnoty neuronové sítě, ale i vektorů slov. Tím dostaneme vektorový prostor slov, na kterém se překládající neuronová síť (jiná než ta, co vyrobila vektorový prostor) naučí překládat velmi lidsky. Stejný vektorový prostor se dá použít i na neuronovou síť generující text.

Když už bylo zmíněno generování, umělé neuronové sítě jsou schopny i generovat obrázky, hudbu, atd.² K tomu se používá systém GAN (tj. Generative adversarial network) [art:GAN], což jsou dvě sítě, jedna generuje a druhá dostane dvojici objekt vytvořený člověkem (resp. skutečností v případě fotek) a objekt vygenerovaný první sítí a má za úkol určit, který je který. Tyto sítě se učí spolu a výsledkem jsou relativně pěkná díla viz obrázek 1.3.

²Stále je to však na základě nějakého datasetu obrázků nebo hudby.

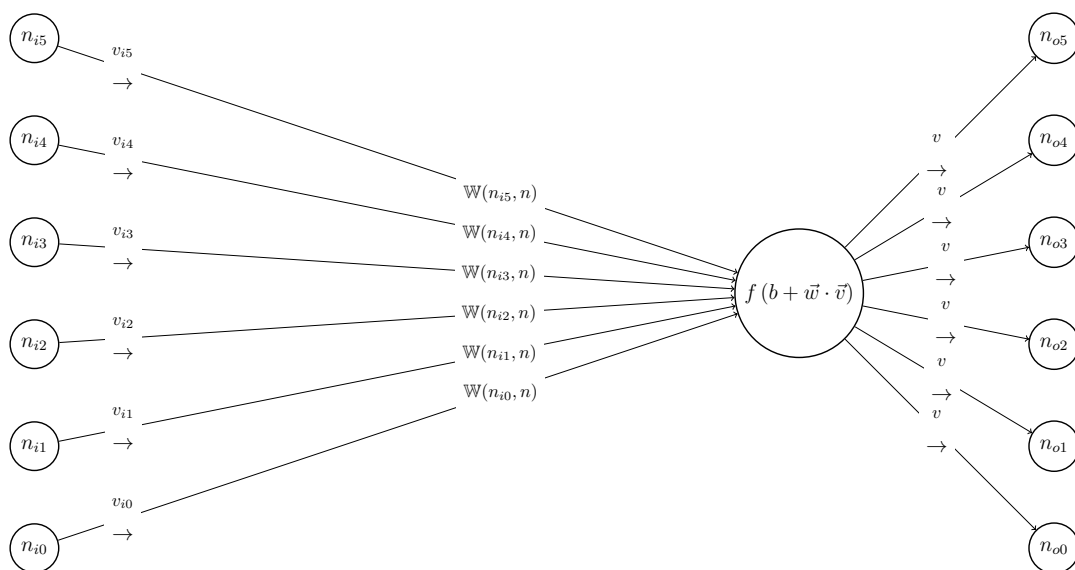
2 Formální náhled

Matematikou za neuronovými sítěmi a její implementací v Pythonu se zabývají videa [vid:NN]. Kniha zabývající se touto problematikou je např. [book:NN].

V dalším textu $\vec{x} \cdot \vec{y}$ značí skalární součin¹ vektorů \vec{x} a \vec{y} . Vektory jsou uvedeny horizontálně, ale chápeme to jako by byly vertikálně².

2.1 Definice neuronu a sítě

Vstupní neurony: N_{in} Axony vstupních neuronů Synapse z n_{i*} do n Dendrity Vyšetřovaný neuron: n Výstupní neurony: N_{out}



Obr. 2.1: Neuron

Označme $\nu = (N, W, F)$ neuronovou síť, N je množina všech jejích neuronů, $W : N \times N \rightarrow \mathbb{R}$ jsou váhy (angl. weights) udávající sílu synapse mezi dvěma neurony (v případě, že mezi neurony synapse není, je W rovno 0) a $F : \mathbb{R}^{|N_v|} \rightarrow \mathbb{R}$ je chybová funkce udávající velikost chyby podle rozdílu reálných hodnot od chtěných hodnot výstupních neuronů (N_v).

¹To jest to samé jako $\vec{x}^T \vec{y}$.

²Mohli bychom doplnit za každou definici vektoru T , třeba (2.2) přepíšeme jako $\vec{v} = (v_1, v_2, \dots)^T$

Nechť $n \in N$, $n = (N_{in}, N_{out}, f, b, v, \varepsilon)$ je neuron, kde $N_{in} = \{n_x \in N | W(n_x, n) \neq 0\}$ je množina neuronů, které vysílají signál do n , $N_{out} = \{n_x \in N | W(n, n_x) \neq 0\}$ je množina neuronů, které přijímají signál od n , $f: \mathbb{R} \rightarrow \mathbb{R}$ je aktivační funkce, $b \in \mathbb{R}$ je bias, $v \in \mathbb{R}$ je signál vycházející z n a ε je chyba (parciální derivace chybové funkce podle $f^{-1}(v)$ ³).

2.2 Dopředná propagace

Potom dopředná propagace (tedy spočítání v) vypadá takto⁴:

$$v = f \left(b + \sum_{n_x \in N_{in}, v_x \in n_x} v_x \cdot W(n_x, n) \right) \quad (2.1)$$

To lze při označení

$$\vec{v} = (v_1, v_2, \dots) \quad (2.2)$$

$$\vec{w} = (w_1, w_2, \dots) \quad (2.3)$$

$$(\forall n_x \in N_{in}) (\exists i \in \mathbb{N}) (v_i \in n_x \wedge w_i = W(n_x, n)) \quad (2.4)$$

zapsat vektorově jako:

$$v = f(b + \vec{w} \cdot \vec{v}) \quad (2.5)$$

Případně můžeme do vektorů „zakomponovat“ i bias⁵:

$$\vec{v} = (1, v_1, v_2, \dots) \quad (2.6)$$

$$\vec{w} = (b, w_1, w_2, \dots) \quad (2.7)$$

$$(\forall n_x \in N_{in}) (\exists i \in \mathbb{N}) (v_i \in n_x \wedge w_i = W(n_x, n)) \quad (2.8)$$

$$v = f(\vec{w} \cdot \vec{v}) \quad (2.9)$$

2.3 Chybová funkce

Anglicky loss function nebo někdy také cost function. Udává, nakolik se neuronová síť strefila do správného výstupu. Většinou nás ale nezajímá její hodnota (rozlišujeme pouze, zda síť odpověděla dobře, nebo ne), používáme ji jen jako pomyslné hodnocení ve zpětné propagaci.

³Derivace aktivačních funkcí se často snadno spočítá z funkční hodnoty, proto uvádím, že hledám derivaci v bodě, kde je daná funkční hodnota, značím přitom $f^{-1}(y) = x \Leftrightarrow f(x) = y$.

⁴ $v_x \in n_x$ značí, že v_x je signál neuronu n_x , obdobně u ostatních informací v neuronu.

⁵To v knihovně není použito z důvodu netriviálního přidávání prvku do vektoru.

Její gradient, tedy derivace podle všech proměnných (vah a biasů) v neuronové síti, totiž udává, jak poupravit hodnoty, aby neuronová síť odpovídala lépe.

Pro naše potřeby stačí pouze jediná chybová funkce

$$E(x) = 0,5 \sum_{n_o \in O} (v_{od} - v_o) \quad (2.10)$$

, kde O je množina výstupních neuronů, v_o jsou jejich výstupní signály a v_{od} jsou odpovídající chtěné výstupní signály. Tato funkce má výhodu, že její derivace podle libovolného v_o je

$$\frac{\delta E}{\delta v_o} = v_{od} - v_o \quad (2.11)$$

, tedy ε výstupních neuronů spočítáme pouze jako rozdíl chtěných a reálných výstupů.

2.4 Zpětná propagace

Při zpětné propagaci je důležitý vzorec pro derivaci složené funkce, někdy také znám jako „řetízkové pravidlo“ (pro funkci jedné proměnné platí rovnice (2.12), pro více pak rovnice (2.13))⁶

$$\frac{dy}{dx} = \frac{dz}{dx} \frac{dy}{dz} \quad (2.12)$$

$$\frac{\delta y}{\delta x} = \sum_z \frac{\delta z}{\delta x} \frac{\delta y}{\delta z} \quad (2.13)$$

Díky tomu můžeme ε neuronu spočítat pomocí

$$f_x^{-1}(v_x) = \sum_{n_y \in N_{out, x}, v_y \in n_y} v_y \cdot W(n_y, n_x) \quad (2.14)$$

tj.

$$\frac{\delta f_x^{-1}(v_x)}{\delta v_y} = W(n_y, n_x) \quad (2.15)$$

takto:

$$\varepsilon = \frac{\delta E}{\delta f^{-1}(v)} = \sum_{n_x \in N_{out}, f_x \in n_x, v_x \in n_x} \frac{\delta E}{\delta f_x^{-1}(v_x)} \cdot \frac{\delta f_x^{-1}(v_x)}{\delta f^{-1}(v)} \quad (2.16)$$

$$\varepsilon = \sum_{n_x \in N_{out}, f_x \in n_x, v_x \in n_x} \frac{\delta E}{\delta f_x^{-1}(v_x)} \cdot \frac{\delta f_x^{-1}(v_x)}{\delta v} \cdot \frac{\delta v}{\delta f^{-1}(v)} \quad (2.17)$$

$$\varepsilon = \frac{\delta v}{\delta f^{-1}(v)} \sum_{n_x \in N_{out}, \varepsilon_x \in n_x} \varepsilon_x \cdot W(n, n_x) \quad (2.18)$$

⁶Pro funkce musí platit, že mají v daných bodech derivaci, viz [book:Matanalysis].

$$\varepsilon = f' (f^{-1}(v)) \sum_{n_x \in N_{out}, \varepsilon_x \in n_x} \varepsilon_x \cdot W(n, n_x) \quad (2.19)$$

ε nás dovede k tomu, o kolik musíme posunout bias. Hlavním parametrem neuronové sítě jsou ale váhy (funkce W). Derivaci chybové funkce podle váhy určíme za pomoci rovnice (2.14), tj.

$$\frac{\delta f_y^{-1}(v_y)}{\delta W(n, n_y)} = v \quad (2.20)$$

a z rovnice (2.1):

$$\frac{\delta E}{\delta W(n, n_y)} = \frac{\delta E}{\delta f_y^{-1}(v_y)} \cdot \frac{\delta f_y^{-1}(v_y)}{\delta W(n, n_y)} = \varepsilon_y \cdot v \quad (2.21)$$

Obdobně jako v předchozím případě definujeme vektory⁷:

$$\vec{\varepsilon} = (\varepsilon_1, \varepsilon_2, \dots) \quad (2.22)$$

$$\vec{w} = (w_1, w_2, \dots) \quad (2.23)$$

$$\frac{\delta E}{\delta \vec{w}} = \left(\frac{\delta E}{\delta w_1}, \frac{\delta E}{\delta w_2}, \dots \right) \quad (2.24)$$

$$(\forall n_x \in N_{out}) (\exists i \in \mathbb{N}) (\varepsilon_i \in n_x \wedge w_i = W(n, n_x)) \quad (2.25)$$

$$\varepsilon = f' (f^{-1}(v)) \cdot (\vec{w} \cdot \vec{\varepsilon}) \quad (2.26)$$

$$\frac{\delta E}{\delta \vec{w}} = \vec{\varepsilon} \cdot v \quad (2.27)$$

Vektor $\frac{\delta E}{\delta \vec{w}}$ už stačí jen přičíst k \vec{w} , abychom upravili hodnoty $W(n, n_x)$.

Pomocí tohoto můžeme spočítat všechno kromě ε na výstupních neuronech. To můžeme z rovnice (2.11) ($n_o \in O$ jsou výstupní neurony, $\varepsilon_o \in n_o$, $f_o \in n_o$ a $v_o \in n_o$):

$$\varepsilon_o = \frac{\delta E}{\delta f_o^{-1}(v_o)} = \frac{\delta E}{\delta v_o} \cdot \frac{\delta v_o}{\delta f_o^{-1}(v_o)} = (v_{od} - v_o) f'_o (f_o^{-1}(v_o)) \quad (2.28)$$

2.5 Síť

V sekci 1.3 jsme se bavili o tom, že nepoužívanější sítě mají neurony seřazené do vrstev. Necht jsou tudíž neurony uspořádány ve vrstvách číslovaných přirozenými čísly od 1 a necht jsou navíc i neurony v každé vrstvě zvlášť očíslovány přirozenými čísly od 1 (tj. vrstva je vlastně vektor neuronů). Potom značme L_x vrstvu s indexem x a $n_{x,y}$ neuron s indexem y příslušící do L_x . To znamená, že pokud $N_{in} \in n_{x,y}$, tak $N_{in} = L_{x-1}$, a pokud $N_{out} \in n_{x,y}$, tak $N_{out} = L_{x+1}$. Následně zavedme vektory ($v_{x,i} \in n_{x,i}$, $b_{x,i} \in n_{x,i}$, $f_{x,i} \in n_{x,i}$ a $\varepsilon_{x,i} \in n_{x,i}$):

⁷Značení $\frac{\delta E}{\delta \vec{w}}$ a $\frac{\delta E}{\delta W}$ z rovnic (2.24) a (2.43) neznačí derivace podle vektoru a matice, ale je to symbolické značení pro vektor a matici derivací podle jednotlivých složek daného tensoru.

$$\vec{v}_x = (v_{x,1}, v_{x,2}, \dots) \quad (2.29)$$

$$\vec{w}_{x,i} = (W(n_{x,1}, n_{x+1,i}), W(n_{x,2}, n_{x+1,i}), \dots) \quad (2.30)$$

$$\vec{w}'_{x,i} = (W(n_{x,i}, n_{x+1,1}), W(n_{x,i}, n_{x+1,2}), \dots) \quad (2.31)$$

$$\vec{b}_x = (b_{x,1}, b_{x,2}, \dots) \quad (2.32)$$

$$\vec{f}_x = (f_{x,1}, f_{x,2}, \dots) \quad (2.33)$$

$$\vec{\varepsilon}_x = (\varepsilon_{x,1}, \varepsilon_{x,2}, \dots) \quad (2.34)$$

$$\frac{\delta E}{\delta \vec{w}'_{x,i}} = \left(\frac{\delta E}{\delta W(n_{x,y}, n_{x+1,1})}, \frac{\delta E}{\delta W(n_{x,y}, n_{x+1,2})}, \dots \right) \quad (2.35)$$

2.5.1 Dopředná propagace

Přepíšeme rovnici dopředné propagace (2.5):

$$v_{x,i} = f_{x,i} (b_{x,i} + \vec{w}_{x-1,i} \cdot \vec{v}_{x-1}) \quad (2.36)$$

Můžeme využít matici vah a maticové násobení (aplikaci vektoru funkcí $\vec{f}(\vec{x})$ chápejme tak, že na každou složku \vec{x} se aplikuje odpovídající složka \vec{f}):

$$\mathbb{W}_x = \begin{pmatrix} w_{x,1} \\ w_{x,2} \\ \vdots \end{pmatrix} = \begin{pmatrix} W(n_{x,1}, n_{x+1,1}) & W(n_{x,2}, n_{x+1,1}) & \dots \\ W(n_{x,1}, n_{x+1,2}) & W(n_{x,2}, n_{x+1,2}) & \dots \\ \vdots & \vdots & \ddots \end{pmatrix} \quad (2.37)$$

$$\vec{v}_x = \vec{f}_x (\vec{b}_x + \mathbb{W}_{x-1} \cdot \vec{v}_{x-1}) \quad (2.38)$$

2.5.2 Zpětná propagace

Nyní přepíšeme rovnice (2.26) a (2.27) zpětné propagace:

$$\varepsilon_{x,i} = f'_{x,i} (f_{x,i}^{-1}(v_{x,i})) \odot (\vec{w}'_{x,i} \cdot \vec{\varepsilon}_{x+1}) \quad (2.39)$$

$$\frac{\delta E}{\delta \vec{w}'_{x,i}} = \vec{\varepsilon}_{x+1} \cdot v_{x,i} \quad (2.40)$$

Rovnici (2.39) můžeme převést hned do maticového tvaru (\mathbb{W}^T značí transponovanou matici \mathbb{W} , $\vec{f}^{-1}(x)$ a $\vec{f}'(x)$ značí aplikaci inverzní funkce a derivace funkce podobně jako v (2.38), \odot značí násobení po složkách⁸):

$$\vec{\varepsilon}_x = \vec{f}'_x \left(\vec{f}_x^{-1}(\vec{v}_x) \right) \odot (\mathbb{W}_x^T \cdot \vec{\varepsilon}_{x+1}) \quad (2.41)$$

Pro rovnici (2.40) potřebujeme spojit definice (2.24) (definice vektoru derivací), kterou přepíšeme do tvaru vrstev:

$$\frac{\delta E}{\delta w_{x,i}} = \left(\frac{\delta E}{\delta W(n_{x,i}, n_{x+1,1})}, \frac{\delta E}{\delta W(n_{x,i}, n_{x+1,2})}, \dots \right) \quad (2.42)$$

a (2.37) (definici matice vah):

$$\frac{\delta E}{\delta \mathbb{W}_x} = \begin{pmatrix} \frac{\delta E}{\delta w_{x,1}} \\ \frac{\delta E}{\delta w_{x,2}} \\ \vdots \end{pmatrix} = \begin{pmatrix} \frac{\delta E}{\delta W(n_{x,1}, n_{x+1,1})} & \frac{\delta E}{\delta W(n_{x,1}, n_{x+1,2})} & \dots \\ \frac{\delta E}{\delta W(n_{x,2}, n_{x+1,1})} & \frac{\delta E}{\delta W(n_{x,2}, n_{x+1,2})} & \dots \\ \vdots & \vdots & \ddots \end{pmatrix} \quad (2.43)$$

Nyní jsme již schopni zapsat rovnici (2.40) maticově:

$$\frac{\delta E}{\delta \mathbb{W}_x} = \vec{\varepsilon}_{x+1} \vec{v}_x^T \quad (2.44)$$

I spočítání ε u poslední vrstvy (tj. neuronů v O , značme ji L_o) lze zapsat vektorově (\vec{v}_{od} zde značí vektor předpokládaných výsledků):

$$\vec{\varepsilon}_o = \vec{f}'_x \left(\vec{f}_x^{-1}(\vec{v}_x) \right) \odot (\vec{v}_{od} - \vec{v}_o) \quad (2.45)$$

2.5.3 Zakomponování biasu

Nejdříve musíme upravit vektory a matice:

$$\vec{v}_x = (1, v_{x,1}, v_{x,2}, \dots) \quad (2.46)$$

$$\vec{f}_x = (1, f_{x,1}, f_{x,2}, \dots) \quad (2.47)$$

$$\vec{\varepsilon}_x = (0, \varepsilon_{x,1}, \varepsilon_{x,2}, \dots) \quad (2.48)$$

$$\mathbb{W}_x = \begin{pmatrix} 1 & 0 & 0 & \dots \\ b_{x+1,1} & W(n_{x,1}, n_{x+1,1}) & W(n_{x,2}, n_{x+1,1}) & \dots \\ b_{x+1,2} & W(n_{x,1}, n_{x+1,2}) & W(n_{x,2}, n_{x+1,2}) & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad (2.49)$$

⁸Násobením vektorů $\vec{x} = (x_1, x_2, \dots)$ a $\vec{y} = (y_1, y_2, \dots)$ tzv. po složkách získáme vektor $\vec{x} \odot \vec{y} = (x_1 \cdot y_1, x_2 \cdot y_2, \dots)$.

$$\frac{\delta E}{\delta \mathbb{W}_x} = \begin{pmatrix} 0 & 0 & 0 & \dots \\ \frac{\delta E}{\delta b_{x+1,1}} & \frac{\delta E}{\delta W(n_{x,1}, n_{x+1,1})} & \frac{\delta E}{\delta W(n_{x,1}, n_{x+1,2})} & \dots \\ \frac{\delta E}{\delta b_{x+1,2}} & \frac{\delta E}{\delta W(n_{x,2}, n_{x+1,1})} & \frac{\delta E}{\delta W(n_{x,2}, n_{x+1,2})} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \quad (2.50)$$

Rovnice (2.38) (samozřejmě bez biasu:

$$\vec{v}_x = \vec{f}_x(W_{x-1} \cdot \vec{v}_{x-1}) \quad (2.51)$$

), (2.41) a (2.44) poté fungují pořád stejně. Rovnice (2.45) funguje také shodně, jelikož prostě řekneme, že první člen odhadu vyšel tak, jak má, tj. $\vec{\varepsilon} = (0, \dots)$

2.6 Aktivační funkce

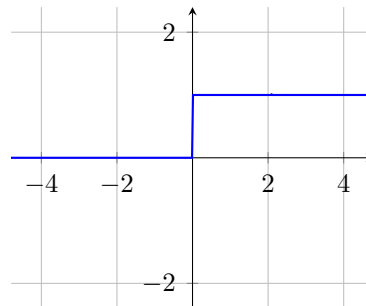
Jelikož neurony mají bias, není nutné udávat aktivační funkce obecně, stačí je jen udat tak, že $x = 0$ odpovídá mezi v pomyslném biologickém neuronu. Mezi aktivační funkce⁹ patří:

- *Binary step*

$$f(x) = \begin{cases} 0, & \text{když } x < 0 \\ 1, & \text{když } x \geq 0 \end{cases} \quad (2.52)$$

$$f'(x) = \begin{cases} 0, & \text{když } x \neq 0 \\ +\infty, & \text{když } x = 0 \end{cases} \quad (2.53)$$

(česky *binární krok*), již zmíněná funkce, jež odpovídá reálnému neuronu, ale není použitelná pro učení na základě gradientu, jelikož má derivaci 0 všude kromě bodu $x = 0$, kde je nespojitá.



Obr. 2.2: Binární krok

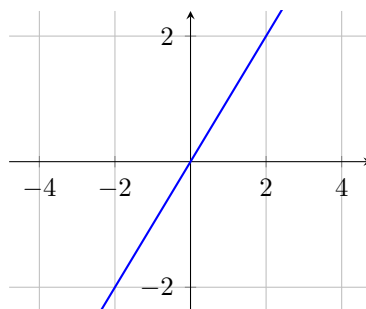
⁹Funkce jsem čerpal převážně z [wiki:ActivationFunctions].

- *Identity*

$$f(x) = x \quad (2.54)$$

$$f'(x) = 1 \quad (2.55)$$

(česky *identita*) odpovídá stavu, jako kdyby tam žádná funkce nebyla. Její derivace je 1, tedy se velmi snadno určí v libovolném bodě.



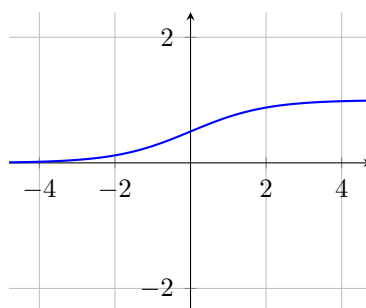
Obr. 2.3: Identita

- *Sigmoid* [**article:AF**] (značí se σ)

$$\sigma(x) = \frac{1}{1 + e^{-x}} \quad (2.56)$$

$$\sigma'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = \frac{1}{1 + e^{-x}} \left(1 - \frac{1}{1 + e^{-x}}\right) = \sigma(x) \cdot (1 - \sigma(x)) \quad (2.57)$$

je jedna z nejznámějších aktivačních funkcí. Je to vlastně takový hladký přechod mezi 0 a 1. Také je na σ dobře vidět, proč se často počítá derivace z funkční hodnoty, místo počítání exponenciální funkce a dělení si vystačíme s násobením a odčítáním. *Sigmoida* se také používá ve spojení s ostatními funkcemi, většinou $\sigma(x)$ pro kladné a druhá funkce pro záporné.



Obr. 2.4: σ

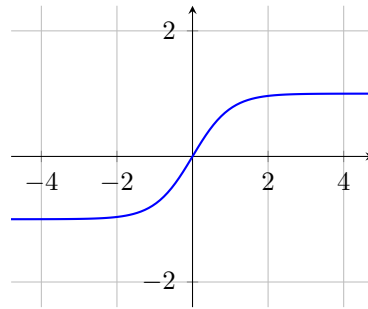
- Nesmíme zapomenout na *sigmoidě* podobnou a také často používanou funkci *hyperbolický tangens* (\tanh) [**article:AF**] [**book:FFActivationFunctions**]:

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{2}{1 + e^{-2x}} - 1 = 2 \cdot \sigma(2x) - 1 \quad (2.58)$$

$$\tanh'(x) = \frac{1}{\cosh^2(x)} = \frac{\cosh^2(x) - \sinh^2(x)}{\cosh^2(x)} = 1 - \tanh^2(x) \quad (2.59)$$

$$\tanh'(x) = 4 \cdot \sigma'(2x) = 4 \cdot \sigma(2x) \cdot (1 - \sigma(2x)) \quad (2.60)$$

Největší rozdíl oproti σ je, že může nabývat i záporných hodnot, což sice moc neodpovídá přírodnímu neuronu, ale když si rozmyslíme, že stačí zvětšit biasy u neuronů, do kterých neuron s aktivační funkcí \tanh vysílá signál, dospějeme k výsledku, že tato funkce také funguje.



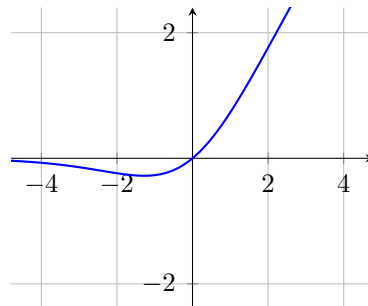
Obr. 2.5: Hyperbolický tangens

- Další funkce s vazbou na *sigmoidu* je funkce *swish* [**article:AF**]:

$$f(x) = x \cdot \sigma(x) = \frac{x}{1 + e^{-x}} \quad (2.61)$$

$$f'(x) = x + \sigma'(x) = x + \sigma(x) \cdot (1 - \sigma(x)) \quad (2.62)$$

Nepodařilo se mi ale najít derivaci za pomoci funkční hodnoty.



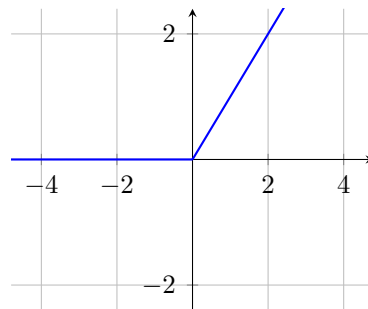
Obr. 2.6: Swish

- Ukazuje se, že identita jako taková se v podstatě použít nedá, ale hojně využívaná je její „upravená“ verze *rectified linear unit* [article:AF] (česky něco jako *napravená přímá úměrnost*), která záporná čísla převádí na nulu a v kladných se chová jako *identita*:

$$f(x) = \begin{cases} 0, & \text{když } x < 0 \\ x, & \text{když } x \geq 0 \end{cases} \quad (2.63)$$

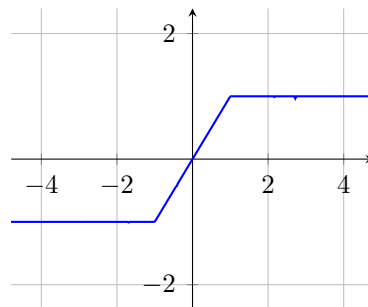
$$f'(x) = \begin{cases} 0, & \text{když } x < 0 \\ 1, & \text{když } x > 0 \\ \text{neexistuje,} & \text{když } x = 0 \end{cases} \quad (2.64)$$

Trochu připomíná biologický neuron, protože pro záporné hodnoty nevysílá, ale na rozdíl od něj má variabilní hodnotu vysílaného signálu. Často se například používá ve filtrech, jelikož chceme detekovat, zda je někde hrana, ale nechceme vysílat záporný signál, když někde hrana není, protože může být o pixel vedle.



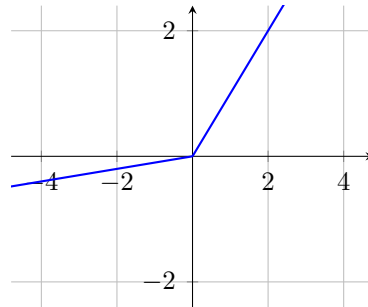
Obr. 2.7: Rectified linear unit

- Kromě této verze je v knihovně ještě *leaky* (děravá či prosakující) *rectified linear unit* [article:AF], která v záporných hodnotách nedává nulu, ale *přímou úměrnost*.



Obr. 2.8: Hard hyperbolic function

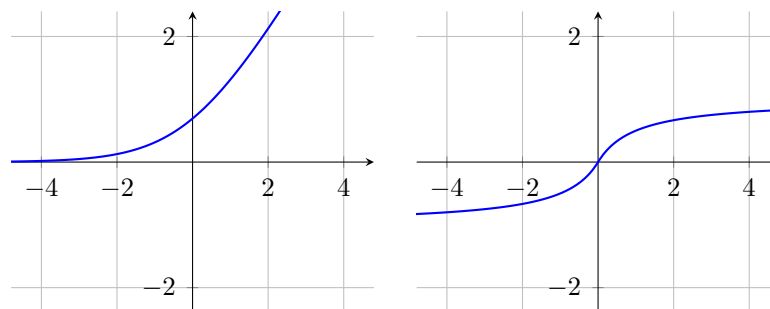
- K těmto funkcím můžeme přiřadit i *hard hyperbolic function*, která je *identitou* pouze na intervalu $(-1, 1)$, tedy odpovídá biologickému neuronu asi nejvíce z těchto „lineárních funkcí“.



Obr. 2.9: Leaky rectified linear unit pro koeficient úměrnosti 0.1

- *Rectified unit* není hladká (nemá derivaci v bodě nula), ale to lze napravit, když použijeme funkci *soft plus* [article:AF] ($\ln(1 + e^x)$). Podobnou úpravu lze udělat i s funkcí *signum* (znaménko, často se značí *sign*), což je téměř *binární krok*¹⁰, akorát v záporných hodnotách nabývá funkční hodnoty -1 místo 0. *Signum* se dá zapsat jako podíl x a $|x|$, tudíž tato úprava (*soft sign* [article:AF]) vypadá následovně:

$$f(x) = \frac{x}{|x| + 1} \quad (2.65)$$



(a) Soft plus

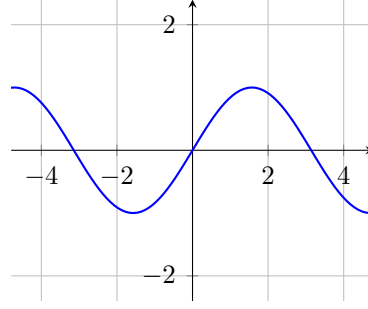
(b) Soft signum

Obr. 2.10: Soft funkce

- Jednou skupinou funkcí, se kterými se sice experimentuje, ale stěží najdete nějaké využití, jsou ty, které nejsou monotonní¹¹, jako sinus, kosinus, *Gaussova funkce* (e^{-x^2}), apod. Vzhledem k jejich mizivému využití je implementován pouze *sinus*.

¹⁰Z důvodu téhle podobnosti není ani implementována.

¹¹Můžeme si všimnout, že téměř všechny předchozí funkce jsou neklesající, většina dokonce rostoucí na celém definičním oboru.



Obr. 2.11: Sinus

2.7 Shrnutí

Rovnice:

- Dopředné propagace, tj. (2.5) resp. (2.9) nebo (2.38) resp. (2.51):

$$v = f(b + \vec{w} \cdot \vec{v})$$

$$v = f(\vec{w} \cdot \vec{v})$$

$$\vec{v}_x = \vec{f}_x(\vec{b}_x + W_{x-1} \cdot \vec{v}_{x-1})$$

$$\vec{v}_x = \vec{f}_x(W_{x-1} \cdot \vec{v}_{x-1})$$

- Zpětné propagace, tj. (2.19) a (2.27) nebo (2.41) a (2.44):

$$\varepsilon = f'(f^{-1}(v)) \sum_{n_x \in N_{out}, \varepsilon_x \in n_x} \varepsilon_x \cdot W(n, n_x)$$

$$\frac{\delta E}{\delta \vec{w}} = \vec{\varepsilon} \cdot v$$

$$\vec{\varepsilon}_x = \vec{f}_x'(\vec{f}_x^{-1}(\vec{v}_x)) \odot (\mathbb{W}_x^T \cdot \vec{\varepsilon}_{x+1})$$

$$\frac{\delta E}{\delta \mathbb{W}_x} = \vec{\varepsilon}_{x+1} \vec{v}_x^T$$

- Prvotní části zpětné propagace, tj. (2.28) nebo (2.45)

$$\varepsilon_o = (v_{od} - v_o) f'_o(f_o^{-1}(v_o))$$

$$\vec{\varepsilon}_o = \vec{f}_x'(\vec{f}_x^{-1}(\vec{v}_x)) \odot (\vec{v}_{od} - \vec{v}_o)$$

nám popisují matematiku stojící za fungováním neuronových sítí, tedy naším cílem bude je implementovat. Navíc k implementování těchto rovnic potřebujeme naprogramovat samotný neuron, který jsme si definovali v sekci 2.1 jako:

$$n = (N_{in}, N_{out}, f, b, v, \varepsilon)$$

Také často používáme aktivační funkce, proto by v naší knihovně neměly chybět.

Část II

Praktická část

Cílem této práce je knihovna, která nám umožní používat neuronové sítě v Kotlinu. Jak bylo řečeno na konci minulé kapitoly, musí obsahovat aktivační funkce, nejlépe všechny uvedené v sekci 2.6, zároveň však umožnit uživateli definovat si funkce vlastní. Poté se zaměříme hlavně na neuronovou síť obsahující vrstvy, její implementace bude zároveň zahrnovat jak implementaci neuronu, tak implementaci dopředné a zpětné propagace.

Konvoluční síť pro jednoduchost naprogramujeme za použití sítí s vrstvami, tedy jedinou věc, kterou potřebujeme implementovat, je způsob používání filtru. Nakonec se podíváme i na asociativní paměť, pro kterou navíc potřebujeme naprogramovat neurony, jelikož na rozdíl od neuronové sítě s vrstvami zde nelze ukládat jednotlivé hodnoty neuronu pohromadě (např. všechny v do jednoho pole, nebo všechny ε do jiného).

Aby nemuselo být v knihovně implementováno maticové násobení, použil jsem knihovnu **koma** (celým názvem Kotlin math), která implementuje základy lineární algebry v Kotlinu. Knihovna je pro JVM, Javascript i pro binární kód, avšak ve Windows ji nelze zkompileovat, proto naše knihovna funguje pouze pro JVM a Javascript. [[online:Koma](#)]

Asociativní paměť se mi bohužel nepodařilo doprogramovat do konce, natož otestovat, tedy není uvedena dále v této kapitole. Implementaci neuronu nalezneme v **core** (viz níže) jako třídu **Neuron**, která se stará přímo o výpočty z kapitoly 2. Uchovávání těchto neuronů a spouštění výpočtů na každém z nich má na starosti třída **AssociativeMemory** taktéž z **core**.

3 Struktura knihovny

Knihovna je rozdělena do dvou balíčků:

- První, a ten hlavní, je `core` (česky jádro), které obsahuje definice neuronových sítí (tj. konvoluční neuronovou síť, obyčejnou neuronovou síť, asociativní paměť) a definice pro ně potřebné (například aktivační funkce). Právě v tomto balíčku je implementováno to, co bylo v teoretické části.
- Druhý je `mnistDatabase`, která se stará o učení neuronových sítí na datech z databází ve formátu MNIST. O databázích ve formátu MNIST se v teorii nepsalo, jsou zmíněny až přímo v sekci 3.2, která pojednává o tomto balíčku.

3.1 `core`

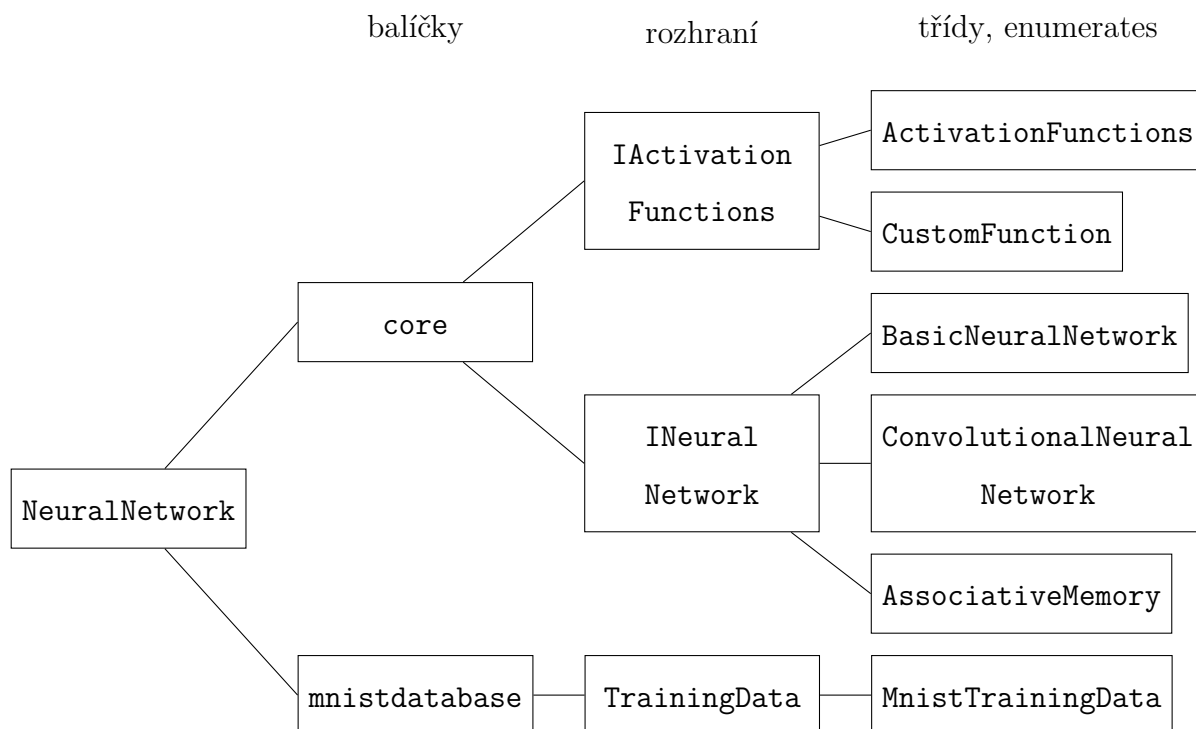
3.1.1 `IActivationFunctions`

Rozhraní, které zahrnuje `ActivationFunction` a `CustomFunction`. Jeho instance se používají jako aktivační funkce. Funkce lze zavolat s parametrem typu `Double`, což nám dá hodnotu funkce v tomto bodě, popřípadě lze obdobně zavolat jejich dvě metody `xD` a `yD` udávající v pořadí hodnotu derivace v bodě `x` a v bodě, kde je funkční hodnota rovna parametru.

3.1.2 `ActivationFunctions`

Enumerate častých funkcí, jež se používají jako aktivační funkce v neuronech. Některé jsou označeny jako překonané (anglicky `deprecated`), jelikož u funkcí, které nejsou všude hladké, neexistuje všude derivace. Taktéž u funkcí, jež nejsou prosté, nelze vždy určit derivaci podle funkční hodnoty.

Implementovány jsou všechny funkce uvedené v sekci 2.6



Obr. 3.1: Struktura knihovny

3.1.3 CustomFunction

Poskytuje možnost implementovat si vlastní aktivační funkci, má stejné metody (zde jsou to vlastnosti typu `() -> Unit`) jako `ActivationFunctions`.

3.1.4 INeuralNetwork

Rozhraní, které implementuje základní funkce neuronových sítí, které mají jako vstup i výstup vektor `Double`. Obsahuje funkce:

- `run(vstupní vektor)`, která je koncipována tak, aby ze vstupního vektoru spočítala vektor výstupní (tedy většinou udělala dopřednou propagaci). Jako vstupní vektor lze dát jak `Matrix<Double>` z knihovny `koma`, tak `DoubleArray`, které je převedeno na `Matrix<Double>`, následně se zavolá funkce `run` s tímto typem a výstup se převede zpět na `DoubleArray`.¹

Navíc (hlavně kvůli konvolučním neuronovým sítím) může být vstup i dvourozměrný, v tomto případě je pak nutno u `DoubleArray` uvést i šířku řádku.

¹`DoubleArray` je použito, protože je to typ Kotlinu samotného, ale jelikož matematika v neuronových sítích je implementována pomocí `Matrix<Double>`, musí se převést mezi typy.

- `train(vstupní vektor, chtěný výstupní vektor)` resp. `train(vstupní vektory, chtěné výstupní vektory)`, která je koncipována tak, aby nejdříve provedla `run(vstupní vektor)`, výsledek porovnála s chtěným a přepočítala váhy v neuronové síti tak, aby se výstup `run(vstupní vektor)` přiblížil (zmenšila se velikost jejich rozdílu) chtěnému výstupnímu vektoru. Kromě verze s parametry typu `DoubleArray` je funkce implementována i pro typ `Array<DoubleArray>`, tedy trénovací vstupy a výstupy lze vložit i všechny najednou.

3.1.5 BasicNeuralNetwork

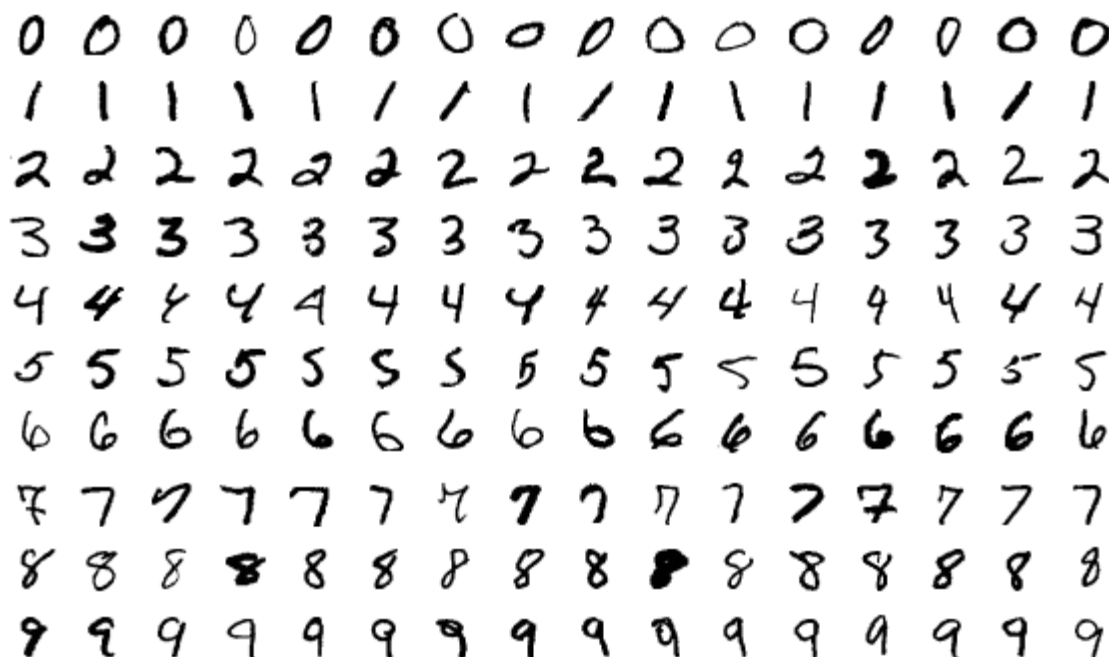
Tato třída rozhraní `INeuralNetwork` implementuje nejčastěji používanou neuronovou síť, kde jsou neurony uspořádány do vrstev a ovlivňují se pouze jedním směrem. Parametry, které lze nastavit, jsou:

- `numberOfHiddenLayers`, neboli počet skrytých vrstev (tj. ty, jež jsou mezi vstupní a výstupní vrstvou). Čím více vrstev je nastaveno, tím hůře se síť učí, většinou je proto třeba nastavit pouze jednu skrytou vrstvu nebo nastavit velmi malou hodnotu `learning rate` (proměnná, jež není v konstruktoru, která udává rychlost změn vah).
- `activationFunctions`, česky aktivační funkce, musí být vybrána z třídy `ActivationFunction`. Při použití funkcí, které nejsou hladké, se neurony mohou chovat nepředvídatelným způsobem.

3.1.6 ConvolutionalNetwork

Tato třída rozhraní `INeuralNetwork` implementuje konvoluční neuronové síť. Její konstruktor přijímá dva parametry typu `BasicNeuralNetwork`, první je filtr, druhá je samotná neuronová síť. Dalším parametrem je logická hodnota, zda se má i filtr učit (to se ale téměř nepoužívá, takže je tato hodnota při neuvedení nastavena na `false`).

Companion object této třídy navíc obsahuje příklad takového filtru (jednoduchý filtr detekující hrany viz obrázek 3.2)



Obr. 3.3: Příklad obrázků z databáze MNIST [wiki:MNIST]

3.2.1 Databáze MNIST

„Databáze MNIST, dataset ručně psaných číslic dostupná na stránkách <http://yann.lecun.com/exdb/mnist/> obsahuje 60 000 tréninkových a 10 000 ověřovacích příkladů. MNIST vychází z databáze spravované NIST (National Institute of Standards and Technology). Číslice mají normalizovanou velikost a jsou vycentrované v obrázcích shodné velikosti.“ [online:MNIST] Ukázku takových obrázků vidíme na obrázku 3.3.

Tuto databázi jsem použil pro první testování své `BasicNeuralNetwork`, jelikož má pro první testování dostačující velikost. Pro pozdější testování využívám převážně EMNIST.

3.2.2 Databáze EMNIST

„Databáze MNIST se stala standardem pro učení umělého vidění. Databáze MNIST je odvozená z databáze NIST Special Database 19, která obsahuje ručně psané číslice a velká i malá písmena. EMNIST (Extended MNIST), varianta celé databáze NIST, přebírá uspořádání z databáze MNIST².“ [article:EMNIST]

Tato databáze obsahuje více příkladů než MNIST, navíc obsahuje i sady s písmeny, proto jsem po prvních pokusech s MNIST přešel na tuto databázi.

²Má však prohozené řádky a sloupce pixelů v obrázcích.

4 Používání knihovny

4.1 Trénování sítě

Příklad takového tréninku je v souboru `NeuralNetworkTestJVM` funkce `mnist()`. Takové trénování ale trvá více než deset minut (konkrétně tato funkce běží asi tři čtvrtě hodiny), tudíž ho nelze zahrnout do testů. V testech je pouze trénování malinké sítě, aby fungovala jako xor.

Nejprve musíme neuronovou síť natrénovat a uložit. Trénování neuronové sítě probíhá za pomoci funkce `train`. Té musíme poskytovat tréninkové vstupy s odpovídajícími výstupy, což můžeme udělat tak, že funkci `train` budeme volat z cyklu, který bude tato data postupně načítat. Dalším způsobem je předat rovnou celý `Array` vstupů a výstupů, to ale často znamená načíst miliony objektů třídy `Double`, proto to může výrazně zpomalit učení. Poslední možností (pokud máme data ve formátu MNIST) je využít třídy `TrainingData`, které poskytneme soubory s daty a ona vytvoří příslušné objekty typu `Double` až ve chvíli, kdy dojde na danou dvojici vstup – výstup.

Dobré je také během učení pomalu snižovat `learningRate`, jelikož nejdříve se neuronová síť vlastně učí hlavně konkrétní obrázky (v této fázi nejlépe poznává obrázky, které dostala v tréninku naposledy), poté ale umí čím dál více věcí a nechceme, aby se přepisovaly již nabyté vědomosti. Já jsem například trénoval síť desetkrát na stejných datech (to není úplně vhodné, data by se měla měnit, aby se co nejméně naučila konkrétní obrázky¹, ale pro jednoduchost to stačí) s tím, že pokaždé jsem `learningRate` vydělil 1,5.

Poté už můžeme síť hned používat (například ji otestovat), ale většinou ji chceme používat víckrát a třeba i v rámci jiného programu. Proto mají třídy rozhraní `INeuralNetwork` funkci `save`, která vrátí data neuronové sítě jako řetězec (takový „osekaný“ JSON), který je pak možno uložit. V JVM je přímo definována funkce `saveFile(název souboru, data)`.

¹Při opakování malého datasetu se může stát, že neuronová síť bude umět rozpoznat jen obrázek, který je na pixel přesně shodný s tréninkovými obrázky.

4.2 Používání sítě

Ukázka načítání sítě je v programu `JSTest2` řádek 43 až 45 a ukázka výpočtu je parametr funkce `evaluateButton.addEventListener`. Můžete si všimnout, že použití je v rámci jednotek řádků kódu, zbytek se pouze stará o uživatelský vstup (program funguje jak za pomoci klikání myši, tak v mobilu pomocí dotyku).

Jakmile máme nějakou síť natrénovanou a uloženou v řetězci, můžeme ji znovu nahrát pomocí funkce `load(data)` nacházející se v companion objectu třídy `BasicNeuralNetwork` nebo `ConvolutionalNeuralNetwork`. Návratovou hodnotou této funkce je samotná neuronová síť, takže ji stačí uložit do proměnné, na které pak zavoláme funkci `run` s vstupním vektorem jako parametrem a tím získáme výstupní vektor, který stačí už jen zpracovat (např. při rozpoznávání číslic to znamená zjistit, který z výsledných 10 neuronů vysílá největší výstupní signál).

4.3 Nastavení hodnot

Neuronová síť má mnoho hodnot, které lze nastavit. Knihovna je vyzkoušena na rozpoznávání čísel v databázích MNIST a EMNIST s následujícím nastavením hodnot:

- Learning rate na 0.1 a každou z 10 epoch (1 epocha = 1 průchod přes všechny obrázky) se snižuje na $\frac{2}{3}$ původní hodnoty.
- Počet skrytých vrstev na jedna (dvě už se nenaučí propojit vstup s výstupem a bez skryté vrstvy síť vůbec nefunguje²).
- Počet neuronů ve skryté vrstvě na 100 (snížení počtu neuronů výsledky zhorší, zvýšení na 200 až 300 výsledky moc nezlepší, navíc trénování větší sítě zabere mnohem více času).
- Aktivační funkce na sigmoidu (jiné jsem moc nezkoušel, sigmoid stačí).

²Pokud byste potřebovali učit síť s více skrytými vrstvami, musíte nastavit learning rate na daleko nižší hodnotu a učit síť daleko déle a na více vstupech.

Závěr

Cílem mé práce bylo implementovat neuronovou síť, což se mi podařilo dokonce do takové míry, že v programu, kde zabírá pár řádků, je schopna rozeznávat číslíce (ukázka je na stránkách moznabude.cz, nebo na přiloženém USB). Největším přínosem je asi třída `BasicNeuralNetwork`, která implementuje velkou část matematiky obtížnou na rozmyšlení a stojící za téměř všemi neuronovými sítěmi, o niž se programátor v Kotlinu díky mojí knihovně už nemusí starat.

Zároveň jsem si díky rozdělení do balíčků a využití možností objektově orientovaného programování připravil dobrý podklad pro rozšiřování knihovny. Dále bych mohl pokračovat například implementováním lepšího ukládání do souboru (ukládání typu `Double` jako textového řetězce není moc efektivní), implementování některých genetických algoritmů, či naprogramování konvoluční sítě tak, aby filtry mohly pracovat n rozměrně.

Pro mě samotného byl asi největší přínos, že jsem si poprvé zkusil napsat formálnější kód a to jak v Kotlinu, tak i v LaTeXu. Navíc, už jen rozmyšlení si, co má tento text obsahovat byla pro mě velká životní zkušenost.

Slovníček pojmů

Array typ v Kotlinu odpovídající tzv. polím či vektorům v jiných programovacích jazycích, uchovává uspořádanou množinu objektů. 31, 34, 37

DoubleArray Array pro typ `Double`. 30–32

Double implementace 64bitových čísel s plovoucí desetinnou čárkou v Kotlinu. 30, 34, 36, 37

Pair typ v Kotlinu obsahující dvě vlastnosti `first` a `second`, dva objekty libovolného typu. 32

false opak `true`, většinou reprezentován 0. 31, 37

true hodnota typu `Boolean` (typ nabývající hodnot `true` a `false`) udávající pravdu, většinou reprezentován 1. 32, 37

axon výběžek vedoucí signál z neuronu. 10, 15, 38

balíček (anglicky `package`) je něco jako složka, používá se k izolování proměnných, funkcí, tříd a rozhraní, které se dají nastavit na použití pouze v daném balíčku, a zároveň také udává samostatné části programu nebo knihovny, které jsou na sobě téměř nezávislé. 29, 32

companion object tzv. statická část třídy v Kotlinu odpovídající modifikátoru `static` v Javě, obsahuje funkce a vlastnosti, které má třída i bez instance. 31, 35

cyklus pojem z teorie grafů, cyklus je posloupnost vrcholů (zde neuronů), přičemž z každého vrcholu do dalšího a z posledního do prvního vede hrana (zde `axon` → `dendrit`), tzn. pokud graf nemá cykly, nemůžeme se do vrcholu dostat vícekrát (zde nemusíme ho počítat vícekrát)

pojem z programování, používá se pro to, aby počítač opakoval kód. 11, 34

dendrit výběžek vedoucí signál do neuronu. 10, 11, 15, 38

enumerate česky výčet, typický prvek Javy či Kotlinu, třída, která má přesně definované instance (např. dny v týdnu by se implementovali jako enumerate). 29, 30, 38

gradient vektor derivací funkce podle jednotlivých proměnných, v našem světě si ho můžeme představit jako vodorovnou šipku (v každém bodě světa), která ukazuje, kterým směrem a jak moc jde krajina nejvíce do kopce z tohoto bodu (proměnné jsou pro tento příklad vodorovné souřadnice, funkcí je výška, braná třeba od moře). 12, 17, 21

JSON zkratka JavaScript Object Notation, lidsky čitelný formát ukládání Javascriptových objektů, každý parametr objektu se uloží jako „nazev”: hodnota“ a celý objekt je obalený „{ }“. 34

JVM Java Virtual Machine je virtuální stroj, který umožňuje běh Java Bytecodu, kódu, do kterého se překládá Java a Kotlin. 8, 28, 32, 34

Kotlin programovací jazyk vyvíjený firmou JetBrains, založen na Javě. 3, 4, 8, 28, 30, 32, 36–38

rozhraní (anglicky interface) je v objektově orientovaném programování zabalení funkcí a vlastností třídy, které by měla každá třída z nějaké skupiny mít (např. každá fronta by měla mít funkci pro přidání a odebrání prvku a jedna z jejích vlastností je velikost). 29–31, 34, 37, 38

synapse spojení (mezera) mezi axonem a dendritem, jež podle svých chemických vlastností zesílí nebo zeslabí signál předávaný z axonu do dendritu. 10–12, 15

typ třída nebo rozhraní, jehož instancí je daný objekt. 29–32, 34, 36, 37

třída (anglicky class) je základní prvek objektově orientovaného programování. Obsahuje funkce a vlastnosti, které bude mít objekt, který se vytvoří z dané třídy (popřípadě třídy, jež budou z této třídy dědit). 28, 30–32, 34–39

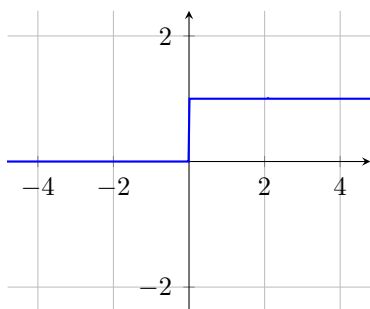
xor tzv. výlučné nebo, neboli binární (tj. přijímá dvě hodnoty / tvrzení) logická funkce, která je pravda právě tehdy, když jedno tvrzení je pravdivé a jedno nepravdivé. 34

Seznam obrázků

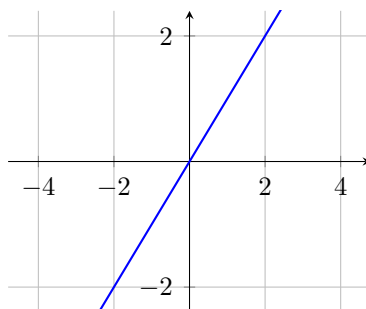
1.1	Běžná neuronová síť (W jsou váhy, n neurony a v je výstupní signál, viz kapitola 2, konkrétně sekce 2.5)	12
1.2	Asociativní paměť, červeně jsou vybuzené neurony	13
1.3	Generovaná tvář [online:Face]	14
2.1	Neuron	15
2.2	Binární krok	21
2.3	Identita	22
2.4	σ	22
2.5	Hyperbolický tangens	23
2.6	Swish	23
2.7	Rectified linear unit	24
2.8	Hard hyperbolic function	24
2.9	Leaky rectified linear unit pro koeficient úměrnosti 0.1	25
2.10	Soft funkce	25
2.11	Sinus	26
3.1	Struktura knihovny	30
3.2	Ilustrace filtru z třídy ConvolutionalNetwork, vstupem je matice 3×3 pixely, ta se po složkách násobí vždy s 1 z 8 matic výše, sečtou se všechny prvky výsledné matice, aplikuje se <i>rectified linear unit</i> a každé z výsledných 8 čísel pak udává, jak moc je v původní matici hrana odpovídající dané matici výše (tzn. jak moc je pixel násobený 1 bílý a pixel násobený -1 černý)	32
3.3	Příklad obrázků z databáze MNIST [wiki:MNIST]	33

Přílohy

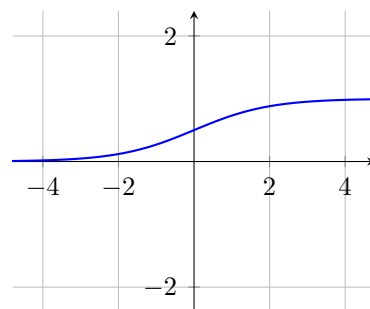
1. Zdrojový kód knihovny (složka NeuralNetwork)
2. Dokumentace (složka Dokumentace)
3. Testovací dataset (složka MNIST)
4. Zdrojový kód ukázkového programu (složka JSTest2)
5. Ukázkový program (soubor JSTest2/Main.html)
6. Zdrojový kód práce v \LaTeX u (složka LaTeX)
7. Přehled grafů aktivačních funkcí (následující stránky)
8. Zdrojový kód knihovny (následující stránky)



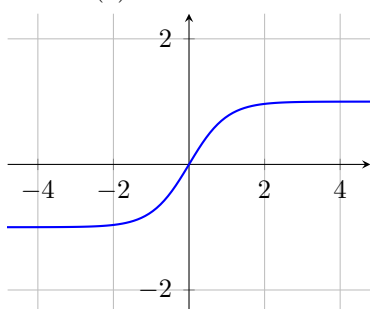
(a) Binární krok



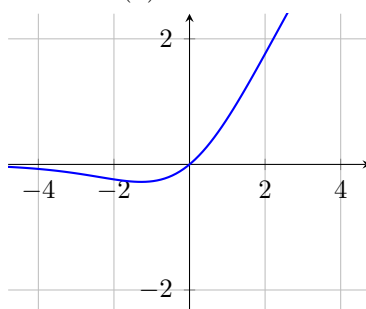
(b) Identita



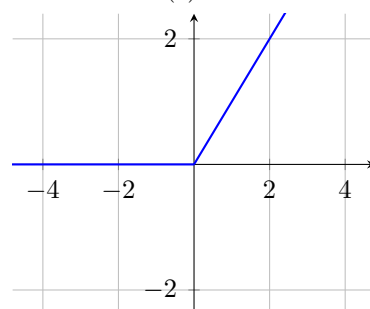
(c) σ



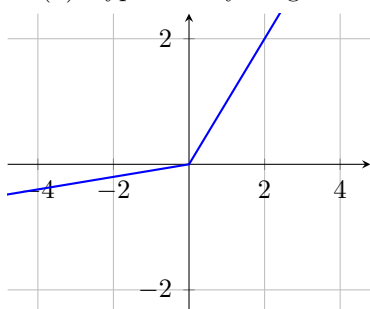
(d) Hyperbolický tangens



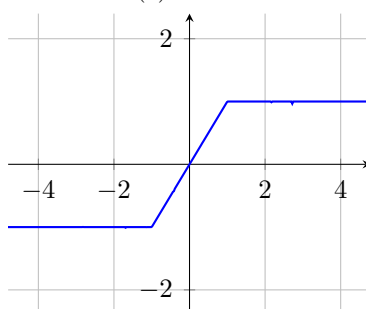
(e) Swish



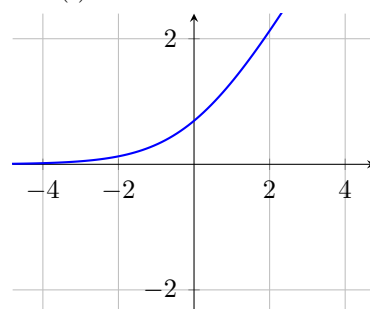
(f) Rectified linear unit



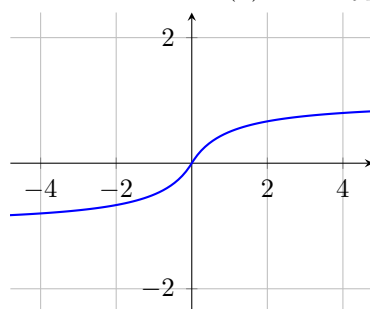
(g) Leaky rectified linear unit



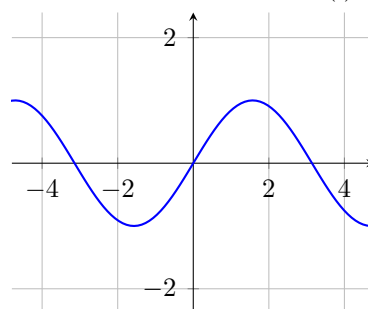
(h) Hard hyperbolic function



(i) Soft plus



(j) Soft signum



(k) Sinus

Přehled grafů aktivačních funkcí

Listing 1: src/commonMain/kotlin/core/ActivationFunctions.kt

```

1  /*
2  * file:                ActivationFunctions.kt
3  * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4  * project:             NeuralNetwork - Kotlin library for NNs
5  * content:             enumerate class ActivationFunctions
6  * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7  */
8  package core
9
10 import kotlin.math.*
11
12 /**
13  * Enumerate of many common functions ([invoke] returns f(x)), with it's derivation (f'(x)) for 2 cases: when we have x - [xD] or
14  * ↪ when we have f(x) - [yD]
15  *
16  * @param[xD] Derivation (f'(x)) when we have x value
17  * @param[yD] Derivation (f'(x)) when we have y = f(x) value
18  */
19 enum class ActivationFunctions(
20     private val function: (Double) -> Double,
21     override val xD: (Double) -> Double,
22     override val yD: (Double) -> Double
23 ) : IActivationFunctions {
24     /**
25      * Zero for negative values, one for others.
26      */
27     @Deprecated("This function isn't smooth", level = DeprecationLevel.WARNING)
28     BinaryStep({
29         if (it < 0) {
30             0.0
31         } else {
32             1.0
33         }
34     }, {
35         if (it == 0.0) {
36             Double.POSITIVE_INFINITY
37         } else {
38             0.0
39         }
40     }, { 0.0 })),
41     /**
42      * Identity for -1 <= x <= 1, -1 for x < -1 and 1 for x > 1
43      */
44     @Deprecated("This function isn't smooth", level = DeprecationLevel.WARNING)
45     HardHyperbolicFunction({
46         when {
47             it < -1 -> -1.0
48             it > 1 -> 1.0
49             else -> it
50         }
51     }, {
52         when {
53             it < -1 || it > 1 -> {
54                 0.0
55             }
56             it == -1.0 || it == 1.0 -> {
57                 Double.NaN
58             }
59             else -> {
60                 1.0
61             }
62         }
63     }, {
64         if (it == -1.0 || it == 1.0) {

```

```

65         0.0
66     } else {
67         1.0
68     }
69 }},
70
71 /**
72  * Zero for negative x, identity for positive x
73  */
74 @Deprecated("This function isn't smooth", level = DeprecationLevel.WARNING)
75 RectifiedLinearUnit({ max(0.0, it) }, {
76     when {
77         it < 0 -> {
78             0.0
79         }
80         it == 0.0 -> {
81             Double.NaN
82         }
83         else -> {
84             1.0
85         }
86     }
87 }, {
88     if (it == 0.0) {
89         0.0
90     } else {
91         1.0
92     }
93 }},
94
95 /**
96  * Identity for positive x, scaled identity ([ALPHA] * x) for negative x
97  */
98 @Deprecated("This function isn't smooth", level = DeprecationLevel.WARNING)
99 LeakyRectifiedLinearUnit({
100     if (it < 0) {
101         ALPHA * it
102     } else {
103         it
104     }
105 }, {
106     when {
107         it < 0 -> {
108             ALPHA
109         }
110         it == 0.0 -> {
111             Double.NaN
112         }
113         else -> {
114             1.0
115         }
116     }
117 }, {
118     if (it < 0.0) {
119         ALPHA
120     } else {
121         1.0
122     }
123 }},
124
125 /**
126  *  $f(x) = x$ 
127  */
128 Identity({
129     it
130 }, {
131     1.0

```

```

132     }, {
133         1.0
134     }},
135
136     /**
137      * Smooth step:  $f(x) = 1 / (1 + e^{-x})$ 
138      */
139     Sigmoid({
140         1 / (1 + exp(-it))
141     }, {
142         val expIt = exp(-it)
143         expIt / (1 + expIt).pow(2)
144     }, {
145         it * (1 - it)
146     }},
147
148     /**
149      * Hyperbolic tangents
150      */
151     Tanh({
152         tanh(it)
153     }, {
154         1 / cosh(it).pow(2)
155     }, {
156         1 - it.pow(2)
157     }},
158
159     /**
160      * Sign with smoothing ( $x / (|x| + 1)$ )
161      */
162     Softsign({
163         it / (abs(it) + 1)
164     }, {
165         1 / (1 + abs(it)).pow(2)
166     }, {
167         (1 - abs(it)).pow(2)
168     }},
169
170     /**
171      * [RectifiedLinearUnit] with smoothing ( $\ln(1 + \exp(x))$ )
172      */
173     Softplus({
174         ln(1 + exp(it))
175     }, {
176         1 / (1 + exp(-it))
177     }, {
178         1 / (2 - exp(it))
179     }},
180
181     /**
182      * Identity for positive x, scaled exponential ( $[ALPHA] * \exp(x) - 1$ ) for negative x
183      */
184     ExponentialLinearUnit({
185         if (it > 0) {
186             it
187         } else (ALPHA * exp(it) - 1)
188     }, {
189         if (it > 0) {
190             1.0
191         } else (ALPHA * (exp(it) + 1) - 1)
192     }, {
193         if (it > 0) {
194             1.0
195         } else (it + ALPHA)
196     }},
197
198     /**

```

```

199     * x * [Sigmoid] (x / (1 + exp(-x)))
200     */
201     Swish({
202         it / (1 + exp(-it))    //it * Sigmoid(it)
203     }, {
204         val expIt = exp(-it)
205         1 / (1 + expIt) + it * expIt / (1 + expIt).pow(2)
206     }, {
207         TODO("WTF?")
208     }),
209
210     /**
211     * [Sigmoid] for negative x, [ExponentialLinearUnit] for positive x and 0
212     */
213     ExponentialLinearSquashing({
214         if (it < 0) (Sigmoid(it)) else (ExponentialLinearUnit(it))
215     }, {
216         if (it < 0) (Sigmoid.xD(it)) else (ExponentialLinearUnit.xD(it))
217     }, {
218         if (it < 0) (Sigmoid.yD(it)) else (ExponentialLinearUnit.yD(it))
219     }),
220
221     /**
222     * ??? for negative x, [ExponentialLinearUnit] for positive x and 0
223     */
224     HardExponentialLinearSquashing({
225         if (it < 0) ((exp(it) - 1) * max(0.0, min(1.0, (it + 1) / 2))) else (ExponentialLinearUnit(it))
226     }, {
227         if (it < 0) (Sigmoid.xD(it)) else (ExponentialLinearUnit.xD(it))
228     }, {
229         TODO("WTF~2")
230     }),
231
232     /**
233     * Simply sinus
234     */
235     Sinus({
236         sin(it)
237     }, {
238         cos(it)
239     }, {
240         sqrt(1 - it.pow(2))
241     })
242     ;
243
244     override operator fun invoke(double: Double) = function(double)
245
246     companion object {
247         const val ALPHA = 1.0
248     }
249 }

```

Listing 2: src/commonMain/kotlin/core/BasicNeuralNetwork.kt

```

1  /*
2   * file:                BasicNeuralNetwork.kt
3   * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4   * project:             NeuralNetwork - Kotlin library for NNs
5   * content:             class BasicNeuralNetwork
6   * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7   */
8  package core
9
10 import koma.create
11 import koma.extensions.map
12 import koma.matrix.Matrix
13 import koma.rand
14 import koma.zeros
15 //import kotlin.math.abs
16 import kotlin.math.sqrt
17
18 /**
19  * Basic Neural Network consisted only of some layers of neurons.
20  *
21  * @constructor creates new [BasicNeuralNetwork] with
22  * * [numberOfHiddenLayers] hidden layers, which have sizes generated by [sizes]
23  * * neurons activating given by [activationFunction]
24  * * input for [inputLayerSize] [Double]s
25  * * answering with [outputLayerSize] [Double]s
26  *
27  * @param [numberOfHiddenLayers] number of hidden layers (without input and output layers)
28  * @param [activationFunction] how neurons are activated
29  * @param [sizes] sizes of hidden layers
30  * @param [inputLayerSize] size of input
31  * @param [outputLayerSize] size of output
32  * @param [weights] list of matrices, which state weights of connections between neurons in previous layer and neurons in next one
33  */
34 class BasicNeuralNetwork(
35     private val numberOfHiddenLayers: Int,
36     val activationFunction: IActivationFunctions = ActivationFunctions.Sigmoid,
37     val sizes: (Int) -> Int = { numberOfHiddenLayers },
38     val inputLayerSize: Int = numberOfHiddenLayers,
39     val outputLayerSize: Int = numberOfHiddenLayers,
40     private val weights: MutableList<Matrix<Double>> = MutableList(numberOfHiddenLayers + 1) {
41         if (numberOfHiddenLayers == 0) {
42             rand(outputLayerSize, inputLayerSize)
43         } else when (it) {
44             0 -> rand(sizes(it), inputLayerSize) * (sqrt(2.0 / (sizes(it) + inputLayerSize)))
45             numberOfHiddenLayers -> rand(
46                 outputLayerSize,
47                 sizes(it - 1)
48             ) * (sqrt(2.0 / (outputLayerSize + sizes(it - 1))))
49             else -> rand(sizes(it), sizes(it - 1)) * (sqrt(2.0 / (sizes(it) + sizes(it - 1))))
50         }
51     },
52     private val biases: MutableList<Matrix<Double>> = MutableList(numberOfHiddenLayers + 1) {
53         //rand(if (it == numberOfHiddenLayers) { outputLayerSize } else { sizes(it) }, 1)
54         zeros(
55             if (it == numberOfHiddenLayers) {
56                 outputLayerSize
57             } else {
58                 sizes(it)
59             }, 1
60         )
61     },
62     private val values: MutableList<Matrix<Double>> = MutableList(numberOfHiddenLayers + 2) {
63         zeros(
64             when (it) {
65                 0 -> inputLayerSize

```

```

66         numberOfHiddenLayers + 1 -> outputLayerSize
67         else -> sizes(it)
68     }, 1
69 )
70 }
71 ) : INeuralNetwork {
72
73     /**
74      * [Double] value which declares how quickly weights and biases are changing
75      */
76     var learningRate = 0.1
77
78     override fun run(input: Matrix<Double>): Matrix<Double> {
79         require(inputLayerSize == input.size) { "Wrong size of input! This NN has input size $inputLayerSize, but you offer it
80             ↳ input with size ${input.size}." }
81         values[0] = input
82         for (index in weights.indices) {
83             values[index + 1] = (weights[index] * values[index] + biases[index]).map { activationFunction(it) }
84         }
85         return values.last()
86     }
87
88     override fun train(input: Matrix<Double>, output: Matrix<Double>) = train(output - run(input))
89
90     fun train(er: Matrix<Double>): Matrix<Double> {
91         var error = er
92         for (i in numberOfHiddenLayers downTo 0) {
93             val derivations = values[i + 1].map { activationFunction.yD(it) }.elementTimes(error)
94             biases[i] += derivations * learningRate
95             error = weights[i].T * derivations
96             weights[i] += derivations * values[i].T * learningRate
97         }
98         return error
99     }
100
101     override fun save() =
102         when (activationFunction) {
103             is ActivationFunctions -> "$numberOfHiddenLayers;$activationFunction;${(0..numberOfHiddenLayers + 1).map(
104                 sizes
105             )};$inputLayerSize;$outputLayerSize;${weights.map { it.toList() }};${biases.map { it.toList() }}"
106             else -> TODO("It's hard to save unknown function")
107         }
108
109     companion object {
110         /**
111          * Load [BasicNeuralNetwork] from [data]
112          */
113         fun load(data: String): BasicNeuralNetwork {
114             val dataList = data.split(";")
115             val numberOfHiddenLayers = dataList[0].toInt()
116             val sizeList = dataList[2].removePrefix("[").removeSuffix("]").split(", ").map { it.toInt() }
117             val sizes: (Int) -> Int = { sizeList[it] }
118             val inputLayerSize = dataList[3].toInt()
119             val outputLayerSize = dataList[4].toInt()
120             return BasicNeuralNetwork(
121                 numberOfHiddenLayers,
122                 try {
123                     ActivationFunctions.valueOf(dataList[1])
124                 } catch (e: Exception) {
125                     TODO("It's hard to save unknown function")
126                 },
127                 sizes,
128                 inputLayerSize,
129                 outputLayerSize,
130                 dataList[5].removePrefix("[[").removeSuffix("]]").split(", ").mapIndexed
131                 { index, it ->

```

```

132         if (numberOfHiddenLayers == 0) {
133             create(
134                 it.split(", ").map { str -> str.toDouble() }.toDoubleArray(),
135                 outputLayerSize,
136                 inputLayerSize
137             )
138         } else when (index) {
139             0 -> create(
140                 it.split(", ").map { str -> str.toDouble() }.toDoubleArray(),
141                 sizes(index),
142                 inputLayerSize
143             )
144             numberOfHiddenLayers -> create(
145                 it.split(", ").map { str -> str.toDouble() }.toDoubleArray(),
146                 outputLayerSize,
147                 sizes(index - 1)
148             )
149             else -> create(
150                 it.split(", ").map { str -> str.toDouble() }.toDoubleArray(),
151                 sizes(index),
152                 sizes(index - 1)
153             )
154         }
155     }.toMutableList(),
156     dataList[6].removePrefix("[[[").removeSuffix("]]").split(", ").mapIndexed
157     { index, it ->
158         create(
159             it.split(", ").map { str -> str.toDouble() }.toDoubleArray(),
160             if (index == numberOfHiddenLayers) {
161                 outputLayerSize
162             } else {
163                 sizes(index)
164             },
165             1
166         )
167     }.toMutableList()
168 )
169 }
170 }
171 }

```


Listing 3: src/commonMain/kotlin/core/ConvolutionalNeuralNetwork.kt

```

1  /*
2  * file:                ConvolutionalNeuralNetwork.kt
3  * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4  * project:             NeuralNetwork - Kotlin library for NNs
5  * content:             class ConvolutionalNeuralNetwork
6  * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7  */
8
9  package core
10
11  import koma.create
12  import koma.extensions.*
13  import koma.matrix.Matrix
14  import koma.sqrt
15
16  /**
17   * Convolutional Neural Network consisted only of two [BasicNeuralNetwork].
18   *
19   * @constructor creates new [ConvolutionalNeuralNetwork] with
20   * * [filter] as small [BasicNeuralNetwork] that applies on every part of image before [neuralNetwork]
21   * * [neuralNetwork] as the main network
22   *
23   * @param [filter] small main network
24   * @param [neuralNetwork] main neural network
25   * @param [trainBoth] if filter should be trained
26   */
27  class ConvolutionalNeuralNetwork(
28      private val filter: BasicNeuralNetwork,
29      private val neuralNetwork: BasicNeuralNetwork,
30      private val trainBoth: Boolean = false
31  ) :
32      INeuralNetwork {
33
34      /**
35       * Size of one side of [filter]
36       */
37      private val filterSizeSqrt: Int
38
39      /**
40       * [Double] value which declares how quickly weights and biases are changing
41       */
42      var learningRate = 0.1
43
44      set(value) {
45          field = value
46          filter.learningRate = value
47          neuralNetwork.learningRate = value
48      }
49
50      init {
51          val s = sqrt(filter.inputLayerSize)
52          filterSizeSqrt = s.toInt()
53          require(s == filterSizeSqrt.toDouble()) { "Filter is not square" }
54          require(neuralNetwork.inputLayerSize % filter.outputLayerSize == 0) { "Filter is not for this neural network" }
55      }
56
57      /**
58       * Applies filter on every square of [input]
59       */
60      private fun runFilter(input: Matrix<Double>): Matrix<Double> {
61          val output = Matrix(
62              (input.numRows() - filterSizeSqrt + 1) * (input.numCols() - filterSizeSqrt + 1) * filter.outputLayerSize,
63              1
64          ) { _, _ -> 0.0 }
65          var offset = 0
66          for (i in 0 until input.numRows() - filterSizeSqrt + 1) {

```

```

66         for (j in 0 until input.numCols() - filterSizeSqrt + 1) {
67             val output1 = filter.run(input[i until i + filterSizeSqrt, j until j + filterSizeSqrt].toDoubleArray())
68             output1.forEachIndexed { it, ele ->
69                 output[offset + it] = ele
70             }
71             offset += output1.size
72         }
73     }
74     return output
75 }
76
77 override fun run(input: Matrix<Double>): Matrix<Double> {
78     val input2 = runFilter(input)
79     require(input2.size == neuralNetwork.inputLayerSize) { "Invalid input matrix size for neural network" }
80     return neuralNetwork.run(input2)
81 }
82
83 override fun train(input: Matrix<Double>, output: Matrix<Double>): Matrix<Double> {
84     val input2 = runFilter(input)
85     val error = neuralNetwork.train(input2, output)
86     return if (trainBoth) {
87         val error2 = Matrix(filter.outputLayerSize, 1) { _, _ -> 0.0 }
88         error.forEachIndexed { idx: Int, ele: Double -> error[idx % filter.outputLayerSize] += ele }
89         filter.train(error2.map { it * filter.outputLayerSize / error.size })
90     } else create(DoubleArray(0))
91 }
92
93 override fun save() = filter.save() + ";;" + neuralNetwork.save()
94
95 companion object {
96     /**
97      * Load [ConvolutionalNeuralNetwork] from [data]
98      */
99     fun load(data: String): ConvolutionalNeuralNetwork {
100         val nns = data.split(";;")
101         return ConvolutionalNeuralNetwork(BasicNeuralNetwork.load(nns[0]), BasicNeuralNetwork.load(nns[1]))
102     }
103
104     /**
105      * Data of [Matrix] for [edgeFilter]
106      */
107     private val edgeFilterData = mutableListOf(
108         mutableListOf(1.0, 1.0, 1.0, 0.0, 0.0, 0.0, -1.0, -1.0, -1.0),
109         mutableListOf(1.0, 0.0, -1.0, 1.0, 0.0, -1.0, 1.0, 0.0, -1.0),
110         mutableListOf(-1.0, -1.0, -1.0, 0.0, 0.0, 0.0, 1.0, 1.0, 1.0),
111         mutableListOf(-1.0, 0.0, 1.0, -1.0, 0.0, 1.0, -1.0, 0.0, 1.0),
112         mutableListOf(1.0, 1.0, 0.0, 1.0, 0.0, -1.0, 0.0, -1.0, -1.0),
113         mutableListOf(-1.0, -1.0, 0.0, -1.0, 0.0, 1.0, 0.0, 1.0, 1.0),
114         mutableListOf(0.0, 1.0, 1.0, 1.0, 0.0, -1.0, -1.0, -1.0, 0.0),
115         mutableListOf(0.0, -1.0, -1.0, -1.0, 0.0, 1.0, 1.0, 1.0, 0.0)
116     )
117
118     /**
119      * Example filter, detects edges
120      */
121     val edgeFilter: BasicNeuralNetwork
122     get() = BasicNeuralNetwork(
123         0, ActivationFunctions.RectifiedLinearUnit, { 0 }, 9, 8,
124         mutableListOf(Matrix(8, 9) { row: Int, cols: Int ->
125             edgeFilterData[row][cols]
126         })
127     )
128 }
129
130 }

```

Listing 4: src/commonMain/kotlin/core/CustomFunction.kt

```

1  /*
2   * file:                CustomFunction.kt
3   * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4   * project:             NeuralNetwork - Kotlin library for NNs
5   * content:             class CustomFunction
6   * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7   */
8
9  package core
10
11  class CustomFunction(
12      private val function: (Double) -> Double,
13      override val xD: (Double) -> Double,
14      override val yD: (Double) -> Double
15  ) : IActivationFunctions {
16      override fun invoke(double: Double): Double = function(double)
17  }

```

Listing 5: src/commonMain/kotlin/core/IActivationFunctions.kt

```

1  /*
2   * file:                IActivationFunctions.kt
3   * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4   * project:             NeuralNetwork - Kotlin library for NNs
5   * content:             interface IActivationFunctions
6   * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7   */
8
9  package core
10
11  /**
12   * Interface for activation functions for neural networks
13   */
14
15  interface IActivationFunctions {
16      /**
17       * Derivation (f'(x)) when we have x value
18       */
19      val xD: (Double) -> Double
20
21      /**
22       * Derivation (f'(x)) when we have y = f(x) value
23       */
24      val yD: (Double) -> Double
25
26      /**
27       * Returns functional value (f([double]))
28       */
29      operator fun invoke(double: Double): Double
30  }

```

Listing 6: src/commonMain/kotlin/core/INeuralNetwork.kt

```

1  /*
2  * file:                INeuralNetwork.kt
3  * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4  * project:             NeuralNetwork - Kotlin library for NNs
5  * content:             interface INeuralNetwork
6  * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7  */
8  package core
9
10 import koma.create
11 import koma.extensions.toDoubleArray
12 import koma.matrix.Matrix
13
14 /**
15  * Neural Network Interface
16  *
17  * Basic usage is train it by [train] and then use it by [run]
18  */
19 interface INeuralNetwork {
20
21     /**
22      * Takes input, process it throw neural network and returns Matrix vector of [Double] outputs
23      */
24     fun run(input: Matrix<Double>): Matrix<Double>
25
26     fun run(input: DoubleArray, numCols: Int = 1) = run(create(input, input.size / numCols, numCols))
27
28     /**
29      * Takes input and desired output, compute estimated output and apply backpropagation
30      */
31     fun train(input: Matrix<Double>, output: Matrix<Double>): Matrix<Double>
32
33     fun train(input: DoubleArray, output: DoubleArray, inNumCols: Int = 1, outNumCols: Int = 1) =
34         train(
35             create(input, input.size / inNumCols, inNumCols),
36             create(output, output.size / outNumCols, outNumCols)
37         ).toDoubleArray()
38
39     fun train(input: Array<DoubleArray>, output: Array<DoubleArray>) {
40         require(input.size == output.size) { "Wrong training sets! Size of input is ${input.size}, size of output is ${output.size}
41             ↪ }." }
42         for (i in input.indices) {
43             train(input[i], output[i])
44         }
45     }
46
47     /**
48      * Returns save of NN in [String]
49      */
50     fun save(): String
51 }

```

Listing 7: src/commonMain/kotlin/mnistDatabase/loadFile.kt

```

1  /*
2   * file:                loadFile.kt
3   * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4   * project:             NeuralNetwork - Kotlin library for NNs
5   * content:             fun loadFile, loadFileString, saveFile, interface TrainingData, class MnistTrainingData, extension function
6   *                       ↪ INeuralNetwork.train (for TrainingData)
7   * Licensed under the MIT License. See LICENSE file in the project root for full license information.
8   */
9
10 package mnistDatabase
11
12 import core.INeuralNetwork
13
14 expect fun loadFile(file: String): ByteArray
15 expect fun loadFileString(file: String): String
16 expect fun saveFile(file: String, text: String)
17
18 private fun Byte.toUInt() = (this.toInt() + 256) % 256
19 private fun Byte.toUNDouble() = ((this.toDouble() + 256.0) % 256.0) / 256
20
21 private fun List<Byte>.toIntArray(): IntArray {
22     require(size % 4 == 0)
23     val result = IntArray(size / 4)
24     for (i in indices) {
25         result[i / 4] += (this[i].toUInt() shl 8 * (when (i % 4) {
26             0 -> 3
27             1 -> 2
28             2 -> 1
29             3 -> 0
30             else -> 4
31         }))
32     }
33     return result
34 }
35
36 typealias TrainingData = Sequence<Pair<DoubleArray, DoubleArray>>
37
38 class MnistTrainingData(imageFile: String, numberFile: String, val inverse: Boolean) : TrainingData {
39     private val imageBytes = loadFile(imageFile.removeSuffix(".idx3-ubyte") + ".idx3-ubyte")
40     private val imageFirstInts = imageBytes.slice(4 until 16).toIntArray()
41     private val numberOfImages = imageFirstInts[0]
42     private val numberOfRows = imageFirstInts[1]
43     private val numberOfColumns = imageFirstInts[2]
44     private val sizeOfImage = numberOfColumns * numberOfRows
45
46     private val numberBytes = loadFile(numberFile.removeSuffix(".idx1-ubyte") + ".idx1-ubyte")
47
48     init {
49         require(numberOfImages == numberBytes.slice(4 until 8).toIntArray().first()) { "Error" }
50     }
51
52     override fun iterator(): Iterator<Pair<DoubleArray, DoubleArray>> {
53         return object : Iterator<Pair<DoubleArray, DoubleArray>> {
54             val data = this@MnistTrainingData
55             val indexes = (0 until numberOfImages).shuffled()
56             var index = 0
57             override fun hasNext() = index < numberOfImages
58
59             override fun next(): Pair<DoubleArray, DoubleArray> {
60
61                 var image =
62                     data.imageBytes.slice(16 + indexes[index] * sizeOfImage until 16 + (indexes[index] + 1) * sizeOfImage)
63                     .map { byte -> byte.toUNDouble() }.toDoubleArray()
64

```

```

65         if (inverse) {
66             val newimage = DoubleArray(sizeOfImage)
67             for (i in 0 until numberOfRows) {
68                 for (j in 0 until numberOfColumns) {
69                     newimage[i * 28 + j] = image[j * 28 + i]
70                 }
71             }
72             image = newimage
73         }
74
75         val position = numberBytes[8 + indexes[index]].toUInt()
76         val number = DoubleArray(10) {
77             if (it == position) {
78                 1.0
79             } else {
80                 0.0
81             }
82         }
83
84         index++
85         return image to number
86     }
87 }
88 }
89 }
90
91 fun INeuralNetwork.train(data: TrainingData, numCols: Int = 1) {
92     for ((input, output) in data) {
93         train(input, output, numCols)
94     }
95 }

```

Listing 8: src/commonTest/kotlin/sample/Constants.kt

```

1  /*
2   * file:           Constants.kt
3   * original author:  Jonáš Havelka <jonas.havelka@volny.cz>
4   * project:         NeuralNetwork - Kotlin library for NNs
5   * content:         constants for tests and defaultDoubleMatrixFactory
6   * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7   */
8
9  package sample
10
11  import koma.matrix.Matrix
12  import koma.matrix.MatrixFactory
13
14  const val wrongInputLayerSize = 1
15  const val numberOfHiddenLayers = 1
16  const val numberOfDigits = 10
17  const val imageWidth = 28
18  const val imageHeight = 28
19  const val blackFrom = 0.5
20  const val learningRateEpochDecrease = 1.5
21
22  val input: DoubleArray = DoubleArray(2) { 1.0 }
23  //get() = DoubleArray(2) { 1.0 }
24  val output: DoubleArray = input
25  //get() = input
26  val inputTest = input.copyOf()
27  val outputTest = output.copyOf()
28
29  expect val defaultDoubleMatrixFactory: MatrixFactory<Matrix<Double>>

```

Listing 9: src/commonTest/kotlin/sample/NeuralNetworkTest.kt

```

1  /*
2   * file:                NeuralNetworkTest.kt
3   * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4   * project:             NeuralNetwork - Kotlin library for NNs
5   * content:             tests
6   * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7   */
8
9  package sample
10
11  import core.BasicNeuralNetwork
12  import koma.create
13  import koma.matrix.Matrix
14  import kotlin.test.Test
15  import kotlin.test.assertFailsWith
16  import kotlin.test.assertTrue
17
18  class NeuralNetworkTest {
19
20      init {
21          Matrix.doubleFactory = defaultDoubleMatrixFactory
22      }
23
24      // @Test
25      fun inputs() {
26          assertFailsWith<IllegalArgumentException>("Wrong size of input! This NN has input size $wrongInputLayerSize, but you offer
27              ↪ it input with size ${input.size}.") {
28              val nn = BasicNeuralNetwork(numberOfHiddenLayers, inputLayerSize = wrongInputLayerSize)
29              nn.run(input)
30          }
31
32      // @Test
33      fun learning() {
34          val nn = BasicNeuralNetwork(numberOfHiddenLayers, inputLayerSize = input.size, outputLayerSize = output.size)
35          repeat(1000) {
36              nn.train(input, output)
37          }
38          assertTrue("Error of simple memory is bigger than 0.1") {
39              (nn.run(input) - create(
40                  output,
41                  output.size,
42                  1
43              )).elementSum() <= 0.1
44          }
45          assertTrue("Input changed (from $inputTest to $input)") { input.contentEquals(inputTest) }
46          assertTrue("Output changed (from $outputTest to $output)") { output.contentEquals(outputTest) }
47      }
48
49      // @Test
50      fun xor() {
51          val dataset = setOf(
52              DoubleArray(2) { listOf(0.0, 0.0)[it] } to DoubleArray(1) { 0.0 },
53              DoubleArray(2) { listOf(1.0, 0.0)[it] } to DoubleArray(1) { 1.0 },
54              DoubleArray(2) { listOf(0.0, 1.0)[it] } to DoubleArray(1) { 1.0 },
55              DoubleArray(2) { listOf(1.0, 1.0)[it] } to DoubleArray(1) { 0.0 }
56          )
57          val nn = BasicNeuralNetwork(
58              numberOfHiddenLayers,
59              inputLayerSize = dataset.random().first.size,
60              outputLayerSize = dataset.random().second.size,
61              sizes = { 2 })
62          repeat(50000) {
63              val (input, output) = dataset.random()
64              nn.train(input, output)

```



```
65     }
66     dataset.forEach {
67         println(it.first.toList())
68         println(it.second.toList())
69         println(nn.run(it.first).toList())
70     }
71 }
72 }
```

Listing 10: src/jsTest/kotlin/sample/ConstantsJS.kt

```

1  /*
2   * file:                ConstantsJS.kt
3   * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4   * project:             NeuralNetwork - Kotlin library for NNs
5   * content:             constant defaultDoubleMatrixFactory
6   * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7   */
8
9  package sample
10
11  import koma.internal.default.generated.matrix.DefaultDoubleMatrixFactory
12  import koma.matrix.Matrix
13  import koma.matrix.MatrixFactory
14
15  actual val defaultDoubleMatrixFactory: MatrixFactory<Matrix<Double>> = DefaultDoubleMatrixFactory()

```

Listing 11: src/jvmMain/kotlin/mnistDatabase/loadFileJVM.kt

```

1  /*
2   * file:                loadFileJVM.kt
3   * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4   * project:             NeuralNetwork - Kotlin library for NNs
5   * content:             fun loadFile, saveFile, loadFileString
6   * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7   */
8
9  package mnistDatabase
10
11  import java.io.*
12
13  actual fun loadFile(file: String) = File(file).readBytes()
14  actual fun saveFile(file: String, text: String) {
15      val f = File(file)
16      f.createNewFile()
17      val bw = BufferedWriter(Writer(f))
18      bw.append(text)
19      bw.close()
20  }
21
22  actual fun loadFileString(file: String): String = BufferedReader(FileReader(File(file))).readLine()

```

Listing 12: src/jvmTest/kotlin/sample/ConstantsJVM.kt

```

1  /*
2   * file:                ConstantsJVM.kt
3   * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4   * project:             NeuralNetwork - Kotlin library for NNs
5   * content:             Constants for tests and defaultDoubleMatrixFactory
6   * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7   */
8
9  package sample
10
11  import koma.internal.default.generated.matrix.DefaultDoubleMatrixFactory
12  import koma.matrix.Matrix
13  import koma.matrix.MatrixFactory
14  import mnistDatabase.MnistTrainingData
15
16  actual val defaultDoubleMatrixFactory: MatrixFactory<Matrix<Double>> = DefaultDoubleMatrixFactory()
17
18  val mnistDigitTrainingDataset = MnistTrainingData("train-images", "train-labels", false)
19  val emnistDigitTrainingDataset = MnistTrainingData("emnist-digits-train-images", "emnist-digits-train-labels", true)

```

Listing 13: src/jvmTest/kotlin/sample/NeuralNetworkTestJVM.kt

```

1  /*
2   * file:                NeuralNetworkTestJVM.kt
3   * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4   * project:             NeuralNetwork - Kotlin library for NNs
5   * content:             tests
6   * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7   */
8
9  package sample
10
11  import core.BasicNeuralNetwork
12  import core.ConvolutionalNeuralNetwork
13  import mnistDatabase.loadFileString
14  import mnistDatabase.saveFile
15  import mnistDatabase.train
16  import org.junit.Test
17
18  class NeuralNetworkTestJVM {
19      @Test
20      fun serialization() {
21          val nn = BasicNeuralNetwork(numberOfHiddenLayers, inputLayerSize = input.size, outputLayerSize = output.size)
22          val saved = nn.save()
23          println(nn.save())
24          val nn2 = BasicNeuralNetwork.load(saved)
25          println(nn2.save())
26          nn2.run(input)
27          nn2.train(input, output)
28      }
29
30      // @Test
31      fun mnist() {
32          val nn = BasicNeuralNetwork(
33              numberOfHiddenLayers,
34              inputLayerSize = imageWidth * imageHeight,
35              outputLayerSize = numberOfDigits,
36              sizes = { 100 })
37          repeat(10) {
38              nn.train(mnistDigitTrainingDataset)
39              nn.train(emnistDigitTrainingDataset)
40              nn.learningRate /= learningRateEpochDecrease
41
42              val data = mnistDigitTrainingDataset.iterator().next()
43              println(nn.run(data.first).toList())
44              println(data.second.toList())
45              saveFile("output.txt", nn.save())
46          }
47      }
48
49      // @Test
50      fun savedNN() {
51          var error = 0
52          repeat(100) {
53              val data = mnistDigitTrainingDataset.iterator().next()
54              val answer =
55                  BasicNeuralNetwork.load(loadFileString("output.txt")).run(data.first).toList()
56              if (answer.indexOf(answer.maxBy { it }) != data.second.indexOf(1.0)) {
57                  error++
58              }
59          }
60          println(error)
61      }
62
63      // @Test
64      fun mnistC() {
65          val nn = ConvolutionalNeuralNetwork(

```

```

66         ConvolutionalNeuralNetwork.edgeFilter,
67         BasicNeuralNetwork(
68             1,
69             inputLayerSize = (imageWidth - 2) * (imageHeight - 2) * 8,
70             outputLayerSize = numberOfDigits,
71             sizes = { 100 })
72     )
73     repeat(10) {
74         nn.train(mnistDigitTrainingDataset, imageWidth)
75         nn.train(emnistDigitTrainingDataset, imageWidth)
76         nn.learningRate /= learningRateEpochDecrease
77
78         saveFile("outputC.txt", nn.save())
79         savedNNC()
80     }
81 }
82
83 //@Test
84 fun savedNNC() {
85     var error = 0
86     repeat(100) {
87         val data = mnistDigitTrainingDataset.iterator().next()
88         val answer =
89             ConvolutionalNeuralNetwork.load(loadFileString("outputC.txt")).run(data.first, imageWidth).toList()
90         if (answer.indexOf(answer.maxBy { it }) != data.second.indexOf(1.0)) {
91             error++
92         }
93     }
94     println(error)
95 }
96
97 //@Test
98 fun print() {
99     fun Pair<DoubleArray, DoubleArray>.print() {
100         for (i in 0 until imageHeight) {
101             for (j in 0 until imageWidth) {
102                 print(
103                     if (first[j + i * imageWidth] < blackFrom) {
104                         "."
105                     } else {
106                         "#"
107                     }
108                 )
109             }
110             println()
111         }
112         println(second.indexOf(1.0))
113     }
114
115     val data = mnistDigitTrainingDataset.iterator().next()
116     data.print()
117 }
118 }

```

Listing 14: src/commonMain/kotlin/core/AssociativeMemory.kt

```

1  /*
2  * file:                AssociativeMemory.kt
3  * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4  * project:             NeuralNetwork - Kotlin library for NNs
5  * content:             class AssociativeMemory
6  * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7  */
8
9  package core
10
11  import kotlin.random.Random
12
13  /**
14   * Associative memory (neural network) consists of [Neuron]s
15   *
16   * @constructor creates new [AssociativeMemory] with
17   * * [neurons] map connect objects to [Neuron]s      OR:
18   * * [ideas] objects which stores Associative memory
19   * * [function] which of [IActivationFunctions] we want to use in [neurons]
20   *
21   * @param[neurons] map connect objects to [Neuron]s
22   */
23  class AssociativeMemory(private val neurons: MutableMap<Any, Neuron> = mutableMapOf()) {
24
25      constructor(ideas: Set<Any>, function: IActivationFunctions = ActivationFunctions.Sigmoid) : this() {
26          ideas.forEach { idea ->
27              neurons[idea] = Neuron(
28                  function,
29                  inputs = neurons.values.map { it to Random.nextDouble(1.0) }.toMutableList()
30              )
31          }
32      }
33
34      /**
35       * gets state after [repeat] times computing values of neurons
36       */
37      fun run(repeat: Int) {
38          repeat(repeat) {
39              neurons.values.forEach { it.prepareForStep() }
40              neurons.values.forEach { it.run() }
41          }
42      }
43
44      /**
45       * gets only next state of [AssociativeMemory]
46       */
47      fun nextStep() {
48          neurons.values.forEach { it.prepareForStep() }
49          neurons.values.forEach { it.run() }
50      }
51  }

```

Listing 15: src/commonMain/kotlin/core/Neuron.kt

```

1  /*
2  * file:                Neuron.kt
3  * original author:     Jonáš Havelka <jonas.havelka@volny.cz>
4  * project:             NeuralNetwork - Kotlin library for NNs
5  * content:             class Neuron
6  * Licensed under the MIT License. See LICENSE file in the project root for full license information.
7  */
8
9  package core
10
11  import kotlin.random.Random
12
13  /**
14   * Neuron for [AssociativeMemory]
15   *
16   * @param [function] activation function for neuron
17   * @param [bias] bias for neuron
18   * @param [inputs] [Pair]s of neurons and weights from them
19   */
20  class Neuron(
21      private val function: IActivationFunctions = ActivationFunctions.Sigmoid,
22      private var bias: Double = Random.nextDouble(1.0),
23      private val inputs: MutableList<Pair<Neuron, Double>> = mutableListOf()
24  ) {
25      var actualValue = 0.0
26      private set
27      private var lastValue = 0.0
28      var error = 0.0
29
30      /**
31       * computation of new value
32       */
33      fun run() {
34          actualValue = function(inputs.sumByDouble { it.first.lastValue * it.second } + bias)
35      }
36
37      /**
38       * trains neuron
39       */
40      fun train() {
41          val error2 = function.yD(actualValue) * error
42          bias -= error2
43          inputs.forEach { it.first.error += error2 * it.second }
44      }
45
46      /**
47       * prepares neuron for next computation
48       */
49      fun prepareForStep() {
50          lastValue = actualValue
51      }
52  }

```