Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Využití Deep Learning v medicínských aplikacích

Plzeň 2023 Pavel Košan

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Pavel KOŠAN**
Osobní číslo:	**A20B0541P**
Studijní program:	**B3902 Inženýrská informatika**
Studijní obor:	**Informační systémy**
Téma práce:	**Využití Deep Learning v medicínských aplikacích**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Popište současný stav Deep Learning.
2. Popište aktuálně nejzajímavější systémy.
3. Navrhněte využití Deep Learning pro analýzu různých typů medicínských dat.
4. Ověřte na vybrané kolekci dat.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce

Vedoucí bakalářské práce: **Doc. Dr. Ing. Jana Klečková**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **3. října 2022**
Termín odevzdání bakalářské práce: **4. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 25. října 2022

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 4. května 2023

Pavel Košan

Poděkování

Děkuji paní Doc. Dr. Ing. Janě Klečkové za odborné vedení, trpělivost a cenné rady při zpracování této práce.

Abstract

The aim of this work is to provide a basic description of deep learning and its possible use for the analysis of biomedical data.

First, the fundamental theoretical concepts and algorithms on which contemporary deep learning is based are explained, then the important classes and architectures of deep neural networks are presented with the possibilities of their application in medicine.

As part of the practical part, this work deals with three examples of deep learning applications for three different medical datasets. Special attention is paid to the last example which deals with segmentation of CT brain images.

Abstrakt

Cílem této práce je poskytnout základní popis hlubokého učení a jeho možná využití pro analýzu biomedicínských dat.

Nejprve jsou vysvětleny základní teoretické koncepty a algoritmy, na kterých je založeno současné hluboké učení, poté jsou představeny důležité třídy a architektury hlubokých neuronových sítí s možnostmi jejich aplikace v medicíně.

V rámci praktické části se tato práce zabývá třemi příklady aplikace hlubokého učení pro tři různé medicínské datové množiny. Zvláštní pozornost je věnována poslednímu příkladu, který se zabývá segmentací CT snímků mozku.

Obsah

1 Úvod

Umělá inteligence zaznamenala v posledních deseti letech značných úspěchů nejen ve vědě, průmyslu a podnikání, ale i v tak citlivé oblasti, jakou je medicína. Je to dáno zejména díky vzestupu hlubokého učení, tedy podoblasti strojového učení, která se zabývá technikami a algoritmy pro učení hlubokých neuronových sítí.

Přestože většina klíčových konceptů hlubokého učení byla známa již mnoho let, skutečný rozmach nastal až po roce 2010. Za tuto prodlevu mohou dva hlavní faktory: obrovská výpočetní náročnost, jakou hluboké učení vyžaduje, a potřeba velkého množství dat pro natrénování hlubokých neuronových sítí. Ovšem s masovým rozšířením počítačů a internetu, stále rostoucí výpočetní sílou, příchodem GPU a později TPU, které umožnili paralelizaci výpočtů, a zefektivněním trénovacích algoritmů se tato limitace hlubokého učení překlenula.

S dostupnými daty, algoritmy a výpočetní silou začalo hluboké učení pronikat do nejrůznějších oblastí. Zvlášť pozoruhodné úspěchy zaznamenalo v oblasti strojového vnímání, jako je počítačové vidění, rozpoznávání řeči či zpracování přirozeného jazyka, což jsou oblasti, které se před hlubokým učením považovaly za algoritmicky velmi obtížné a zároveň to jsou oblasti s velikým potenciálem pro medicínu.

Medicínský sektor, stejně jako většina ostatních sektorů, produkuje stále více dat. Jedná se například o medicínské snímky, záznamy elektrické aktivity mozku nebo záznamy srdeční aktivity. S přibývajícím množstvím dat, které lze o pacientovi získat, přirozeně vyvstává i otázka, jak tato data efektivně analyzovat a výstupy využít ke zkvalitnění lékařského rozhodování a poskytovaní služeb.

V této práci jsou definovány, rozebírány a analyzovány metody hlubokého učení a jejich využití v analýze biomedicínských dat. Jsou zde demonstrovány tři konkrétní příklady ukazující, jak lze hluboké učení použít pro lékařskou diagnostiku.

První dva příklady jsou určeny zejména pro ukázku konkrétních architektur na obrazových a sekvenčních datech.

Poslední, třetí příklad se zabývá segmentací CT snímků mozku pro dodanou datovou množinu.

2 Hluboké učení a umělé neuronové sítě

Tato kapitola pokrývá základní myšlenky a konstrukce, na kterých je postaveno hluboké učení. Definuje elementární algoritmy a modely a přibližuje jejich matematickou podstatu.

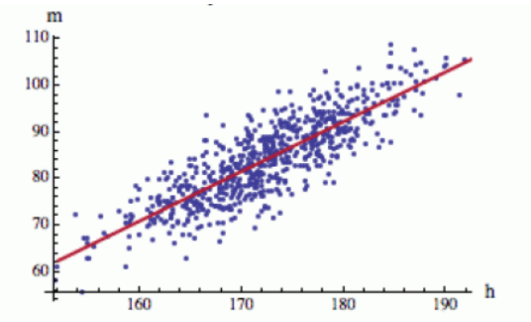
2.1 Strojové učení

Strojové učení je obecně podoblastí umělé inteligence zabývající se tvorbou systémů, které jsou schopny automaticky měnit svůj vnitřní stav na základě vstupních podnětů, čímž se přizpůsobují pro řešení daného problému.

Jde o vědní a inženýrskou oblast využívající matematické, statistické a informatické metody pro vývoj a analýzu algoritmů, jež umožňují učícím se systémům hledat takové modely, které dokážou dostatečně obecně popsat určitý problém a jsou aplikovatelné i na dosud neznámá data.

Modelem se v kontextu strojového učení rozumí konkrétní matematická reprezentace výstupu z procesu učení. Jde zpravidla o více či méně komplexní matematickou funkci mapující vstupní data na výstupy.

Na obrázku 2.1 je příklad natrénovaného modelu z algoritmu lineární regrese, který zachycuje vztah mezi výškou a váhou člověka.



Obrázek 2.1: Model zachycující vztah mezi výškou h a váhou m

Co vlastně vedlo k potřebě studia metod strojového učení? Když řešíme problémy pomocí tradičního vytváření expertních systémů, potřebujeme mít k dispozici dostatek znalostí o problému, abychom je mohli do systému explicitně naprogramovat. To je jeden ze zásadních důvodů vedoucích k rozvoji

strojového učení, protože shánění potřebných znalostí od expertů je zdlouhavé a nákladné.

Strojové učení přistupuje k problému jiným způsobem. Místo abychom do systému všechny znalosti explicitně vkládali, naprogramujeme algoritmus, který si dokáže potřebné znalosti v datech najít a naučit sám. [1, 2]

2.1.1 Data a trénovací datová množina

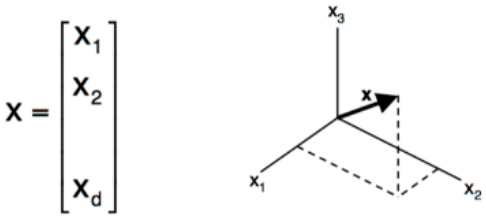
Daty se rozumí pozorování či měření určitých aspektů fyzikální reality. Přidáme-li k těmto datům i význam, stanou se z nich informace. Pro příklad si představme jedno dlouhé pole hodnot $T = [t_1, t_2, ..., t_n]$. Nyní jsou to pro nás jen pouhá data, ale když je poskytnut kontext, např. hodnota $t_i, i = 1, ..., n$ znamená konkrétní naměřenou teplotu v i-tý den z n dní, kdy měření teplot probíhalo, už jsou to pro nás informace. Dokážeme-li navíc tyto informace aplikovat na řešení problému, rozhodování či tvorbu jiných informací, můžeme to potom označit za znalost. Strojové učení se tedy snaží v datech hledat užitečné informace a vytvářet znalosti. V tomto kontextu lze chápat naučený model strojového učení jako procedurální znalost extrahovanou z dat.

Data jsou tudíž naprosto zásadní komoditou pro algoritmy strojového učení. Kolekci dat potřebných pro naučení systému se říká *Trénovací datová množina*.

Chceme-li vyřešit problém pomocí metod strojového učení, musíme mít k dispozici data, která jsou vůči tomuto problému relevantní. Budeme-li například učit systém predikovat počasí, nevybereme si jako trénovací datovou množinu záznamy z měření srážky dvou černých děr, jelikož tato data nám moc relevantních informací o vývoji počasí neposkytnou. Místo toho si jako trénovací množinu vybereme měření z předchozích dní obsahující maximální a minimální teploty, rychlost větru, objem srážek atd., tedy data, která určitým způsobem korelují s vývojem počasí.

Každý aspekt našeho světa jde popsat pomocí více či méně dat neboli pozorovaní o určitých charakteristických vlastnostech tohoto aspektu. Těmto datům se v umělé inteligenci a strojovém učení říká příznaky.

Všechny příznaky potřebné k popsání aspektu se sdružují do příznakového vektoru. Poté lze vytvořit tzv. příznakový prostor aspektu. Jde o abstraktní n-rozměrný vektorový prostor, který obsahuje všechny možné (i nemožné) instance tohoto aspektu. Každý bod tohoto prostoru je konkrétní instance určená konkrétními příznaky. (viz. obr. 2.2)



Obrázek 2.2: Příznakový vektor a příznakový prostor

Pro příklad uvažujme, že chceme popsat černobílý obraz s rozlišením 300x300. Tento obraz se skládá z diskrétních obrazových částic - pixelů. Abychom ho dokázali plně popsat, budeme potřebovat informaci o hodnotě každého pixelu. Každý pixel tak tvoří jeden příznak obrazu a celý obraz je potom popsán pomocí 90000-rozměrného příznakového vektoru a je jedním jediným bodem v 90000-rozměrném příznakovém prostoru všech černobílých obrazů s rozlišením 300x300.

Lze předpokládat, že velmi podobné obrazy mají velmi podobné příznaky a jako body v příznakovém prostoru jsou tak blízko u sebe. Tohoto principu různým způsobem využívají klasifikační či shlukovací algoritmy strojového učení.

Trénovací datová množina je tvořena daty ve formě příznakových vektorů a v závislosti na druhu učícího algoritmu zde může (ale nemusí) být obsažena i informace o požadovaném výstupu.

2.2 Učení s učitelem

Učení s učitelem (supervised learning) je typ strojového učení, který využívá informace o požadovaných výstupech obsažené v trénovací datové množině. Model je pak aproximací, která minimalizuje rozdíl mezi požadovaným výstupem a provedenou predikcí. [1]

Máme tedy trénovací datovou množinu $D = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$, kde $x_i \in R^d$ je příznakový vektor z d-dimenzionálního příznakového prostoru a $y_i \in V$ je odpovídající výstup z prostoru všech možných výstupů daného problému.

Cílem učení s učitelem je nalezení takové funkce $h : R^d \to V$ z prostoru funkcí H, aby platilo $h(x_i) \approx y_i$ pro $(x_i, y_i) \in D$ neboli aby nejlépe aproximovala výstupy trénovací datové množiny a zároveň byla natolik obecná, aby pro každé nové pozorování x platilo $h(x) = y$, kde $y \in V$. [1,3]

Mezi typické úlohy, které se řeší pomocí učení s učitelem patří klasifikace a regrese.

Úloha klasifikace spočívá v nalezení modelu (funkce h), který bude schopný zařadit objekt reprezentovaný příznakovým vektorem x do třídy z předem známých tříd. Výstupní prostor V je v této úloze diskrétní a konečný.

Úloha regrese spočívá v nalezení modelu, který popisuje závislost mezi vstupními a výstupními hodnotami. Výstupní prostor V je v této úloze spojitý a může být nekonečný.

2.3 Učení bez učitele

Učení bez učitele (unsupervised learning) je typ strojového učení, kde trénovací množina neobsahuje žádnou dodatečnou informaci. Trénovací množina $D = \{x_1, x_2, ..., x_n\}$ tedy obsahuje jen samotné příznakové vektory x_i, ale už žádný požadovaný výstup. Cílem tohoto typu učení je analyzovat vstupní data a porozumět jejich struktuře jen na základě jich samých a vytvořit matematický model, který tato data vysvětluje.

Cílem učení bez učitele je nalezení takové funkce $h : R^d \rightarrow R^d$ z prostoru funkcí H, pro kterou platí, že přibližně rekonstruuje původní hodnotu, tedy $h(x_i) \approx x_i$. Jinými slovy chceme, aby funkce obnovila určitou strukturu vstupu. Například pro algoritmus k-means bude funkce h vypadat následovně:

$$h(x_i) = arg \min_{\mu} ||\mu - x_i||^2$$

Kde $\mu \in \mu_1, ..., \mu_k$ jsou centroidy shluků. Taková funkce nám rozděluje data do shluků tak, že vybere ten shluk, pro který platí, že $h(x_i)$ obnovuje přibližnou hodnotu x_i nejlépe. [3]

Obrázek 2.3: Princip shlukování

Nejtypičtější úlohou, kterou řeší učení bez učitele patří shlukování.

Shlukování je úloha, která se zabývá rozdělením množiny dat do podmnožin nazývaných shluky a to na základě jejich podobnosti (obr. 2.3).

2.4 Ztrátová funkce

Máme-li k dispozici datovou množinu D a určený prostor funkcí H (například prostor všech lineárních funkcí se dvěma parametry), je nutné zvolit si jakousi metriku, která bude informovat o (ne)úspěšnosti predikcí učiněných konkrétními funkcemi. [1]

Jako metriku definujeme tzv. ztrátovou funkci (loss function) $L : H \to R$, jež počítá ztrátu pro danou funkci $h \in H$.

Ztrátová funkce mění problém nalezení nejlepší aproximace na problém matematické optimalizace. Formálně zapsáno:

$$arg\min_{h \in H} L(h) = arg\min_{h \in H} \frac{1}{n} \sum_{i=1}^{n} l(h(x_i), y_i)$$

Kde $l(h(x_i), y_i)$ je ztráta jedné datové dvojice (x_i, y_i) pro hypotézu h. Pro úplnost je nutno dodat, že pro učení bez učitele bude platit $y_i = x_i$. [1, 2, 3]

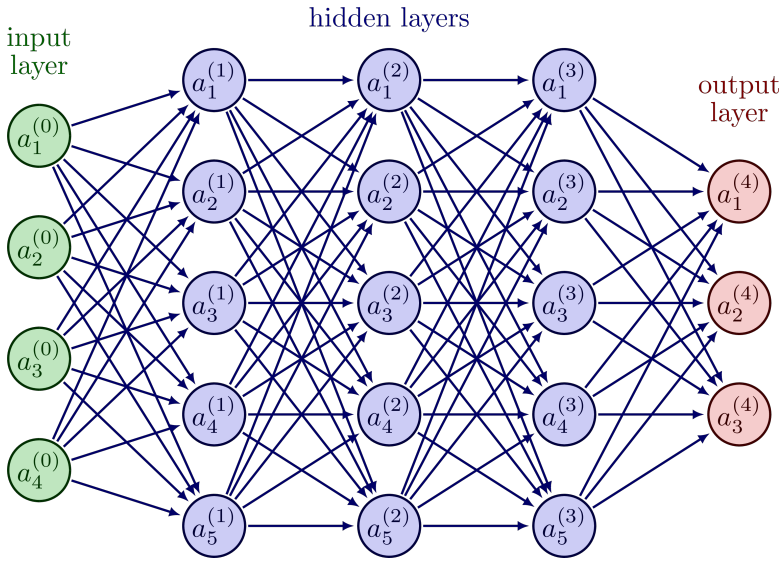
2.5 Neuronové sítě

Neuronová síť patří mezi základní metody strojového učení a je výchozím bodem pro hluboké učení, které se zabývá jejím trénováním a vývojem nových neuronových architektur, jež dokážou řešit i velmi složité nelineární problémy.

Umělou neuronovou síť lze chápat jako matematický model inspirovaný funkcí biologické neuronové sítě v lidském mozku, který pracuje na principu komunikace velkého množství malých výpočetních jednotek (neuronů) paralelně zpracovávajících informace.

Nejedná se o přesný model přímo kopírující biologickou neuronovou síť, ale jen o hrubé přiblížení těch nejzákladnějších principů.
Základním stavebním kamenem neuronové sítě je neuron. Ten je ovšem sám o sobě relativně nepoužitelný pro složitější problémy, avšak v kombinaci se stovkami nebo tisíci jiných neuronů vzájemně propojených synapsemi, vytváří vztahy a výsledky, které často předčí jakékoli jiné metody strojového učení.

Nejobecnější schéma umělé neuronové sítě je znázorněno na obrázku 2.4.



Obrázek 2.4: Základní schéma neuronové sítě

Jednotlivé neurony jsou organizovány do vrstev. Vícevrstvá neuronová síť se pak skládá z několika vrstev propojených synapsemi. První vrstvě se říká vstupní vrstva a slouží pro rozvedení vstupu na všechny neurony následující vrstvy. Počet neuronů ve vstupní vrstvě se tak rovná počtu příznaků ve vstupním příznakovém vektoru.

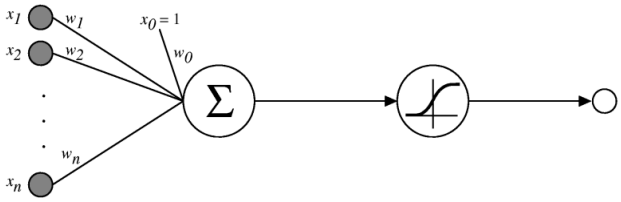
Po vstupní vrstvě obvykle následují skryté vrstvy (případně výstupní, nejedná-li se o hlubokou síť). Tyto vrstvy transformují data a hledají v nich užitečné vztahy a vzorce.

Poslední, výstupní vrstva provádí konečnou transformaci dat do užitečné podoby odpovídající druhu řešené úlohy.

Neuronová síť se učí tak, že své synaptické váhy iterativně upravuje a extrahuje dostatečně obecné vzory z trénovacích dat, které dokáže následně využít pro řešení problému i na datech dosud neviděných. Dalo by se říci, že neuronová síť při procesu učení do své struktury zakóduje znalosti, které potom využívá při řešení daného problému. [1,2]
Přes všechny vzletné názvy vypůjčené z biologie však umělé neuronové sítě nejsou ve své podstatě ničím jiným než jednoduchou, ale mocnou matematikou a pro nejlepší možné pochopení neuronových sítí je nutné se s touto matematikou seznámit.

2.5.1 Matematický model jednotlivého neuronu

Jednotlivý neuron (viz. obr. 2.5) je elementární jednotkou celé neuronové sítě.



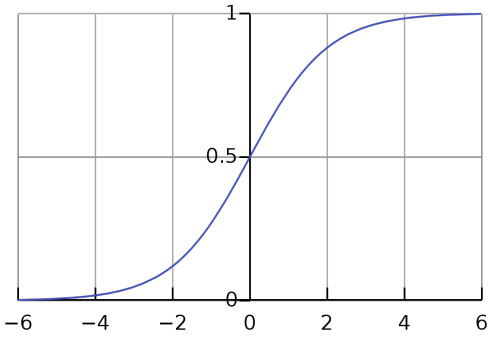
Obrázek 2.5: Formální model neuronu

Matematicky jej lze vyjádřit následovně:

$$z = \sum_{i=1}^{n} w_i x_i + b$$
$$y = f(z)$$

Kde $x = (x_1, x_2, ..., x_n)$ je vstupní příznakový vektor, $w = (w_1, w_2, ..., w_n)$ je vektor vah, b je práh neboli bias neuronu a $f(z)$ je nelineární aktivační funkce.

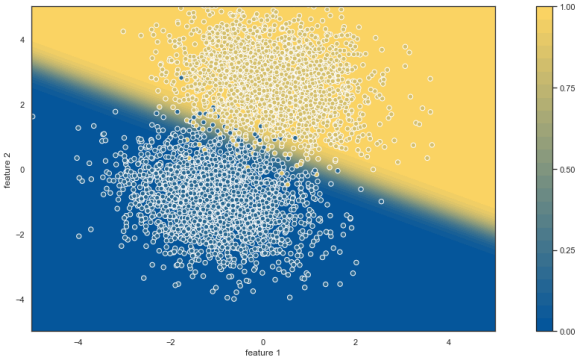
Budeme-li uvažovat aktivační funkci $f(z) = \frac{1}{1+e^{-z}}$, kde $z = w_1x_1 + w_2x_2 + b$, potom je-li $f(z) < 0,5$ =>klasifikujeme vstup do třídy 0, a je-li $f(z) > 0,5$ =>klasifikujeme vstup do třídy 1.



Obrázek 2.6: Graf aktivační funkce sigmoida

Představíme-li si vstupní příznakový prostor R^n a předpokládáme-li, že již máme pevně stanovený vektor vah a bias neuronu, pak zjistíme, že sigmoidní aktivační funkce nám jej lineárně rozdělila na dva poloprostory, jak je vidět na obrázku 2.7. Lze si také povšimnout, že sigmoidní funkce nám

udává i jakousi jistotu klasifikace. Tedy čím blíže jsme rozhodovací hranici, tím nejistější je klasifikace neuronu.



Obrázek 2.7: Lineárně separovaný vstupní příznakový prostor

Jednotlivý neuron tak není nic jiného než lineární funkce, kde váhy a bias neuronu neboli koeficienty lineární funkce udávají sklon a posunutí rozhodovací hranice (obecně n-1 rozměrné nadroviny), která lineárně separuje vstupní příznakový prostor. Jde tedy o jednoduchý lineární klasifikátor.

2.5.2 Přidávání vrstev

Každý neuron v jedné vrstvě generuje lineární rozhodovací hranici a další vrstvy tyto hranice zkombinují do složitějších nelineárních rozhodovacích hranic. Vytváří se tak schopnost neuronové sítě řešit i nelineární problémy.[1]

2.5.3 Matematický model neuronové sítě

Neuronovou síť lze matematicky vyjádřit například takto:

Definice: Nechť $d \in N$ je dimenze vstupní vrstvy, L počet vrstev, $N_0 := d.N_l, l = 1, ..., L$ jsou dimenze skrytých vrstev a výstupní vrstvy, $f : R \to R$ je aktivační funkce. Pro $l = 1, ..., L$ nechť je $z^{(l)}$ afinní transformace

$$z^{(l)} : R^{N_{l-1}} \to R^{N_l}, z^{(l)} = W^{(l)}x + b^{(l)}$$

Kde $W^{(l)} \in R^{N_l x N_{l-1}}$ je matice vah a $b^{(l)} \in R^{N_l}$ je prahový vektor l-té vrstvy. Potom $F : R^d \to R^{N_L}$ takové, že

$$F(x) = f(z^{(L)}f(z^{(L-1)}f(...f(z^{(1)})...)), x \in R^d$$

je hluboká (dopředná) neuronová síť.
V hlubokých neuronových sítích většinou jako základní stavební prvek považujeme celé vrstvy, nikoli jednotlivé neurony. Každou vrstvu lze chápat

jako funkci f, jejíž výstup je vstupem další funkce (vrstvy). Neuronová síť v podstatě není nic jiného než složitá, složená vícerozměrná funkce F.

2.5.4 Geometrická interpretace neuronových sítí

Velmi užitečný pohled na neuronové sítě je nahlížet na její vrstvy jako na tenzorové operace, protože takovýto výraz vrstvy l

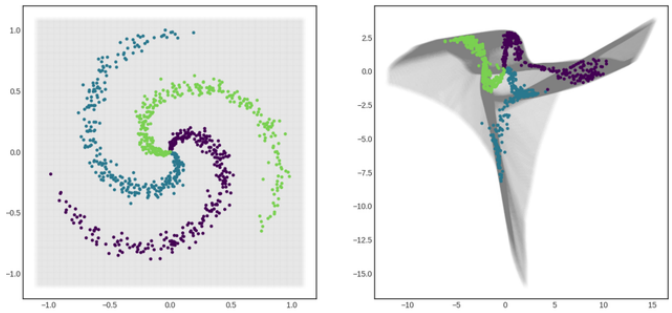
$$z^{(l)} = W^{(l)}x + b^{(l)}$$

$$a^{(l)} = f(z^{(l)})$$

není nic jiného než tenzorová operace složená z elementárních tenzorových operací, kde $z_1^{(l)} = W^{(l)} * x$ je tenzorový součin, $z_2^{(l)} = z_1^{(l)} + b^{(l)}$ je tenzorový součet, $a^{(l)} = f(z_2^{(l)})$ je nelineární zobrazení tenzoru $z_2^{(l)}$ na tenzor $a^{(l)}$.

Protože tyto tenzorové operace lze interpretovat jako geometrické transformace (afinní transformace, translace, rotace, škálování), můžeme celou neuronovou síť interpretovat jako velice složitou nelineární geometrickou transformaci v rozsáhlém mnohorozměrném příznakovém prostoru, realizovanou prostřednictvím posloupnosti mnoha jednoduchých tenzorových operací. [2]

Jak již víme, vstupní data leží v nějakém d-rozměrném příznakovém prostoru. Ovšem tento prostor nemusí být vhodnou reprezentací pro určitý klasifikační problém, protože datové body každé třídy mohou být mezi sebou složitě "zamotány". Hlavním cílem neuronové sítě je potom najít takovou geometrickou transformaci, která by daný příznakový prostor transformovala do podoby, díky níž od sebe všechny datové body nepatřící do jedné třídy oddělila a data se poté dala velice jednoduše klasifikovat, jak je vidět na obrázku 2.8.



Obrázek 2.8: Transformace vstupního prostoru (nalevo) do vhodnější podoby pro klasifikaci (napravo)

2.5.5 Aktivační funkce

Aktivační funkce jsou esenciální složkou neuronových sítí. Přidávají do nich nelinearitu. Kdyby neuronové sítě nelineární aktivační funkce neměli, byly by schopny provádět jen afinní transformace. Jinými slovy by nebyly schopny zachytit nelineární vztahy v řešeném problému a množina všech problémů, které bychom byli schopni neuronovou sítí řešit, by byla příliš omezená a ani přidávání dalších vrstev by tuto množinu nerozšířilo [2]. Aktivační funkce nám tak tuto množinu podstatně rozšiřují.

Mezi nejrozšířenější aktivační funkce patří zejména sigmoida, ReLU, hyperbolický tangens a softmax. Proto si je pojďme lehce představit.

Sigmoida:

$$f(x) = \frac{1}{1 + e^{-x}}$$

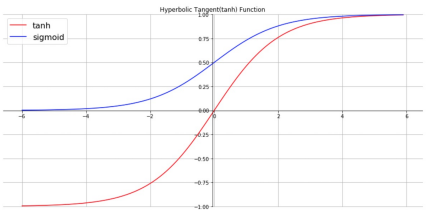
Jde o jednu z nejranějších aktivačních funkcí používaných v neuronových sítí. Tato funkce vezme jakékoli reálné číslo a převede ho na číslo z intervalu $(0, 1)$, jak je vidět z grafu na obrázku 2.6.

Velkou nevýhodou této funkce je, že trpí problémem mizejícího gradientu, což je problém, kdy se při trénování sítě dostává gradient k nulové hodnotě, což velmi stěžuje, případně znemožňuje, její učení. To je také důvod, proč se sigmoida nepoužívá ve skrytých vrstvách, ale používá se až ve výstupní vrstvě sítě, kde její výstup lze, například pro binární klasifikační problém, interpretovat jako pravděpodobnost. [5]

Hyperbolický tangens:

$$f(x) = Tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Tato funkce je velmi podobná sigmoidě. Hlavní rozdíl je v tom, že mapuje vstup na interval $(-1, 1)$, viz. obrázek 2.9. Tato funkce se často používá ve výstupní vrstvě sítě pro regresní problémy. [5]

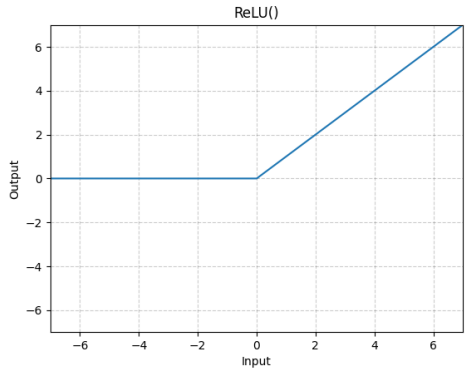


Obrázek 2.9: Graf funkce Tanh (červená) a jeho srovnání se Sigmoidou (modrá)

ReLU:

$$f(x) = max(0, x)$$

Toto je velmi důležitá aktivační funkce, neboť je poměrně odolná proti problému mizejícího gradientu. Používá se ve skrytých vrstvách neuronové sítě. Z obrázku 3.0 nebo z předpisu funkce je vidět, že nuluje všechny záporné vstupy a pro kladné vstupy se chová lineárně.



Obrázek 2.10: Graf funkce ReLU

Další výhodou ReLU je její výpočetní jednoduchost, což ve velmi hlubokých neuronových sítích snižuje výpočetní složitost učení. Dále ReLU produkuje řídké aktivace v síti, což znovu snižuje výpočetní zátěž sítě a zároveň pomáhá proti přeučení.

Nevýhodou ReLU je, že může trpět problémem tzv. dying ReLU neboli umírající ReLU. To nastane, když se některé neurony stávají trvale neaktivními během trénování, protože jejich aktivace se stávají nulovými a nemohou se zotavit. K tomu může dojít, když je vstup do neuronu s ReLU aktivační funkcí negativní a gradient ztrátové funkce vzhledem ke vstupu je také negativní. To způsobuje aktualizaci vah takovým způsobem, že se následný vstup bude trvale držet na negativní úrovni.

Pro zmírnění tohoto problému bylo navrženo několik variant ReLU, například PReLU:

$$f(x) = max(x, ax)$$

Tato varianta zavádí malý, kladný sklon pro negativní vstupy, čímž brání umírající Relu. [2,5]

Softmax

$$f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{k} e^{x_j}}$$

Poslední zde diskutovanou aktivační funkcí je softmax. Tato aktivační funkce, podobně jako sigmoida, stlačí ve výstupní vrstvě vstupní hodnoty na interval $(0, 1)$. Na rozdíl od sigmoidy se však používá pro klasifikační problémy s klasifikací do více tříd.

Softmax bere k-rozměrný vektor jako vstup a vrací k-rozměrný vektor pravděpodobností, přičemž platí, že každá složka tohoto výstupního vektoru představuje pravděpodobnost, že vstup patří do dané třídy. Také platí, že součet všech pravděpodobností ve výstupním vektoru je roven 1.

2.5.6 Ztrátová funkce neuronové sítě a gradientní sestup

Jak již bylo uvedeno v sekci 2.4, pro učení neboli pro hledání funkce, která nejlépe aproximuje řešený problém, potřebujeme metriku k určování úspěšnosti predikcí.

Konkrétní podoba ztrátové funkce záleží na charakteru řešeného problému. Pro regresní problémy se často jako ztrátová funkce používá střední kvadratická odchylka neboli MSE (mean squared error) definována následovně:

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Kde y je požadovaná hodnota a $\hat{y}$ je výstup sítě.

Další ztrátovou funkcí, kterou definujeme, je křížová entropie používaná pro klasifikační problémy.

$$L(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{j=1}^{m} y_{ij} \log(\hat{y}_{ij})$$

Kde n je počet trénovacích příkladů, m je počet tříd a y_{ij} je binární hodnota (0 nebo 1), která indikuje, zda daný příklad patří do třídy j, a $\hat{y_{ij}}$ je predikovaná pravděpodobnost, že daný příklad patří do třídy j.

V případě binární klasifikace lze předešlý výraz zjednodušit, neboť máme jen dvě třídy a platí, že $y \in \{y, 1-y\}$ a $\hat{y} \in \{\hat{y}, 1-\hat{y}\}$. Poté vypadá ztrátová funkce následovně:

$$L(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)]$$

To jsou 3 nejpoužívanější ztrátové funkce v neuronových sítích. [5] Ztrátových funkcí je samozřejmě mnohem více, nicméně cílem této práce není všechny popsat. Proto budeme-li dále v textu potřebovat použít jinou ztrátovou funkci, definujeme ji až tam.

Když už máme ztrátovou funkci, potřebujeme nějaký mechanismus, jak její výstup zpracovat. Cílem je, aby byl výstup ztrátové funkce minimální neboli, aby síť měla nejmenší ztrátu. Proces minimalizace ztrátové funkce a aktualizace parametrů sítě podle této ztráty se nazývá učení neuronové sítě.

Učení se realizuje pomocí algoritmu gradientního sestupu. Jde o iterativní algoritmus matematické optimalizace založený na výpočtu gradientu a aktualizaci parametrů na základě gradientu.

Gradient vícerozměrné funkce $f(x_1, x_2, ..., x_n)$ je diferenciální operátor, který lze chápat jako zobecnění derivace do více rozměrů. Výsledkem gradientu je vektorové pole, které vyjadřuje směr největšího růstu v každém bodě skalárního pole.

Jedná se o vektor parciálních derivací

$$\nabla f(x_1, x_2, ..., x_n) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(x_1, x_2, ..., x_n) \\ \frac{\partial f}{\partial x_2}(x_1, x_2, ..., x_n) \\ ... \\ \frac{\partial f}{\partial x_n}(x_1, x_2, ..., x_n) \end{bmatrix}$$

Záporný gradient poté vyjadřuje směr největšího spádu, čehož algoritmus gradientního sestupu využívá.

Gradientní sestup se dá vyjádřit následujícím výrazem:

$$x_i := x_i - \alpha \frac{\partial f}{\partial x_i}$$

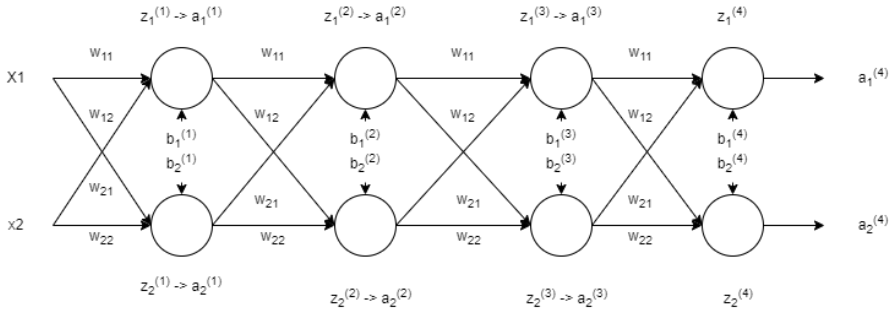
Kde α představuje délku kroku ve směru záporného gradientu. V terminologii strojového učení se kroku říká *stupeň učení* (learning rate).[2]

Dlouhou dobu byl velký problém, jak vypočítat gradient ztrátové funkce neuronové sítě neboli jak rozdistribuovat ztrátu neuronové sítě od výstupní vrstvy až ke vstupní vrstvě. Velký průlom přišel, když byl objeven algoritmus zpětného šíření chyby (backpropagation algorithm). Tento algoritmus podstatě přispěl k rozvoji neuronových sítí a hlubokého učení [6].

2.5.7 Algoritmus zpětného šíření chyby

Algoritmus zpětného šíření chyby je základní algoritmus pro efektivní učení neuronové sítě. Počítá gradient ztrátové funkce podle všech jejích parametrů.

Pro odvození obecných vztahů v algoritmu využijeme síť na obrázku 2.11.



Obrázek 2.11: Referenční neuronová síť

Zaměříme se na ztrátu sítě pro jeden konkrétní vstup x. Ztrátová funkce bude pro jednoduchost střední kvadratická odchylka:

$$L_x = \sum_{i=1}^{2} (\hat{a}_i - a_i^{(4)})^2$$

Nejdříve si odvodíme vzorec pro počítání parciálních derivací pro neurony ve výstupní vrstvě.

Rovnice aktivace prvního výstupního neuronu $a_1^{(4)}$ vypadají takto:

$$z_1^{(4)} = w_{1i} a_i^{(3)} + b_1^{(4)}, \quad pro \quad i = 1, 2$$

$$a_1^{(4)} = \sigma(z_1^{(4)})$$

Můžeme si povšimnout, že aktivace neuronu se skládá z několika funkcí. Chceme-li potom vypočítat parciální derivaci ztráty L_x podle váhy $w_{11}^{(4)}$, použijeme řetězové pravidlo pro počítání složených funkcí a získáme následující výraz:

$$\frac{\partial L_x}{\partial w_{11}^{(4)}} = \frac{\partial L_x}{\partial a_1^{(4)}} \frac{\partial a_1^{(4)}}{\partial z_1^{(4)}} \frac{\partial z_1^{(4)}}{\partial w_{11}^{(4)}} = 2(\hat{a_1} - a_1^{(4)}) \sigma'(z_1^{(4)}) a_1^{(3)}$$

Pro výpočet druhého parametru $w_{12}^{(4)}$ aktivace $a_1^{(4)}$ je výraz téměř totožný:

$$\frac{\partial L_x}{\partial w_{12}^{(4)}} = \frac{\partial L_x}{\partial a_1^{(4)}} \frac{\partial a_1^{(4)}}{\partial z_1^{(4)}} \frac{\partial z_1^{(4)}}{\partial w_{12}^{(4)}} = 2(\hat{a_1} - a_1^{(4)}) \sigma'(z_1^{(4)}) a_2^{(3)}$$

Vidíme, že se výraz liší jen v aktivaci předchozí vrstvy, proto pro další výpočty můžeme provést zjednodušení:

$$\delta_1^{(4)} = \frac{\partial L_x}{\partial z_1^{(4)}} = 2(\hat{a_1} - a_1^{(4)}) \sigma'(z_1^{(4)})$$

Potom pro výstupní neuron s aktivací $a_1^{(4)}$ bude platit:

$$\frac{\partial L_x}{\partial w_{11}^{(4)}} = \delta_1^{(4)} a_1^{(3)}, \quad \frac{\partial L_x}{\partial w_{12}^{(4)}} = \delta_1^{(4)} a_2^{(3)}, \quad \frac{\partial L_x}{\partial b_1^{(4)}} = \delta_1^{(4)} 1$$

Pro druhý výstupní neuron s aktivací $a_2^{(4)}$ platí stejný princip:

$$\frac{\partial L_x}{\partial w_{21}^{(4)}} = \delta_2^{(4)} a_1^{(3)}, \quad \frac{\partial L_x}{\partial w_{22}^{(4)}} = \delta_2^{(4)} a_2^{(3)}, \quad \frac{\partial L_x}{\partial b_2^{(4)}} = \delta_2^{(4)} 1$$

Nyní se zaměříme na první neuron poslední skryté vrstvy s aktivací $a_1^{(3)}$. Pro výpočet parametru $w_{11}^{(3)}$ budeme pokračovat stejným způsobem, tedy zřetězením derivace zpět do předchozí vrstvy. To si můžeme dovolit, neboť, jak jsme si již říkali, celá neuronová síť je velká složená funkce.

Protože se aktivace $a_1^{(3)}$ podílí na všech aktivacích následující vrstvy, musíme všechny jejich derivace zahrnout ve výpočtu. K tomu nám pomůžou již vypočítané δ z předchozí výstupní vrstvy:

$$\frac{\partial L_x}{\partial w_{11}^{(3)}} = (\sum_{k=1}^{2} \frac{\partial L_x}{\partial a_k^{(4)}} \frac{\partial a_k^{(4)}}{\partial z_k^{(4)}} \frac{\partial z_k^{(4)}}{\partial a_1^{(3)}}) \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial w_{11}^{(3)}} = (\sum_{k=1}^{2} \delta_k^{(4)} w_{k1}^{(4)}) \sigma'(z_1^{(3)}) a_1^{(3)}$$

Pro druhý parametr bude výpočet obdobný:

$$\frac{\partial L_x}{\partial w_{12}^{(3)}} = (\sum_{k=1}^{2} \frac{\partial L_x}{\partial a_k^{(4)}} \frac{\partial a_k^{(4)}}{\partial z_k^{(4)}} \frac{\partial z_k^{(4)}}{\partial a_1^{(3)}}) \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial w_{12}^{(3)}} = (\sum_{k=1}^{2} \delta_k^{(4)} w_{k1}^{(4)}) \sigma'(z_1^{(3)}) a_2^{(3)}$$

Je vidět, že jsou oba výrazy zase velice podobné, proto lze provést znovu zjednodušení:

$$\delta_1^3 = (\sum_{k=1}^{2} \delta_k^{(4)} w_{k1}^{(4)}) \sigma'(z_1^{(3)})$$

Obecný vzorec pro výpočet δ pro všechny neurony skryté vrstvy vypadá následovně:

$$\delta_j^3 = (\sum_{k=1}^{2} \delta_k^{(4)} w_{kj}^{(4)}) \sigma'(z_j^{(3)})$$

Tento vzorec lze nakonec zobecnit pro všechny skryté vrstvy, kde pro δ v předchozích vrstvách se využijí δ neuronů z vrstvy bezprostředně následující:

$$\delta_j^l = (\sum_{k=1}^{2} \delta_k^{(l+1)} w_{kj}^{(l+1)}) \sigma'(z_j^{(l)})$$

Kde $\delta_j^{(l)}$ je delta j-tého neuronu v l-té vrstvě, $a_k^{(l-1)}$ je aktivace k-tého neuronu v $l-1$-té vrstvě, $w_{kj}^{(l)}$ je váha j-tého neuronu v $l-1$-té vrstvě a k-tého neuronu v l-té vrstvě.

Parciální derivace jednotlivých parametrů jsou tedy následující:

$$\frac{\partial L_x}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}, \quad \frac{\partial L_x}{\partial b_j^{(l)}} = \delta_j^{(l)} 1$$

Tímto způsobem dokážeme spočítat gradient ztrátové funkce podle všech parametrů neuronové sítě. [1, 7]

Obecný algoritmus pro učení neuronové sítě tedy vypadá následovně:

1. krok: Inicializujeme všechny váhy w a prahy b neuronové sítě náhodně.

2. krok: Pro každý trénovací příklad z trénovací datové množiny.

2.1. Spočítáme aktivace pro celou síť:

$$Z^{(l)} = W^l a^{(l-1)} + b^{(l)}$$

$$a^{(l)} = \sigma(Z^{(l)})$$

2.2. Vypočítáme δ pro neurony výstupní vrstvy:

$$\delta_j^{(l)} = 2(\hat{a}_j - a_j^{(l)})\sigma'(z_j^{(l)})$$

2.3. Spočítáme δ pro všechny neurony z předchozích vrstev:

$$\delta_j^l = (\sum_{k=1}^{2} \delta_k^{(l+1)} w_{kj}^{(l+1)})\sigma'(z_j^{(l)})$$

2.4. Spočítáme gradient ztrátové funkce pro všechny w a b pro příklad x:

$$\frac{\partial L_x}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)}, \quad \frac{\partial L_x}{\partial b_j^{(l)}} = \delta_j^{(l)} 1$$

3. krok: Zprůměrujeme gradient přes celou trénovací množinu:

$$\frac{\partial L}{\partial w_{jk}^{(l)}} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L_i}{\partial w_{jk}^{(l)}}, \quad \frac{\partial L}{\partial b_j^{(l)}} = \frac{1}{n} \sum_{i=1}^{n} \frac{\partial L_i}{\partial b_j^{(l)}}$$

4. krok: Upravíme parametry s použitím algoritmu gradientního sestupu:

$$w_{jk}^{(l)} := w_{jk}^{(l)} - \alpha \frac{\partial L}{\partial w_{jk}^{(l)}}, \quad b_j^{(l)} := b_j^{(l)} - \alpha \frac{\partial L}{\partial b_j^{(l)}}$$

5. krok: Opakovat kroky 2-4 dokud L>ϵ

2.5.8 Regularizace

Neuronové sítě mají velmi silnou reprezentační sílu, která jim dovoluje se naučit i velmi komplexní problémy.

Tato síla však může způsobovat problémy (zvláště pokud máme malou datovou množinu). Jedná se o problém, kdy se neuronová síť naučí až příliš dobře aproximovat trénovací datovou množinu, což se poté odrazí na její

neschopnosti činit dobré predikce na datech, které ještě neviděla. Tomu se říká přeučení (anglicky overfitting).

Je to zapříčiněno tím, že na malé datové množině má velká neuronová síť kapacitu se naučit i takové transformace, které jsou charakteristické pouze pro konkrétní trénovací množinu, nikoliv pro řešený problém.

Cílem učení neuronové sítě je vytvořit model, který se dokáže naučit dostatečně obecné transformace, aby dokázal co nejlépe generalizovat celý řešený problém.

Naštěstí existuje tzv. regularizace, to je technika používaná k omezení přeučení sítě. Mezi hlavní regularizační metody neuronových sítí patří L1/L2 regularizace či dropout regularizace.

L1/L2 regularizace přidává ke ztrátové funkci penalizaci (regularizační člen) za velké hodnoty vah. L1 regularizace trestá velké hodnoty vah absolutní hodnotou a L2 regularizace trestá velké hodnoty vah čtvercem. To omezuje velké výkyvy vah a zabraňuje modelu příliš se přizpůsobit trénovacím datům, protože se zvyšuje celková hodnota ztráty, na kterou poté reaguje algoritmus učení.

Vzorečky L1 a L2 regularizaci:

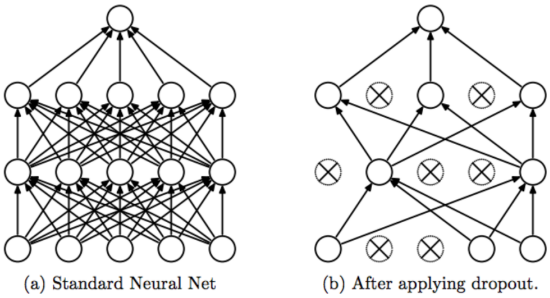
$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{N} L(y_i, \hat{y}_i) + \frac{\lambda}{2} \sum_{j=1}^{p} |w_j|$$

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{N} L(y_i, \hat{y}_i) + \frac{\lambda}{2} \sum_{j=1}^{p} w_j^2$$

Kde n je počet trénovacích příkladů, p je počet vah v síti a λ je parametr udávající sílu regularizačního členu.

Další široce využívanou regularizační technikou je dropout (výpadek). Hlavní myšlenkou dropoutu je náhodné deaktivování neuronů během trénování (viz. obrázek 2.12), tím se síť učí nezávislosti mezi neurony v dané vrstvě a nutí ji se spoléhat na informace z jiných částí sítě. To má za následek, že síť se stává robustnější vůči šumu v datech a zmenšuje se pravděpodobnost, že se naučí nezajímavé charakteristiky v datech, které by mohly vést k přeučení.

Výsledkem použití dropoutu je, že síť se učí vytvářet redundance v reprezentaci dat, což vede k robustnějším a obecnějším modelům.



Obrázek 2.12: Dropout regularizace

Nyní bychom již měli mít vybudovanou solidní představu o strojovém učení a zejména o jeho podoblasti, hlubokém učení, které se zabývá studiem hlubokých neuronových sítí. [2]

V další kapitole se podíváme na různé důležité třídy a architektury neuronových sítí, vysvětlíme si, jak vypadají, jak fungují a jak bychom je mohli využít v medicínských aplikacích.

3 Třídy a architektury neuronových sítí a jejich možná medicínská využití

Až doposud jsme se na hluboké neuronové sítě dívali v jejich, dalo by se říci, nejzákladnějším tvaru. Tedy jako na neurony organizované do plně propojených vrstev bez cyklů. Jedná se o třídu neuronové sítě nazývanou *vícevrstvý perceptron*. Existují však i třídy neuronových sítí, které nejsou plně propojené nebo obsahují cykly. Ukázalo se, že různé třídy jsou velice vhodné pro práci s různými typy dat. Některé jsou vhodnější pro práci s obrazovými daty, jiné zase se sekvenčními či grafovými daty.

V této kapitole si přiblížíme několik důležitých tříd a architektur neuronových sítí a ukážeme si jejich aplikaci pro zpracování a analýzu různých typů medicínských dat.

Než však pokročíme k prvnímu typu neuronové sítě, je záhodno upřesnit a doplnit zde používanou terminologii. Tato terminologie slouží pouze pro tento text, neboť v praxi není jednotná a různé literatury používají termíny jako třída a architektura odlišně.

Třídou neuronové sítě budeme rozumět obecný popis toho, jak jsou neuronové sítě propojeny, jaké používají typy vrstev nebo zda se jedná o dopředné sítě, či sítě obsahující cykly.

Architekturou budeme rozumět její konkrétní popis. Tedy konkrétní počet vrstev s konkrétním počtem neuronů v každé vrstvě, určenými aktivačními a ztrátovými funkcemi.

Model je již naučená neuronová síť s konkrétními hodnotami parametrů.

3.1 Konvoluční neuronová síť

Konvoluční neuronové sítě, zkráceně CNN, jsou velkou třídou neuronových sítí. Skládají se ze série vrstev, které se od klasického vícevrstvého perceptronu liší tím, že provádějí matematickou operaci zvanou konvoluce. Takovým vrstvám se poté říká konvoluční vrstvy.

První konvoluční neuronová síť, která se dala trénovat pomocí gradientního sestupu a algoritmu zpětného šíření chyby, byla vyvinuta v roce 1989 francouzským počítačovým vědcem *Yann André LeCunem* přímo pro účely

zpracování obrazu. Jejich vývoj byl inspirován sérií experimentů vizuálního kortexu koček v roce 1962 až 1965 prováděných neurofyziology *Torstenem Wieselem* a *Davicem H. Hubelem*, které prokázaly existenci specializovaných buněk selektivně reagujících na různé vizuální prvky, jako jsou hrany nebo linie. Zjistili, že tyto buňky jsou organizovány hierarchicky, s jednoduchými buňkami na nižších úrovních reagujícími na základní prvky a složitými buňkami na vyšších úrovních reagujícími na složitější vzory.

Objev těchto specializovaných buněk a jejich hierarchické organizace poskytl inspiraci pro vývoj konvolučních neuronových sítí, které jsou založeny na myšlence použití konvolučních operací (filtrů) k extrakci stále složitějších vzorů ze vstupních dat. Filtry jsou navrženy tak, aby napodobovaly reakční vlastnosti specializovaných buněk v vizuálním kortexu a hierarchická organizace filtrů umožňuje reprezentaci stále abstraktnějších a složitějších vzorů.

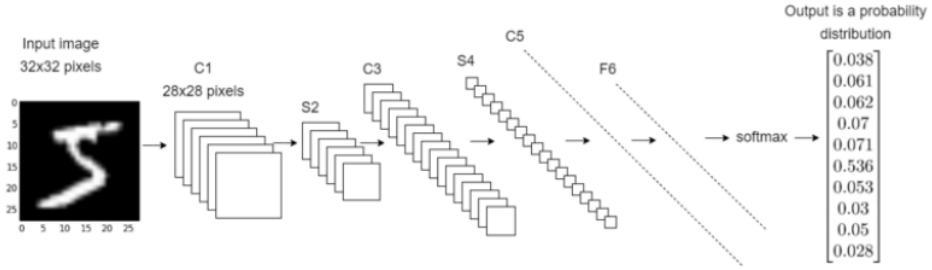
V roce 2012 vynesla konvoluční neuronové sítě do povědomí širší veřejnosti tzv. "ImageNet Challenge", soutěž v klasifikaci obrazů s více než milionem trénovacích obrázků a tisícem kategoriích. Vítězný model *AlexNet*, vyvinutý týmem vedeným vědcem Andrewem Ngem, dosáhl výrazně lepších výsledků než ostatní modely, což potvrdilo potenciál konvolučních neuronových sítí v oblasti zpracování obrazu.[1]

Konvoluční neuronové sítě jsou zodpovědné za ohromný pokrok v oblasti zpracování digitálního obrazu a počítačového vidění. Jsou schopné dosáhnout vynikajících výsledků v rozpoznávání, klasifikaci a segmentaci obrazu. Aplikují se ale také v jiných oblastech, jako např. zpracování přirozeného jazyka. V těchto aplikacích se CNN používají ke zpracování sekvencí slov nebo vět a jsou schopné rozpoznat jazykové vzorce a sémantické vztahy v textu.

Mají široké využití v průmyslu, zdravotnictví i v akademickém světě.

3.1.1 Obecná struktura CNN

Konvoluční neuronové sítě jsou dopředné neuronové sítě sestavené z různých druhů vrstev, z nichž nejčastější jsou konvoluční a poolingové (sdružovací) vrstvy. Tyto vrstvy postupně zpracovávají vstupní data pomocí operace konvoluce, následného sdružování informací a aplikace nelineární funkce. Výstupem konvolučních vrstev jsou mapy příznaků, které zobrazují přítomnost různých vizuálních prvků, jako jsou hrany, textury a tvary. Poslední vrstvy konvoluční neuronové sítě bývají plně propojené a slouží ke klasifikaci nebo regresi na základě příznaků získaných v předchozích vrstvách (viz. obr. 3.1). [8]



Obrázek 3.1: Struktura CNN

3.1.2 Konvoluce

Jak již bylo uvedeno, hlavním prvkem CNN jsou konvoluční vrstvy provádějící konvoluci.

Konvoluce je matematická operace široce využívaná v oblastech jako například fyzika, statistika, zpracování signálů, strojové učení či zpracování obrazu. Konvoluce ze dvou funkcí $f(x)$ a $h(x)$ vytvoří funkci $g(x) = (f * h)(x)$ definovanou vztahem:

$$g(x) = (f * h)(x) = \int_{-\infty}^{+\infty} f(a)h(x-a)da = \int_{-\infty}^{+\infty} f(x-a)h(a)da$$

Význam konvoluce lze vnímat jako vážený průměr funkce f s váhovou funkcí h pro každou hodnotu funkce f. Váhová funkce h se často nazývá jádro konvoluce, konvoluční maska nebo filtr.

Pro diskrétní konvoluci, která nás v kontextu konvolučních neuronových sítí bude primárně zajímat, nechť $f(m)$ a $h(m)$ jsou diskrétní funkce, potom platí následující výraz:

$$g(m) = (f * h)(m) = \sum_{r=-\infty}^{+\infty} f(r)h(m-r)$$

Pro dvourozměrný případ platí:

$$g(m,n) = (f * h)(m,n) = \sum_{r=-\infty}^{+\infty} \sum_{s=-\infty}^{+\infty} f(r,s)h(m-r,n-s)$$

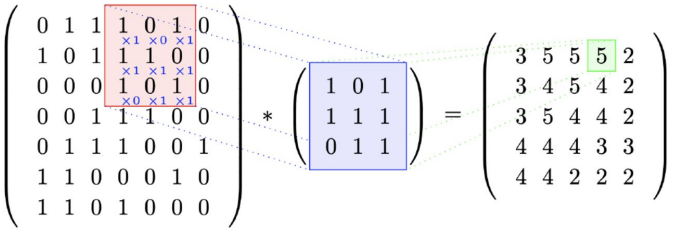
Funkce $f(m,n)$ a $h(m,n)$ můžeme chápat jako matice, poté $f(m,n)$ bude představovat obrazovou matici pixelů a $h(m,n)$ bude matice vah neboli konvoluční filtr. [9]

3.1.3 Filtry ve zpracování obrazu

Konvoluce se prakticky uplatňovala (a stále uplatňuje) ve zpracování obrazu ještě před nástupem konvolučních neuronových sítí.

Konvoluci lze s použitím vhodných konvolučních filtrů použít například pro redukci šumu v obraze, zvýraznění hran, rozostření, detekci textur, segmentaci a mnoho dalších úkolů.

Prakticky si lze konvoluci představit tak, že se matice filtru posouvá po obraze a pro každý pixel se vypočítá vážený průměr hodnot v jeho okolí, jak znázorňuje obrázek 3.2.



Obrázek 3.2: Praktický výpočet obrazové konvoluce

Existuje mnoho typů konvolučních filtrů, jako jsou například Gaussův filtr pro rozostření nebo Laplaceův filtr pro detekci detailů. Každý z těchto filtrů má své vlastní specifické využití a efekt na obraz.

Pro ukázku vezměme Sobelův filtr, který je často používán pro detekci hran v medicínských obrazech. Pomocí tohoto filtru lze snadno nalézt okraje orgánů nebo lézí v MRI nebo CT snímcích.

Sobelův filtr lze aplikovat v horizontálním a vertikálním směru.

Pro horizontální detekci hran se používá následující matice:

$$S_x = \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

Pro vertikální detekci hran se používá matice S_y, která se získá transpozicí matice S_x:

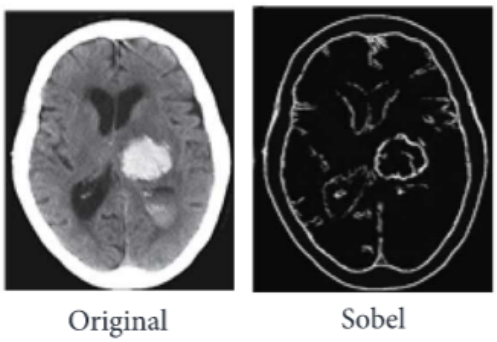
$$S_y = S_x^T = \begin{pmatrix} -1 & 0 & -1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}$$

Konvoluce obrazu s maticí S_x detekuje horizontální hrany a s maticí S_y vertikální hrany. Výsledný obraz pak může být získán jako odmocnina součtu druhých mocnin horizontálního a vertikálního gradientu obrazu:

$$g(x, y) = \sqrt{g_x(x, y)^2 + g_y(x, y)^2}$$

Kde $g_x(x, y)$ a $g_y(x, y)$ jsou odpovídající výsledky konvoluce s maticemi S_x a S_y. [10]

Na obrázku 3.3. můžeme vidět aplikaci Sobelova filtru na snímek mozku s viditelným tumorem.



Obrázek 3.3: Využití Sobelova filtru pro detekci hran v CT snímku mozku

3.1.4 Konvoluční vrstvy

Nyní již máme vybudovanou základní intuici konvoluce a efektu konvoluce obrazu a vhodného filtru, proto se pojďme podívat na konvoluční vrstvy.

Konvoluční vrstvu si můžeme představit jako mnoho různých filtrů, které klouzají po obraze a hledají určité vzory. Tam, kde se část obrazu shoduje se vzorem filtru, vrátí konvoluce velkou kladnou hodnotu, a pokud nedojde k žádné shodě, vrátí nulu nebo menší hodnotu.

Konvoluční vrstvy pracují obecně nikoli s maticemi, ale s tenzory. Tenzor si můžeme zjednodušeně představit jako zobecnění matice nebo jako vícerozměrné číselné pole.

Obvyklým vstupem konvoluční vrstvy je tedy trojrozměrný tenzor, kde první dva rozměry odpovídají výšce a šířce obrazu a třetí rozměr představuje počet kanálů (tedy 1 pokud se jedná o šedotónový obraz a 3 pokud se jedná o barevný RGB).

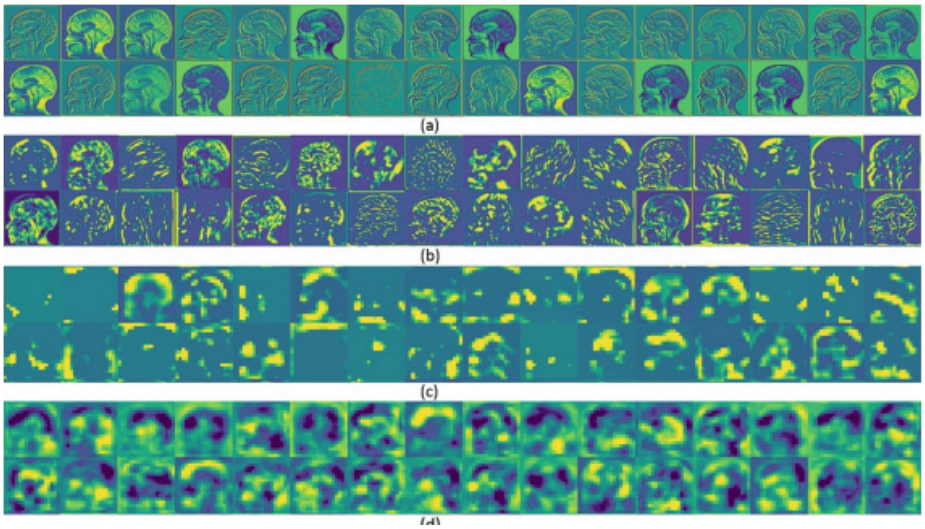
Výstupem konvoluční vrstvy je nová reprezentace vstupních dat, nový tenzor, který zachycuje detekované příznaky v různých oblastech vstupu. Tento tenzor se nazývá mapa příznaků a má stejnou hloubku jako počet filtrů v konvoluční vrstvě (hloubka již tedy nesouvisí s počtem barevných kanálů). Vstup do další vrstvy je tvořen touto mapou příznaků.

Učení v konvolučních vrstvách znamená naučení se takových vhodných filtrů (neboť filtr není nic jiného, než matice vah představující synaptické

spoje mezi neurony), které ze vstupního obrazu dokážou extrahovat užitečné lokální příznaky, které se využijí pro klasifikaci, detekci, segmentaci v posledních plně propojených vrstvách.

Obecně platí, že první konvoluční vrstva se naučí jednoduché lokální příznaky, jako třeba hrany. Druhá vrstva se bude z výstupu první vrstvy učit složitější příznaky. Čím je konvoluční vrstva hlouběji, tím abstraktnější a komplexnější příznaky se naučí. [2]

Na obrázku 3.4 můžeme vidět vizualizaci výstupů konvolučních vrstev MRI snímku. Čím níže se dostáváme, tím jsou mapy příznaků komplexnější a pro rozhodování neuronové sítě hodnotnější.



Obrázek 3.4: Vizualizace map příznaků jednotlivých konvolučních vrstev

Po každé konvoluční vrstvě následuje aplikace nelineární aktivační funkce, která přidává do sítě nelinearitu, aby se síť mohla učit komplexnější příznaky. Pokud by aktivační funkce nebyla přítomna, všechny vrstvy neuronové sítě by nedělali nic jiného, nežli lineární transformace, neboť konvoluce není nic jiného, než lineární operace [8].

3.1.5 Pooling vrstvy

Pooling vrstvy jsou jedním ze stavebních bloků konvolučních neuronových sítí. Zatímco konvoluční vrstvy extrahují příznaky pomocí filtrů, pooling vrstvy agregují vlastnosti získané konvoluční vrstvou. Jejich účelem je postupné zmenšování prostorového rozměru reprezentace s cílem minimalizovat počet parametrů a výpočtů v síti. Tímto způsobem umožňují efektivní zpracování větších datových sad a zrychlení učení v neuronové síti.

Další podstatnou výhodnou vlastností poolingových vrstev je zavedení lokální invariance vůči translaci. Mapa příznaků vytvořená filtry konvolučních vrstev je závislá na umístění. Pokud by se například objekt na obrázku trochu posunul, nemusel by být rozpoznatelný konvoluční vrstvou. To znamená, že mapa příznaků zaznamenává přesné polohy prvků ve vstupu. Poolingové vrstvy poskytují invarianci, díky které je CNN invariantní vůči posunům, tj. i když je vstup CNN posunut, CNN bude stále schopna rozpoznat vlastnosti ve vstupu.

Nejpoužívanější přístupy pro realizaci poolingových vrstev jsou max-pooling (sdružování dle maxima) a average-pooling (sdružování dle průměru).

Max-pooling probíhá tak, že se pomocí filtru s rozměrem (většinou) 2x2 a krokem 2 posouváme po mapě příznaků a z oblasti překrytou filtrem vybereme maximální hodnotu, tedy:

$$P_{max}(\begin{pmatrix} 2 & 3 & 6 & 1 \\ 4 & 10 & 0 & 1 \\ 8 & 0 & 0 & 2 \\ 1 & 2 & 1 & 1 \end{pmatrix}) = \begin{pmatrix} 10 & 6 \\ 8 & 2 \end{pmatrix}$$

Max-pooling zachovává nejvýraznější rysy mapy příznaků a vrácený obrázek je ostřejší než původní.

Average-pooling probíhá velice podobně jako max-pooling s tím rozdílem, že v oblasti překryté filtrem spočítáme průměr hodnot, tedy:

$$P_{avg}(\begin{pmatrix} 2 & 3 & 6 & 1 \\ 4 & 10 & 0 & 1 \\ 8 & 0 & 0 & 2 \\ 1 & 2 & 1 & 1 \end{pmatrix}) = \begin{pmatrix} 4.75 & 2 \\ 2.75 & 1 \end{pmatrix}$$

Average-pooling zachovává průměrné hodnoty prvků mapy příznaků. Vyhladí obraz a zároveň zachová podstatu příznaků v obraze. [2,11]

3.1.6 Efektivita konvolučních neuronových sítí

V klasické plně propojené neuronové síti každý neuron v první skryté vrstvě přijímá všechny hodnoty neuronů ze vstupní vrstvy, která jen kopíruje a posílá vstupní vektor do sítě. Tento přístup ale způsobuje prudký nárůst parametrů zejména pokud je vstup sítě vysokodimenzionální. Například pokud má vstupní vektor rozměr 90 000, což by odpovídalo obrázku o rozměrech

300×300 pixelů, měl by každý neuron první skryté vrstvy 90 000 parametrů. Pokud by tato vrstva obsahovala byť jen 256 neuronů, celkový počet parametrů této vrstvy by přesáhl 23 miliónů.

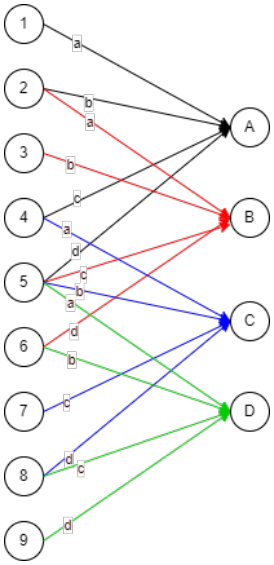
Konvoluční neuronové sítě naproti tomu zavádějí princip, kdy je každý neuron napojen jen na malou část neuronů vstupní vrstvy (např. výsek obrázku odpovídající konvolučnímu filtru). Tento jev označujeme pojmem lokální konektivita. Lokální konektivita se v konvolučních neuronových sítích využívá nejen ve vrstvě napojené na vstupní vrstvu, ale také ve skrytých vrstvách a je propagována do celé sítě.

Dále konvoluční neuronová síť využívá principu tzv. sdílení parametrů, což znamená, že neurony v jedné vrstvě sdílejí stejné hodnoty parametrů. Toto přináší výhodu v paměťové náročnosti, protože počet parametrů, které je třeba uložit se výrazně snižuje. Tohoto sdílení lze dosáhnout právě operací konvoluce.

Pro vizualizaci tohoto principu mějme následující konvoluční operaci mezi dvěma maticemi (mezi obrázkem a filtrem).

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} * \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

Graf neuronové sítě realizující tuto operaci je vidět na obrázku 3.5. Můžeme si ihned povšimnout, že neurony ve skryté (v tomto případě zároveň i výstupní) vrstvě mají každý jen 4 parametry a že neurony sdílejí parametry mezi sebou. Celkový počet parametrů této jednoduché sítě je jen 4.



Obrázek 3.5: Sdílení parametrů

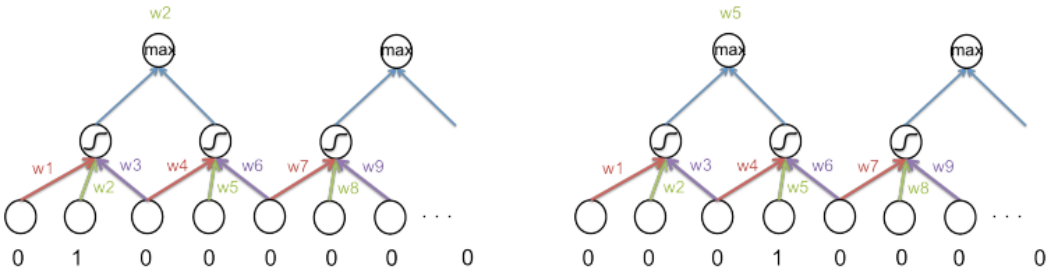
Sdílení parametrů vychází z předpokladu, že každý vzorek vstupu obsahuje nějaké lokální vlastnosti, které se v rámci vzorku opakují. Má tedy smysl detekovat stejnou vlastnost na všech částech vstupu. Vlastnost, kterou detekujeme, je reprezentována právě souborem parametrů neboli filtrem konvoluce, který je sdílený. Vrstva poté obsahuje několik filtrů a na výstupu se pak objeví několik aktivačních map, každá pro jednu vlastnost (jeden filtr).

Sdílení vah je tedy jeden z hlavních důvodů stojících za efektivitou konvolučních neuronových sítí. [11]

Dalším důvodem efektivity CNN je invariance vůči translaci. To je dosaženo (jak již bylo řečeno) poolingovými vrstvami.

Pro velmi jednoduchou ilustraci uveďme následující příklad. Představme si, že následující dva vektory $x_1 = (0, 1, 0, 0, 0, 0, 0, ...)$ a $x_2 = (0, 0, 0, 1, 0, 0, 0, ...)$ představují dva vstupní 1D obrazy, každý s jednou bílou tečkou.

Oba obrazy jsou stejné, až na to, že v x_2 se bílý bod posune o dva pixely doprava ve srovnání s x_1. Na obrázku 3.6 vidíme, že když se tečka posune o 2 pixely doprava, změní se hodnota prvního max-pooling neuronu z w_2 na w_5. Ale protože $w_2 = w_5$, hodnota neuronu se nemění. [11]



Obrázek 3.6: Max-pooling invariance

3.1.7 Využití CNN v medicínských aplikacích

Díky své schopnosti efektivně zpracovávat obrazová data jsou konvoluční neuronové sítě hojně využívány v různých oblastech, včetně automobilového průmyslu, bezpečnosti, astronomie nebo nás primárně zajímající medicíny.

Právě v medicíně tyto sítě umožňují rychlejší a přesnější diagnózu různých patologií a poskytují lékařům cenné informace pro plánování léčby.

Přirozeným využitím konvolučních neuronových sítí v medicíně je zpracování a analýza snímků, které jsou výstupem medicínských zobrazovacích metod.

Medicínské zobrazování zahrnuje různé techniky a postupy, které umožňují získat obrazy vnitřní anatomie člověka a umožňují lékařům detailně analyzovat různé aspekty těla bez nutnosti provádět invazivní zákroky. Dnes existuje mnoho strategií, jak tyto snímky pořizovat, například ultrazvuk, rentgen, počítačová tomografie, magnetická rezonance (funkční i strukturální), pozitronová emisní tomografie, endoskopie, termografie, externí zobrazování (např. melanomů) či mikroskopie.

Interpretace výsledků medicínského zobrazování a jejich analýza není snadný úkol. Každá technika se řídí fyzikálním principem a vyžaduje odborného profesionála. Nicméně v úkolech jako je klasifikace, detekce, segmentace, predikce a dalších úkolech se ukazuje velký přínos a potenciál konvolučních neuronových sítí, které v těchto úkolech mnohdy dokáží předčit tradiční metody. [12]

Konvoluční neuronové sítě pomáhají například s detekcí nádorů, lézí po mrtvici nebo v diagnostice onemocnění, jako je Alzheimerova choroba a Parkinsonova choroba, kde se mohou naučit detekovat abnormality v mozku specifické pro tyto choroby.

Konvoluční neuronové sítě se využívají také ve zpracování sekvenčních dat a jsou využívány v oblastech analýzy lidského genomu a vývoji nových léků.

Například neuronová architektura DeepCCI (CCI=Chemical-chemical interaction), která byla navrhnutá pro predikci pravděpodobnosti interakce dvou chemikálií využívá 1D konvoluční vrstvy pro extrakci různých funkčních skupin a jejich prostorových uspořádání v molekulách pro identifikaci chemické vazby mezi molekulami, kde vstupní chemikálie jsou reprezentovány SMILES řetězci. [13]

Celkově lze říci, že konvoluční neuronové sítě jsou jedna z nejvyužívanějších a nejúspěšnějších neuronových sítí v medicíně a do budoucna mají stále velký potenciál zefektivnit zdravotnictví, zejména oblast lékařské diagnostiky a medicínského výzkumu.

3.1.8 Příklad 1: klasifikace rentgenových snímků pomocí CNN

Jako první praktický příklad si ukážeme binární klasifikaci rentgenových snímků do dvou tříd pomocí jednoduché CNN. Pro tento a všechny další příklady bude využíván framework Keras s TensorFlow backendem a bude během nich doplňována terminologie a vysvětlovány potřebné koncepty.

Keras je framework pro hluboké učení v pythonu, který byl navržen výzkumníky pro rychlé experimentování s hlubokým učením. Keras má jed-

noduché a uživatelsky přívětivé API, umožnuje běh stejného kódu, jak na CPU, tak na GPU, má vestavěnou podporu pro počítačové vidění a zpracování sekvenčních dat a lze v něm naimplementovat jakoukoliv neuronovou architekturu. Keras poskytuje uživatelům vyšší úroveň abstrakce, což umožňuje snadné použití a rychlé prototypování, ale může vést k menší kontrole nad výkonem.

Značnou Nevýhodou tohoto frameworku je omezená flexibilita, přestože v jistých ohledech se jedná o velmi flexibilní framework, má některá omezení v porovnání s jinými knihovnami pro hluboké učení. Například není tak jednoduché vytvářet vlastní vrstvy, což může být omezující pro některé uživatele. Pro naše účely je nicméně naprosto dostačující.

Cílem tohoto příkladu je demonstrovat využití konvolučních neuronových sítí v lékařské diagnostice. Konkrétně se nyní budeme zabývat naučením jednoduchého CNN klasifikátoru, který dokáže klasifikovat rentgenové snímky hrudníku pacienta do dvou tříd (zda-li pacient má, nebo nemá zápal plic), jedná se tedy o úlohu binární klasifikace.

Naše datová množina je organizován do tří adresářů (train, test, val). Každý adresář obsahuje dva podadresáře pro dvě třídy snímků (PNEUMONIA/NORMAL). K dispozici je celkem 5 863 rentgenových snímků ve formátu JPEG.

Rentgenové snímky hrudníku jsou od dětských pacientů ve věku od jednoho do pěti let pořízené z Guangzhou Women and Children's Medical Center. Veškeré snímky byly pořízeny jako součást rutinní klinické péče o pacienty.

Datová množina je obsažena v příloze bakalářské práce v adresáři chest_xray a zároveň je volně dostupná na webové stránce Kaggle:
`https://www.kaggle.com/datasets/paultimothymooney/chest-xray-pneumonia`

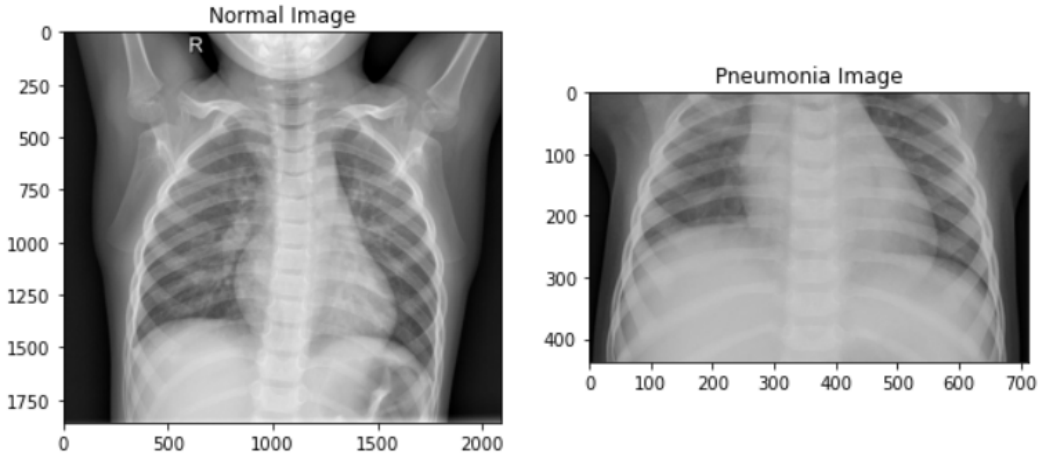
Než začneme, nejdříve prozkoumejme rozložení snímků v adresářích. Obecným postupem přípravy dat je rozdělení datové množiny na 3 podmnožiny - trénovací, validační a testovací.

Trénovací množina (70-80 % dat) je určená k trénování neuronové sítě. Validační množina (15-20 % dat) hodnotí přesnost sítě již během trénování. Není součástí gradientního sestupu a síť se z ní nic neučí. Používá se k ladění hyperparametrů (parametry sítě, které určuje programátor, například počet vrstev, neuronů, velikost filtrů atd.) a optimalizaci výkonu modelu. Testovací množina (zbytek dat) je využívána ke zhodnocení již naučeného modelu.

Naše datová množina obsahuje celkem 5 216 trénovacích snímků, 16 validačních snímků a 624 testovacích.

Protože ve validační množině je poměrně málo snímků, k validaci při trénování použijeme testovací množinu, abychom měli přesnější výsledky.

Nyní se podíváme na první snímky podadresářů trénovací množiny.



Obrázek 3.7: První dva snímky trénovací množiny

Můžeme si hned povšimnout, že oba snímky jsou odlišně škálováni. To není pro konvoluční neuronovou síť dobré a budeme je muset napřed předzpracovat, než je pustíme do sítě.

Obecně jakákoliv data by měla být před trénováním nejprve předzpracována, aby se trénování sítě zefektivnilo. Nejdůležitějším krokem předzpracování u obrazových dat je převedení výšky a šířky (případně i hloubky) na jeden společný rozměr a normalizace hodnot pixelů (neuronové sítě preferují vstupy z intervalu <0,1> [2]). Další kroky předzpracování závisejí na datové množině a konkrétní úloze.

Keras nám dovoluje použít třídu *ImageDataGenerator*, která nám umožní nastavit generátory, jež budou automaticky převádět adresáře se snímky na disku do dávek předzpracovaných tenzorů. Zároveň nám umožňuje nastavit další parametry pro automatickou augmentaci dat pro rozšíření datové množiny.

Pojem dávka znamená množství trénovacích dat, které je zpracováno najednou během jedné iterace gradientního sestupu při trénování sítě. Další důležitý pojem je epocha. Jedná se o dokončení celého průchodu trénovací množinou.

Pokud máme 1000 trénovacích příkladů a nastavíme velikost dávky na 100, trénování celé epochy bude probíhat v 10 iteracích. Poté se epocha opakuje a sít se trénuje dále. Počet epoch a velikost dávek jsou hyperparametry sítě a jejich hodnoty jsou v rukou programátora.

Augmentace dat je technika rozšiřování datové množiny pomocí geometrických transformací snímků.

Nyní si tedy definujeme generátory dat po naší síť.

```
from keras.preprocessing.image import ImageDataGenerator

# jednotna velikost vsech snimku
img_size = 150
# nastaveni poctu epoch a velikosti davek
epochs = 10
batch_size = 20


#instance tridy ImageDataGenerator pro predzpracovani a
    augmentaci dat.
train_datagen = image_gen = ImageDataGenerator(
                                rescale = 1./255,
                                shear_range = 0.2,
                                rotation_range=40,
                                zoom_range = 0.2,
                                horizontal_flip = True,
                                width_shift_range=0.1,
                                fill_mode='nearest')

# instance pro validacni mnozinu - jen preskalujeme hodnoty
    pixelu
val_datagen = ImageDataGenerator(
                                  rescale=1./255)

#generator dat pro trenovaci mnozinu, ktera nam bude
    generovat davky dat ze souboru.
train_gen = train_datagen.flow_from_directory(
                directory= path + 'train',
                target_size=(img_size, img_size),
                batch_size=batch_size,
                class_mode='binary',
                shuffle=True)

# generator pro validacni mnozinu
val_gen = val_datagen.flow_from_directory(
                directory= path + 'test',
                target_size=(img_size, img_size),
                batch_size=batch_size,
                class_mode='binary',
                shuffle=True)
```

Listing 3.1: definice generátorů

Generátory *train_gen* a *val_gen* nám tedy generují tenzory ve tvaru $[32, 150, 150, 3]$ tedy dávky o 32 snímcích velikosti 150x150 a 3 kanály.

Nyní, když už máme připravené generátory, můžeme nadefinovat a natrénovat naší konvoluční neuronovou síť. Jako jedna z nejefektivnějších konvolučních architektur pro klasifikaci rentgenových snímků se ukázala architektura zvaná DenseNet [13].

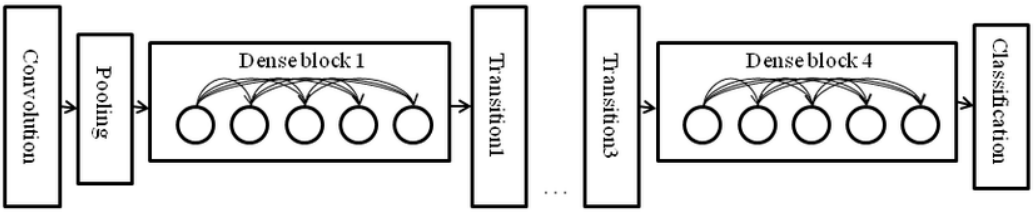
Tato architektura se skládá z několika bloků (blokem označujeme specifickou podmnožinu vrstev sítě).

Nejdůležitějším blokem je tzv. dense blok. Ten se skládá z několika podbloků, kde každý podblok obsahuje normalizační vrstvu (tj. vrstvu provádějící transformaci, která normalizuje výstupy předchozí vrstvy během učení a napomáhá tak šíření gradientu a generalizaci) a konvoluční vrstvu. Každý podblok bloku dostává jako vstup výstupy všech předchozích podbloků v bloku. Toto propojení umožňuje modelu zachytit informace z celého obrazu a zároveň pomáhá k lepšímu šíření gradientu skrze síť.

Další tzv. přechodový blok slouží ke snížení rozměrů map příznaků. Obsahuje konvoluční vrstvu následovanou vrstvou average nebo max-poolingu (viz. sekce 3.1.5). Tato vrstva snižuje velikost map příznaků a zároveň umožňuje zachování informace.

Nakonec se použije blok plně propojených vrstev, který provede samotnou klasifikaci extrahovaných příznaků.[15]

Jednoduché schéma DenseNet je vidět na obrázku 3.8.



Obrázek 3.8: Jednoduché schéma DenseNet

Jako ztrátovou funkci volíme binární křížovou entropii viz. sekce 2.5.6.

Naší jednoduché implementaci DenseNet započneme tím, že si napřed nadefinujeme funkci *dense_block* reprezentující dense blok.

```
def dense_block(x, subblocks, filters):
    for i in range(subblocks):
        #konvoluce
        x1 = Conv2D(filters,
                    kernel_size=3,
                    padding='same')(x)
```

```
        #normalizace
        x1 = BatchNormalization()(x1)
        #aktivace
        x1 = ReLU()(x1)
        #zretezeni tenzoru
        x = Concatenate()([x, x1])

    return x
```

Listing 3.2: definice dense bloků

Tato funkce má 3 vstupní parametry: 3D tenzor x, což jsou vstupní data, *subblocks* je počet podbloků dense bloku a *filters* je počet filtrů pro konvoluční vrstvy.

Funkce definuje nový tenzor *x1* a aplikuje na něj konvoluční a normalizační vrstvu a aktivační funkci ReLU, nakonec výsledný tenzor spojí se vstupním tenzorem. Toto se opakuje pro každý podblok.

Dále si definujeme funkci *transition_block* reprezentující přechodový blok.

```
def transition_block(x, filters):
    x = Conv2D(filters,
        kernel_size=1,
        padding='same')(x)
    x = BatchNormalization()(x)
    x = ReLU()(x)
    #max-pooling
    x = MaxPool2D(pool_size=2, strides=2)(x)

    return x
```

Listing 3.3: definice přechodových bloků

Tato funkce je velmi podobná, jen nemá žádné podbloky a místo spojení tenzorů aplikuje na vstup max-pooling vrstvu.

Nyní si můžeme naimplementovat celou DenseNet. Bude se skládat ze vstupního bloku, čtyř dense bloků, tří přechodových bloků a výstupního bloku s dvěma plně propojenými vrstvami (mějme na paměti, že počet dense a přechodových bloků jsou hyperparametry sítě a lze je tedy měnit při optimalizaci sítě).

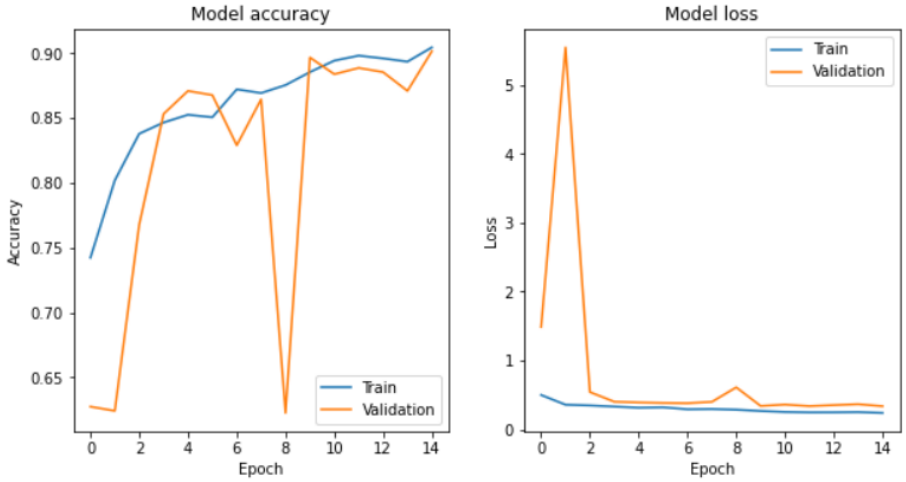
```
from keras.layers import Input, Dense, Flatten, Dropout,
   BatchNormalization
from keras.layers import SeparableConv2D, MaxPool2D,

#DenseNet Architektura

#vstupni blok
inputs = Input(shape=(img_size, img_size, 3))
```

```
x = Conv2D(16, kernel_size=3, strides=1,
           padding='same')(inputs)
x = BatchNormalization()(x)
x = ReLU()(x)
x = MaxPool2D(pool_size=2, strides=2, padding='same')(x)

#dense a prechodove bloky
x = dense_block(x, subblocks=1, filters=32)
x = transition_block(x, filters=16)

x = dense_block(x, subblocks=2, filters=64)
x = transition_block(x, filters=32)

x = dense_block(x, subblocks=1, filters=128)
x = transition_block(x, filters=64)
x = Dropout(0.1)(x)

x = dense_block(x, subblocks=1, filters=256)
x = Dropout(0.2)(x)

#plne propojeny blok
x = Flatten()(x)
x = Dense(64, activation='relu')(x)
x = Dropout(0.5)(x)
x = Dense(32, activation='relu')(x)
x = Dropout(0.3)(x)

output = Dense(1, activation='sigmoid')(x)

# Vytvoreni a zkompilovani modelu
model = Model(inputs=inputs, outputs=output)
model.compile(optimizer='Adam', loss='binary_crossentropy',
   metrics=['accuracy'])

#uceni modelu
hist = model2.fit(
            train_gen,
            steps_per_epoch=train_gen.samples // batch_size,
            epochs=epochs,
            validation_data=val_gen,
            validation_steps=val_gen.samples // batch_size,
            callbacks=[checkpoint, reduce_lr])
```

Listing 3.4: definice a trénování modelu

Pomocí funkce *model.fit()* vykonáme učení neuronové sítě. Po 10 epochách trénování skončí a mi si můžeme prohlédnout základní statistiky uložené v proměnné *hist*. Pro lepší přehlednost si tyto statistiky vizualizujeme, viz.

obrázek 3.9.



Obrázek 3.9: Vizualizace učení CNN

Všimněme si, že v prvních dvou epochách byla validační přesnost a ztráta velice vychýlená v porovnání s trénovací, to kvůli velkému počátečnímu přeučení, které se poté zmenšilo a validační křivky začaly, až na občasné fluktuace (epocha 8), sledovat ty trénovací. Nakonec přesnost klasifikace na validačních datech dosáhla v poslední epoše 90 % a validační ztráta hodnoty 0,3 (pro binární křížovou entropii je obecně dobré, když se ztráta pohybuje mezi 0 a 1, čím nižší ztráta tím lepší). Vzhledem k velikosti datové množiny jde o uspokojivý výsledek. Samozřejmě by se dal ještě zlepšit zvětšením datové množiny nebo by se dalo experimentovat s jiným nastavením hyperparametrů či jinými architekturami sítě.

Vyzkoušíme ještě jiný přístup nazývaný transfer learning. Tato metoda využívá předtrénované konvoluční neuronové sítě, konvoluční báze, naučené na jiném, větším datasetu. Jedná se o často používanou metodu, neboť naučené transformace jsou často v počátečních vrstvách velice obecné.

Ke konvoluční bázi připojíme plně propojené vrstvy a pro poslední konvoluční vrstvy konvoluční báze využijeme metody jemného dolaďování vah, kdy tyto vrstvy přizpůsobíme pro naši úlohu.

Použijeme předtrénovanou konvoluční architekturu VGG16 [16] natrénovanou na datasetu *imagenet*, což je velký vizuální dataset obsahující přes 14 milionů anotovaných obrázků.

```
from keras.applications import VGG16
conv_base = VGG16(weights='imagenet',
                  include_top=False,
                  input_shape=(img_size, img_size, 3))
```

Listing 3.5: Definice konvoluční báze

K této konvoluční bázi připojíme plně propojenou vrstvu, kterou budeme trénovat ještě před jemným dolaďováním posledních vrstev konvoluční báze. To je nutné proto, abychom si nezničili naučené transformace v konvoluční bázi. Je důležité, abychom váhy v konvoluční bázi zmrazily, aby se nemohly upravovat a zachovaly si tak naučené transformace.

```
#zmrazeni vah
conv_base.trainable = False

#plne propojeny blok
x = Flatten()(conv_base.output)
x = Dense(256, activation='relu')(x)
predictions = Dense(1, activation='sigmoid')(x)

#cely model
model2 = Model(inputs=conv_base.input, outputs=predictions)
model2.compile(optimizer='Adam',
               loss='binary_crossentropy',
               metrics=['accuracy'])

#trenovani modelu
hist2 = model2.fit(
            train_gen,
            steps_per_epoch=train_gen.samples // batch_size,
            epochs=epochs,
            validation_data=val_gen,
            validation_steps=val_gen.samples // batch_size,
            callbacks=[checkpoint, reduce_lr])
```

Listing 3.6: Definice a učení klasifikátoru s konvoluční bází

Až budeme mít klasifikátor natrénovaný, použijeme techniku jemného doladění, abychom model naučili extrahovat pro nás užitečné příznaky. Při jemném dolaďování se dolaďují i váhy plně propojené sítě, aby se na nové příznaky adaptovaly.[2]

```
conv_base.trainable = True
set_trainable = False

#rozmrazeni vah poslednich vrstev konvolucni baze
for layer in conv_base.layers:
```

```
    if layer.name == 'block5_conv1':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False

model2.compile(optimizer=keras.optimizers.RMSprop(lr=2e-5),
               loss='binary_crossentropy',
               metrics=['accuracy'])

hist3 = model2.fit(
            train_gen,.
            steps_per_epoch=train_gen.samples // batch_size,
            epochs=epochs,
            validation_data=val_gen,
            validation_steps=val_gen.samples // batch_size,
            callbacks=[checkpoint])
```

Listing 3.7: Jemné doladění

S tímto postupem jsme na našem datasetu dosáhli validační přesnosti téměř 94 % s validační ztrátu 0,21.

To byla ukázka prvního příkladu využití konvolučních neuronových sítí na medicínských datech. Nutno podotknout, že neuronové sítě používané v medicínských zařízeních jsou natrénované na mnohem větších datových sadách, používají pokročilejší architektury a předtrénované sítě, takže jejich přesnost je mnohdy ještě lepší.

3.2 Rekurentní neuronové sítě

Další významnou třídou neuronových sítí jsou rekurentní neuronové sítě (RNN).

Všechny třídy sítí, které jsme dosud zmiňovali (plně propojené a konvoluční sítě) nemají schopnost uchovávat informace o předchozích vstupech. Tedy nemají paměťovou složku. To znamená, že zpracovávají celý vstup najednou, CNN neuchovává žádný stav mezi jednotlivými příznaky vstupu. Také požadují, aby velikost vstupu byla fixní.

Existují však data, která mají sekvenční charakter. Tato data jsou zpravidla složena z řady prvků, přičemž každý prvek je závislý na předchozím a může ovlivnit následující prvky. Sekvenční data zahrnují například řečové záznamy, časové řady, průběhy signálů či text.

Pro zpracování těchto sekvenčních dat byly vytvořeny rekurentní neuronové sítě. Oproti plně propojeným a konvolučním sítím mají RNN jednu klíčovou vlastnost - paměť. To znamená, že rekurentní neuronové sítě jsou schopny uchovávat informace o předchozích vstupech a využívat je při zpracování nových vstupů. [1, 2]

3.2.1 Obecná struktura RNN

Základní charakteristikou rekurentní neuronové sítě je, že obsahuje vnitřní cyklus, který jí dovoluje iterovat přes jednotlivé prvky vstupní sekvence.

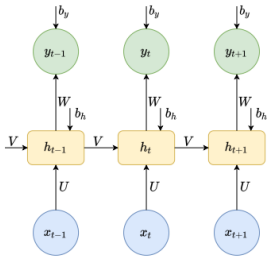
RNN zpracovává každý prvek x_t pomocí tzv. skrytého stavu h_{t-1}(kontextu) vytvořeného z předchozí iterace zpracováním předchozího prvku. Obsahuje celkem dvě až 3 matice vah U, W, V a jeden až dva prahy b_h, b_y, které jsou stejné pro každý vstupní prvek sekvence.[2] Transformační rovnice RNN, pro $t = 0, 1, ..., T$, vypadají následovně:

$$h_t = \sigma_h(Ux_t + Vh_{t-1} + b_h)$$
$$y_t = \sigma_y(Wh_t + b_y)$$

Nový skrytý stav je tedy vytvořen afinní transformací následovanou nelineární aktivací.

Bylo zmíněno, že RNN obsahuje dvě až tři matice vah a jeden až dva prahy, To je dáno tím, že se definice jednoduché RNN často lehce liší, například neobsahují druhou napsanou rovnici a platí že $h_t = y_t$ nebo, že v druhé rovnici je pro afinní transformaci použita matice U a práh b_h.

Pro lepší mentální představu rekurentních neuronových sítí si prohlídněme obrázek 3.10., který odpovídá rovnicím napsaným výše.



Obrázek 3.10: Graf RNN

Rekurentní neuronové sítě mohou pracovat ve čtyřech režimech označovaných jako vec2seq, seq2vec, obecné seq2seq a synchronizované seq2seq.

Vec2seq znamená, že RNN bere jako vstup vektor s fixní velikostí a na výstupu vrací sekvenci proměnlivé velikosti (například na vstup přijímá obrázek a na výstup generuje textový popis obrázku).

V režimu seq2vec přijímá RNN na vstup sekvenci a na výstupu vrací fixní vektor (například při klasifikaci textu přijímá RNN text a na výstupu text klasifikuje do daného počtu tříd).

Obecné seq2seq na vstupu přijímá a na výstupu generuje sekvenci. Velikost vstupní a výstupní sekvence nemusí být totožná (například strojový překlad z jednoho jazyka do jiného)

Synchronizované seq2seq znamená, že RNN přijímá jako vstup sekvenci o neznámé velikosti a na výstup produkuje sekvenci stejné velikosti jako byla vstupní sekvence (například na vstup přijímá video a na výstup klasifikuje každý rám videa). [17]

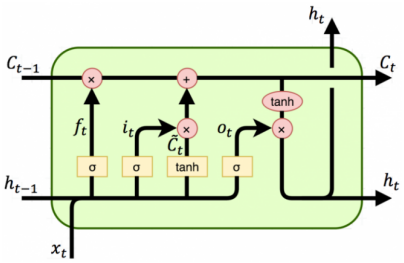
Nicméně velkým problémem, se kterým se jednoduché rekurentní neuronové sítě potýkají, je problém mizejícího gradientu, jenž způsobuje, že se síť nedokáže naučit dlouhodobé závislosti. Jinak řečeno, čím delší je vstupní sekvence, tím hlubší je RNN a tím více se problém mizejícího gradientu projevuje. Proto je pro dlouhé textové řetězce či časové řady jednoduchá RNN prakticky nepoužitelná.

Naštěstí existuje robustní RNN architektura zvaná Long short-term memory (LSTM), která problém mizejícího gradientu a s ním spojenou dlouhodobou závislost řeší. [1, 2, 5]

3.2.2 Architektura LSTM

Dalo by se říci, že LSTM je nejdůležitější RNN architektura a jedna z nejdůležitějších architektur pro zpracování sekvenčních dat celkově. Byla vyvinutá v roce 1997 Seppem Hochreiterem a Jurgenem Schmidhuberem [18] pro to, aby řešila problém mizejícího gradientu rekurentních neuronových sítí a dokázala udržet dlouhotrvající závislosti.

LSTM (stejně jako jednoduchou RNN) lze chápat jako řetězec buněk (dalo by se říci i bloků, ale terminologie okolo LSTM využívá označení buňka). Na rozdíl od jednoduché RNN, která má v každé buňce jen jednu vrstvu obsahuje buňka LSTM čtyři vrstvy. Na obrázku 3.11, můžeme vidět, jak vypadá struktura buňky LSTM Žluté čtverce představují jednotlivé plně propojené vrstvy a červená kolečka představují bodové operace.



Obrázek 3.11: Buňka LSTM

Hlavní myšlenkou LSTM je udržení datového toku řetězcem buněk, na který mohou být aplikovány jen malé lineární interakce v buňce (sčítání, násobení). Tím se mohou data dostat jednoduše přes celý řetězec významněji nezměněna.

Toho je docíleno pomocí tzv. cell state vektoru. Jedná se o nejvyšší horizontální tok na obrázku 3.11. Mechanizmus, jak ovlivňovat cell state vektor a využívat těchto informací pro výstup z buňky je realizován pomocí tří bran (gates).

První brána tzv. forget gate generuje rozhodnutí o tom, které informace v cell state vektoru mají být ponechány a které by měly být zapomenuty. V rovnici pro tuto bránu se používá plně propojená vrstva se sigmoidální aktivační funkcí, která výstup transformuje do intervalu (0, 1).

Vzorec pro forget gate bránu vypadá následovně:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

Kde W_f je váhová matice a b_f je práh pro forget gate, $[h_{t-1}, x_t]$ znamená zřetězení vektoru skrytého stavu z předchozí buňky se vstupním vektorem sekvence. Výstup f_t z forget gate brány je váhový vektor, kde každá hodnota reprezentuje důležitost hodnoty v cell state vektoru. Vyšší hodnoty v f_t znamenají, že informace v cell state vektoru jsou důležité a nebudou zapomenuty, zatímco nižší hodnoty f_t znamenají, že informace v cell state vektoru jsou méně důležité a budou zapomenuty. Celkově tedy forget gate umožňuje LSTM sítím efektivně filtrovat důležité informace ze vstupů a předchozích stavů a potlačovat nepodstatné informace.

Druhá brána se nazývá input gate. Tato brána rozhoduje o tom, jakou novou informaci budeme chtít uložit do cell state vektoru.

Vzorce input gate v LSTM lze zapsat jako:

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$
$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

První rovnice počítá vektor $\tilde{C}_t$ značící novou informaci, která má být zapamatována. Tento vektor se vypočítá jako afinní transformace (plně propojená síť) vstupu následována hyperbolickým tangensem jako aktivační funkcí.

V druhé rovnici se počítá vektor i_t s hodnotami v intervalu (0, 1), který udává, jak moc se má nová informace (reprezentovaná $\tilde{C}_t$) použít pro aktualizaci cell state vektoru.

Dosavaď vypočítané vektory f_t, i_t a $\tilde{C}_t$ představují rozhodnutí buňky, jak pozměnit cell state vektor. Následující rovnice tato rozhodnutí provede.

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

Kde $f_t \cdot C_{t-1}$ provede rozhodnutí o zapomenutí nedůležitých informací z předchozího cell state vektoru a $i_t \cdot \tilde{C}_t$ vybere, které nové hodnoty z $\tilde{C}_t$ mají být přidány do nového cell state vektoru.

Poslední, třetí brána nazývaná output gate počítá nový skrytý stav (a výstup buňky) na základě nového cell state vektoru.

Rovnice output gate vypadají následovně:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \cdot tanh(C_t)$$

Nejdříve se vypočítá vektor o_t a na základě předchozího skrytého stavu a vstupního vektoru. Tento vektor se následně vynásobí vektorem nového cell state vektoru, který byl předtím ještě transformován hyperbolickým tangensem. Tím získáme nový skrytý stav, který půjde (spolu s novým cell state vektorem) do následující buňky LSTM. [19]

Díky této struktuře dokáže LSTM uchovávat dlouhodobé závislosti a překonala tak nepoužitelnost RNN pro dlouhé sekvence. Existuje více variant LSTM nebo architektur, které z LSTM vychází, a které se mírně liší ve struktuře buňky, ale princip, který klasická LSTM přinesla zůstává.

3.2.3 Využití RNN v medicíně

Rekurentní neuronové sítě mají v medicíně a zdravotnictví obecně mnoho aplikací. V medicíně jsou využívané například díky své schopnosti analyzovat biologické signály, jako jsou elektrokardiogramy (EKG) nebo elektroencefalogramy (EEG).

Například v oblasti kardiologie mohou RNN pomoci detekovat srdeční arytmie a predikovat riziko srdečních onemocnění. V oblasti neurologie mo-

hou RNN pomoci identifikovat epileptické záchvaty nebo predikovat vývoj Alzheimerovy choroby.

Dalším významným využitím RNN je analýza elektronických zdravotnických záznamů (EHR). Například v roce 2018 byla provedena studie, která se zabývala použitím rekurentních neuronových sítí k detekci lékařských událostí v elektronických zdravotnických záznamech. EHR obsahuje mnoho informací jako jsou výsledky laboratorních testů, záznamy o vitálních funkcích, diagnostické zprávy, léky, chirurgické zákroky, atd, které jsou zaznamenávány v nelineárních sekvencích, což ztěžuje tradiční metody analýzy dat. RNN však dokáží efektivně modelovat takové sekvence a identifikovat v nich klíčové lékařské události.

Autoři v článku popisují použití této metody k detekci tří různých typů událostí v EHR: alergických reakcí, krvácení a infarktů. Výsledky experimentů ukázaly, že navrhovaná metoda dosáhla vysoké přesnosti v identifikaci těchto událostí.

Konkrétně byla dosažena přesnost detekce alergických reakcí 95,5 %, krvácení 96,2 % a infarktů 94,7 %. Autoři také porovnali svou metodu s jinými metodami strojového učení pro detekci lékařských událostí v EHR a ukázali, že navrhovaná metoda dosahuje lepších výsledků než tyto metody.

Celkově tedy tento článek popisuje úspěšné použití RNN k detekci lékařských událostí v EHR a ukazuje, že tato metoda může být velmi užitečná pro zlepšení kvality a efektivity zdravotnické péče. [20]

Využití RNN v medicíně má obecně velký potenciál a v budoucnosti může přinést mnoho inovativních způsobů, jak diagnostikovat a léčit nemoci.

3.2.4 Příklad 2: klasifikace EKG signálů pomocí LSTM

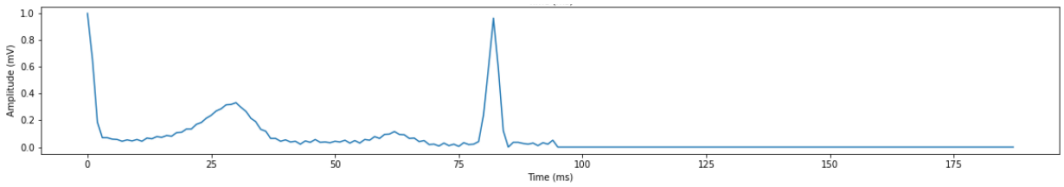
Jako druhý praktický příklad si ukážeme multi-class klasifikaci EKG záznamů signálů do 5 tříd pomocí RNN architektury LSTM. Cílem tohoto příkladu je demonstrovat využití rekurentních neuronových sítí v lékařské diagnostice. Konkrétně se nyní budeme zabývat analýzou a klasifikací EKG záznamů elektrické aktivity srdce.

Datová množina je organizována do dvou CSV souborů (mitbih_train, mitbig_test). Množina obsahuje elektrokardiografické (EKG) záznamy srdce s kategoriemi 5 různých srdečních stavů N, S, V, F, Q, kde N jsou normální a zdravé srdeční tepy, S jsou supraventrikulární ektopické tepy neboli srdeční tepy, které vycházejí nad komorami a objevují se dříve, než se očekávalo, V jsou srdeční tepy vycházející z komor a objevují se dříve, než se očekávalo, F jsou sdružené tepy nebo neznámé tepy, které jsou sloučením normálních a abnormálních tepů nebo srdeční tepy, které nelze s jistotou klasifikovat jako

normální, supraventrikulární nebo ventrikulární a Q jsou tepy, jež nelze s jistotou zařadit do žádné z ostatních čtyř kategorií.

Celkový počet EKG záznamů signálů je 109 446 (87 554 pro trénování a 21 892 pro validaci/testování) s jednotnou délkou 188 vzorků (pokud se záznam do této délky nevešel, byl doplněn nulami). Vzorkovací frekvence je 125Hz. Signály byly získány z 47 různých pacientů, kteří měli různé typy srdečních arytmií. Značky jednotlivých tříd jsou vždy uvedeny jako poslední prvek konkrétního EKG záznamu. Datová množina je obsažena v příloze bakalářské práce v adresáři ECG_dataset a zároveň je volně dostupná na webu Kaggle:
`https://www.kaggle.com/datasets/shayanfazeli/heartbeat`

Data načteme z CSV souborů pomocí knihovny Pandas. Pro představu si na začátek vizualizujeme, jak vypadá jeden záznam signálu.



Obrázek 3.12: Vizualizace EKG signálu. Osa x představuje čas(ms), osa y představuje amplitudu signálu (mV)

Tento signál je reprezentován 188 rozměrným vektorem s hodnotami z intervalu od 0 do 1, díky čemuž nám odpadá potřeba normalizace hodnot pro tuto datovou množinu.

Jak již bylo uvedeno, budeme provádět klasifikaci signálů do pěti různých tříd podle srdečních stavů. Protože rozložení záznamů v třídách je silně nerovnoměrné (přes 82 % všech záznamů patří do třídy N a jen 0,2 % do třídy F), což může negativně ovlivnit výsledný model, musíme četnost záznamů narovnat. Abychom se tohoto problému zbavili, použijeme jednoduchou, ale účinnou doporučenou techniku zvanou *resampling*.

Nejprve je nutné určit pevný počet záznamů, který bude mít každá třída, poté podle ve všech třídách, které tohoto počtu nedosáhnou, náhodně duplikujeme existující záznamy. Protože duplikujeme záznamy v některých třídách, musíme pečlivě zvážit, jaký pevný počet záznamů zvolíme. Bude-li to příliš vysoký počet, hrozí u některých tříd až přílišné množství duplikovaných záznamů, bude-li to příliš malý počet, hrozí že bude velké množství záznamů z některých tříd nevyužito (v úlohách s podobnou datovou množinou lze i výběr počtu záznamů ve třídách považovat za jeden z hyperparametrů, se

kterým lze experimentovat).

My si zvolíme hodnotu 20 000 záznamů pro každou třídu a pomocí knihoven Pandas a scikit-learn upravíme trénovací množinu.

```
from sklearn.utils import resample

# r_i je vlastni dataset pro kazdou tridu
r1=data_train[data_train[187]==1]
r2=data_train[data_train[187]==2]
r3=data_train[data_train[187]==3]
r4=data_train[data_train[187]==4]
r0=(data_train[data_train[187]==0]).sample(n=20000,
    random_state=20)

# duplikace hodnot v kazdem datasetu, ktery nedosahuje 20 000
     zaznmu
r1_upsample=resample(r1,replace=True,n_samples=20000,
    random_state=21)
r2_upsample=resample(r2,replace=True,n_samples=20000,
    random_state=22)
r3_upsample=resample(r3,replace=True,n_samples=20000,
    random_state=23)
r4_upsample=resample(r4,replace=True,n_samples=20000,
    random_state=24)

#spojeni mnozin zpet do hladniho trenovaciho datasetu
data_train=pd.concat([r0,r1_upsample,r2_upsample,r3_upsample,
    r4_upsample])
```

Listing 3.8: Resample datove mnoziny

Po úpravě datové množiny extrahujeme a do numpy tenzoru uložíme cílové výstupy (označení tříd pro každý záznam). Poté převedeme cílové výstupy všech záznamů na one-hot vektory. Jedná se o reprezentaci, kde každé označení třídy je vektor o rozměru rovnému počtu tříd úlohy. Každý vektor je nulový s jedinou jedničkou na místě, které reprezentuje třídu.

Pro představu pro záznam x, který patří do třídy S (máme celkem 5 tříd, tedy vektor (N, S, V, F, Q)) bude jeho výstup reprezentován pětirozměrném one-hot vektorem (0, 1, 0, 0, 0). One-hot reprezentace se hodí, pokud je počet tříd malý a umožňuje efektivní porovnání predikce s cílovým výstupem, protože se jedná o dva binární vektory, které lze snadno porovnat pomocí metrik jako je například křížová entropie. [5,21]

Keras poskytuje funkci *to_categorical()*, která převede vstupní tenzor na one-hot reprezentaci.

```
from keras.utils import to_categorical
# extrakce vystupu
y_train=data_train[187]
y_test=data_test[187]

# prevod na one-hot repreyentaci
y_train = to_categorical(y_train)
y_test = to_categorical(y_test)
```

Listing 3.9: Převod na one-hot reprezentaci

Před definováním a trénováním modelu ještě musíme převést vstupní data na tenzory tvaru požadovaného pro LSTM. 2D tenzory tvaru (počet záznamů, počet příznaků) transformujeme do 3D tenzoru (počet záznamů, časové kroky, počet příznaků).

```
#prevedeme si data z pandas dataframu na numpy pole a
   vypustime posledni sloupec (jde o oznaceni tridy)
x_train = data_train.iloc[:, :-1].values
x_test = data_test.iloc[:, :-1].values

# transformace na vhody tvar pro LSTM (pocet zaznamu, casov
    kroky, pocet priznaku)
x_train = np.reshape(x_train, (x_train.shape[0], 1, x_train.
   shape[1]))
x_test = np.reshape(x_test, (x_test.shape[0], 1, x_test.shape
   [1]))
```

Listing 3.10: Transformace vstupních tenzorů

Časové kroky nám říkají, v kolika krocích budeme zpracovávat jednu sekvenci. Náš původní tenzor měl tvar (100000, 187), tedy 100 000 záznamů, každý s délkou 187 vzorků. Transformací na tenzor (100000, 11, 17) máme 100 000 záznamů, které budeme postupně zpracovávat po 17 vzorcích v 11 krocích.

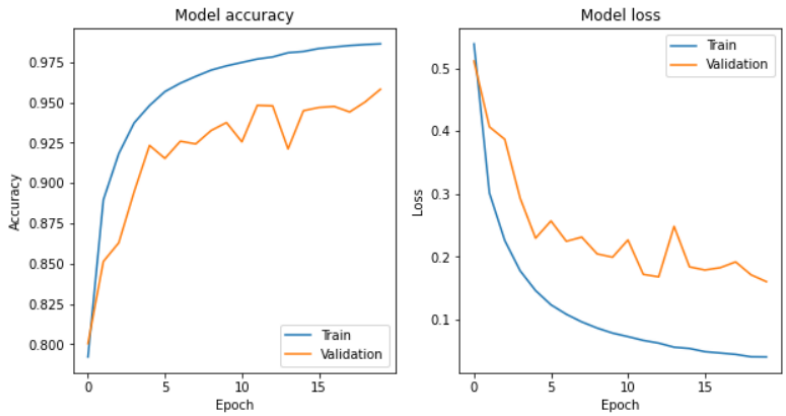
Nyní přejdeme k definici a natrénování jednoduché LSTM sítě. Síť je tvořena ze dvou LSTM bloků a jedné plně propojené vrstvy pro klasifikaci. Jako ztrátovou funkci použijeme křížovou entropii, která je vhodná pro klasifikaci do více tříd.

```
from keras.models import Model
from keras.layers import Input, Dense, LSTM
from keras.callbacks import ModelCheckpoint

# LSTM
inputs = Input(shape=(x_train.shape[1], x_train.shape[2]))
x = LSTM(64, return_sequences=True)(inputs)
```

```
x = LSTM(32)(x)
outputs = Dense(5, activation='softmax')(x)

model = Model(inputs=inputs, outputs=outputs)
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
model.summary()

#callback pro ulozeni nejlepsich parametru na validacnich
    datech behem uceni
checkpoint = ModelCheckpoint(filepath='best_weights2.hdf5',
                             save_best_only=True,
                             save_weights_only=True)

# trenovani modelu
history = model.fit(x_train,
                    y_train,
                    validation_data=(x_test, y_test),
                    epochs=20,
                    batch_size=16,
                    callbacks=[checkpoint])
```

Listing 3.11: Definice a trénování LSTM modelu

Když je trénování u konce, vizualizujeme si, z informací uložených v proměnné *history*, jeho průběh.



Obrázek 3.13: Trénování LSTM

Můžeme se všimnout, že přesnost na trénovacích datech dosáhla téměř 99 % a na validačních datech téměř 96 %. To už jsou poměrně pěkné výsledky.

3.3 Transformery

Poslední třídou neuronových sítí, která bude v této práci představena je třída transformerů.

Transformery byly poprvé představeny v roce 2017 v článku *Attention is all you need* [22]. Tyto neuronové sítě byly původně navrženy pro seq2seq úlohu strojového překladu textů a ukázalo se, že dosahují mnohem lepších výsledků než RNN architektury.

Díky těmto výsledkům se transformery dostali do středu zájmu odborníků na zpracování přirozeného jazyka a stali se hlavním nástrojem v této oblasti. Využívají se v jazykovém modelování a generování textu, strojovém překladu, sumarizaci textu a dokonce i v úlohách zpracování obrazu. [23]

Výhodou transformerů je, že oproti RNN se například lépe vypořádávají s mnohem delšími sekvencemi a nevyžadují ukládání vnitřního stavu. Další výhodou je, že výpočet lze paralelizovat. RNN zpracovává prvky sekvence postupně, proto je učení na dlouhých sekvencích velmi pomalé, možnost paralelizace (zpracování všech prvků sekvence současně) výrazně urychluje učení transformerů.[23]

Na druhé straně však potřebují velké množství dat pro trénování a jsou více náročné na výpočetní zdroje.

3.3.1 Obecná struktura transformerů

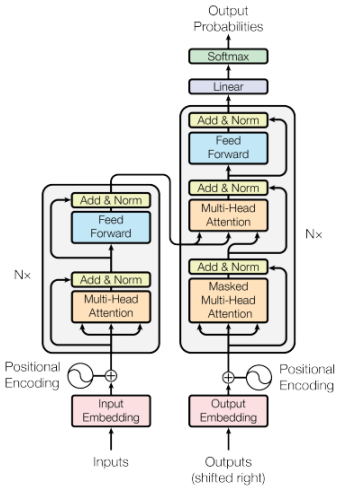
Transformery mají encoder-decoder strukturu, která je typická pro seq2seq úlohy.

Encoder-decodery jsou neuronové sítě, které se skládají ze dvou bloků - encoderů a decoderů. Encoder postupně přijímá vstupní sekvenci ve formě embeddingů, což je vektorová reprezentace prvků vstupu (jedno slovo je jeden vektor), a produkuje užitečnou abstraktní vnitřní vektorovou reprezentaci vstupu tzv. encoding. Decoder přijímá encoding a na základě něho autoregresivně generuje výstupní prvky, tzn. že již vygenerované prvky s enkodingem použije jako další vstup pro generování dalšího prvku. Jinými slovy se decoder učí, jak předvídat další prvek sekvence na základě výstupů, které již vygeneroval, a encodingu.[5, 22]

Transformer následuje tuto obecnou architekturu a přidává do ní poziční kódování vstupu a zavádí mechanismus multi-head attention, to mu dovoluje zpracovávat celou sekvenci paralelně najednou a pro každý prvek sekvence vypočítat kontext neboli důležitost ostatních prvků v sekvenci.

Obecná struktura neuronové sítě transformer je vidět na obrázku 3.14. Skládá se z dvou logických bloků, které se nazývají blok encoderů a blok

decoderů. Encodery i decodery obsahují multi-head attention vrstvy a plně propojené vrstvy.

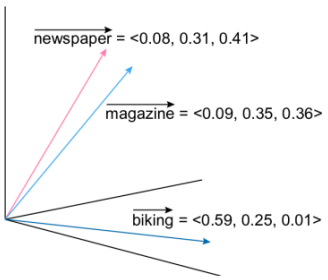


Obrázek 3.14: Struktura transformeru

Nyní si pojďme popsat jednotlivé části podrobněji.

3.3.2 Vstup do encoderu a poziční kódování

Prvním krokem je převedení prvků sekvence (slov) do matematické podoby, se kterou bude umět síť pracovat a zároveň bude v této podobě obsažená sémantika slov. Word embedding neboli vnoření slov, je technika, která tento převod provádí. Slovo je reprezentováno sémantickým vektorem v mnohorozměrném sémantickém prostoru, kterému se říká embedding. Geometrické vlastnoti vektorů v tomto prostoru potom odráží ty sémantické. Platí, že vektory významově blízkých slov jsou ve vytvořeném prostoru blízko u sebe, tedy že jejich kosínová vzdálenost je malá, jak ilustruje obrázek 3.15. Pro transformaci slov na sémantické vektory se využívá (většinou) předtrénovaná neuronová síť (v transformeru tato síť představuje první vrstvu). [2, 5]



Obrázek 3.15: Sémantický prostor

Protože encoder transformeru nemá rekurenci jako rekurentní neuronové sítě, musí se do sémantických vektorů přidat nějaké informace o pozicích prvků v sekvenci. To se provádí pomocí pozičního kódování. Pozičního kódování vytváří vektory obsahující informace o relativní pozici jednotlivých slov ve vstupní sekvenci.

Poziční kódování je efektivní, protože relativní pozice mezi slovy je významnější než jejich absolutní pozice. Vědět, že slovo „neuronová“ je na indexu 6 a slovo „síť“ je na indexu 5, je méně důležité než si uvědomit, že „neuronová“ často předchází slovo „síť“.

Pro každou pozici slova v sekvenci se vytvoří vektor, který se skládá z hodnot získaných aplikací sinu a cosinu o různých frekvencích. Každá frekvence odpovídá jedné složce vektoru.

Tyto hodnoty jsou vypočítány podle vzorců:

$$PE_{(pos,2i)} = sin(pos/10000^{\frac{2i}{d}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{\frac{2i}{d}})$$

Kde pos je pozice slova v sekvenci, i je index do výsledného vektoru, a d je rozměr sémantického vektoru.

Výsledný vektor pozičního kódování se pak přičte k sémantickému vektoru slova, čímž vznikne jeden vektor reprezentující význam slova a obsahující informaci o jeho relativní pozici v sekvenci. [5,22]

3.3.3 Multi-head attention

Nejdůležitější částí transformeru je multi-head attention mechanismus. Multi-head attention spočívá ve výpočtu kontextu důležitého pro jednotlivé prvky sekvence. Tento mechanismus umožňuje přidělit větší váhu důležitým prvkům a ignorovat méně důležité prvky.

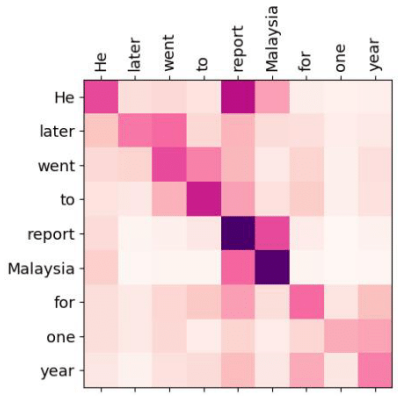
Než si vysvětlíme, co je to multi-head attention, musíme si nejdříve vysvětlit, co je to self attention, ze kterého se multi-head attention skládá.

Self attention mechanismus pracuje se třemi maticemi $Q \in R^{n*d_k}$ (querys), $K \in R^{n*d_k}$ (keys) a $V \in R^{n*d_v}$ (values). Tyto matice jsou získány lineární transformací (plně propojenou vrstvou bez aktivační funkce) vstupní matice $X \in R^{n*d}$ (matice složená ze vstupních vektorů).

Tedy:

$$Q = XW^Q$$
$$K = XW^K$$
$$V = XW^V$$

Kde $W^Q, W^K \in R^{d*d_k}$ a $W^V \in R^{d*d_v}$ jsou trénovatelné matice vah. Poté vynásobením matice Q s transponovanou maticí K^T získáme tzv. attention skóre, což je $n * n$ rozměrná matice (n odpovídá počtu vstupních vektorů), která pro každý prvek vstupní sekvence přiřazuje důležitost všech ostatních prvků, jak je vidět na obrázku 3.16.



Obrázek 3.16: Attention skóre

Hodnoty attention skórů jsou škálovány odmocninou ze sloupcové dimenze Q, K, tedy hodnotou $\sqrt{d_k}$, aby se zamezilo explozivnímu růstu hodnot a zajistil se lepší výpočet gradientů [22]. Nakonec se na attention skóre aplikuje softmax funkce, která zajistí, že všechny hodnoty budou z intervalu (0,1) a budou představovat váhy pro hodnoty v matici V. Výstupem je potom vážená $n * d_v$ matice neboli n abstraktních vektorů o rozměru d_v.

Celý tento proces lze shrnout jednou rovnicí:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V = softmax(\frac{(XW^Q)(XW^K)^T}{\sqrt{d_k}})(XW^V)$$

A co je to tedy multi-head attention? Multi-head attention spočívá v paralelním provedení několika menších self attention neboli v několika lineárních zobrazeních Q, KaV do různých reprezentačních prostorů.

O matici attention score lze uvažovat jako a filtru, který pro každý prvek ze sekvence filtruje pro nej zajímavé části (kontext). Myšlenka za multi-head

attention je vytvořit více menších filtrů, které budou filtrovat pokaždé jiný důležitý kontext. Díky domu dokáže transformer extrahovat více informací z různých reprezentací téže sekvence.

Multi-head attention lze matematicky vyjádřit pomocí následující rovnice:

$$MultiHead(Q, K, C) = Concat(head_1, ..., head_h)W^O$$

kde:

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

Kde lineární zobrazení se provádí pomocí (trénovatelných) matic $W_i^Q, W_i^k \in R^{n*d_k}, W_i^V \in R^{n*d_v}$. Všechny vypočítané self attention (víme, že se jedná o matice), kterým se v tomto případě říká heads (hlavy) se zřetězí do jedné velké $n * hd_v$ matice, kde h představuje počet hlav. Posledním krokem je lineární transformace této matice pomocí (trénovatelné) matice vah $W^O \in R^{hd_v*n}$ čímž získáme výstupní matici z multi-head attention, která obsahuje informace z různých kontextů.

Multi-head attention vyskytuje jednou v encoderu a dvakrát v decoderu. V encoderu se počítá mezi prvky vstupní sekvence. V decoderu se počítá mezi prvky výstupní sekvence (masked multi-head attention), kde se v attention score matici "zamaskují"prvky, které ještě nebyly vygenerovány a druhá multi-head attention se počítá mezi prvky vstupní a výstupní sekvence. [22,23]

Díky multi-head attention mechanismům je transformer schopen zvládat i příklady, kdy například stejný prvek (stejné slovo) je sekvenci sekvenci použit opakovaně, avšak pokaždé v jiném kontextu a s jiným významem. Zároveň dokáže celkový kontext sekvence udržovat mnohem déle než dokáží rekurentní neuronové sítě a je tady schopen zpracovávat mnohem delší sekvence.

3.3.4 Zbylé vrstvy transformeru

Zbylé vrstvy, které můžeme na schématu transformeru (obr. 3.14) vidět, jsou plně propojené vrstvy (feed forward), které už však známe a vrstvy reziduální a normalizační (add and norm).

Reziduální vrstva spočívá v přičtení výstupu některé z předchozích vrstev v síti k výstupu poslední vrstvy. To dovoluje sítí udržovat informace, které by jinak zapomněla.

Normalizační vrstva, jak bylo nastíněno v prvním praktickém příkladu, slouží k lineární transformaci, konkrétně normalizaci, výstupu předchozí

vrstvy, tak aby všechny hodnoty měli střední hodnotu $\mu = 0$ a směrodatnou odchylku $\sigma = 1$.

Poslední softmax vrstva převádí výstup transformeru na pravděpodobnosti. Prvek, který má na základě kontextu nejvyšší pravděpodobnost bude vygenerován.[22,23]

3.3.5 Využití transformerů v medicíně

Díky své neuvěřitelné efektivitě v oblasti zpracování přirozeného jazyka mají transformery v medicíně velmi slibnou budoucnost.

Tradiční úlohou zpracování přirozeného jazyka je klasifikace textu. Transformery mohou být v medicíně využity pro klasifikaci klinických textových dokumentů na základě jejich obsahu. Klasifikace může být různá, lze klasifikovat elektronické dokumenty pacientů podlé stanovené diagnózy, podle podávaných léků či obecně podle různých informací obsažených v textu.

Transformery by mohly podle informací v dokumentech dokonce prvotní diagnózu sami vykonávat.

Další důležitou úlohou zpracování přirozeného jazyka je sumarizace textu. To se dá využít v medicíně pro sumarizaci rozsáhlých klinických dokumentů, což může významně urychlit lékařům hledání a studium potřebných informací.

Transformery se v dnešní době používají pro tvorbu velkých jazykových modelů a chatbotů (například známý chatGPT). Takový chatboti (samozřejmě velmi dobře naučení na relevantní datové množině) mohou být velice užiteční při poskytování základních informací a odpovědí pacientům.

Obecně jsou nyní transformery velmi studovanou třídou neuronových sítí, a dokonce už existují i architektury transformerů, které se využívají pro zpracování obrazu, tzv. ViT (vision transformer). Lze tedy předpokládat, že využití transformerů v různých oblastech medicíny bude přibývat.

Vzhledem k malému množství volně dostupných relevantních textových medicínských dat, nebude k transformerům uveden žádný příklad.

Tím jsme dokončili třetí kapitolu zabývající se nejdůležitějšími třídami neuronových sítí (nikoliv všech) a jejich aplikací v medicíně.

Ve čtvrté kapitole podíváme na hlavní praktický příklad této práce.

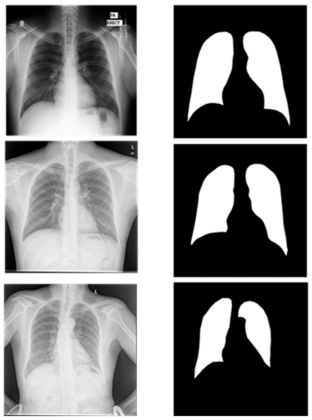
4 Segmentace CT snímků mozku

Jako poslední a hlavní praktický příklad si ukážeme segmentaci CT snímků mozku. Cílem tohoto příkladu je demonstrovat využití hlubokého učení pro segmentaci medicínských snímků.

4.1 Obecná definice úlohy segmentace obrazu

Segmentace obrazu patří mezi základní úlohy v oblastech zpracování obrazu a počítačového vidění. Spočívá v rozdělení obrazu do jednoduše rozlišitelných oblastí (oblasti zájmu a případně pozadí).

Oblast zájmu je cokoliv, co chceme segmentovat. Může se jednat o jediný objekt (segmentace nádoru na snímku mozku), více stejných objektů (segmentace krevních buněk) nebo segmentace více rozdílných objektů obrazu (segmentace orgánů na snímku lidského těla). Pro ilustraci je na obrázku 4.1. vidět segmentace plic.



Obrázek 4.1: Příklad segmentace plic

Pro segmentaci se využívají znalosti o řešeném problému. Platí, že čím více se o problému ví, tím lépe. Lze totiž provést lepší předzpracování a vybrat nejvhodnější metodu segmentace. [24]

4.1.1 Klasické metody segmentace

Protože je segmentace častou a důležitou úlohou, existuje spousta metod, jak ji provádět. Mezi nejzákladnější metody patří prahování, detekce hran či segmentace nárůstem oblastí.

Prahování je nejjednodušší a nejstarší segmentační metoda. Spočívá ve volbě prahu T a rozdělení obrazu na dvě oblasti.

$$S(i,j) = \begin{cases} 0 & \text{pro } I(i,j) < \mathrm{T} \\ 1 & \text{pro } I(i,j) \geq \mathrm{T} \end{cases} \tag{4.1}$$

Kde S je nový vysegmentovaný obraz a I je původní obraz. Prahování lze modifikovat volbou n prahů a rozdělením obrazu na n oblastí.

Výhodou tohoto přístupu je jednoduchá implementace a nízké výpočetní nároky. Nevýhodou je, že tuto metodu lze použít pouze na určitou třídu obrazů (objekty jsou v obraze snadno jasově rozlišitelné). [24]

Detekce hran je metoda založená na (jak název napovídá) detekci hran tvořících hranice objektů v obraze.

Každý objekt je popsán hranicí, která se skládá z hran. Hranou se obvykle rozumí místa obrazu, kde dochází k určité nespojitosti, většinou v jasu, ale také v barvě, textuře, hloubce apod [24].

Pro detekci hran existuje spousta operátorů jako je například Sobelův operátor (filtr), který byl představen v sekci 3.1.3.

Segmentace na bázi detekce hran je výpočetně rychlá a levná, avšak je velmi náchylná na šum a velmi složité textury. [9, 24]

Segmentace nárůstem oblastí je založena v rozdělení obrazu na největší souvislé oblasti, které jsou z hlediska určité charakteristiky homogenní. Homogenita může být založená například na jasových vlastnostech.

Přirozenou metodou spojování oblastí je vycházet z počátečního rozložení, kdy každý obrazový prvek představuje samostatnou oblast. Postupně se budou spojovat dvě sousední oblasti, pokud výsledná oblast bude splňovat kritérium homogenity.

Tradičních metod segmentace obrazu je mnohem více, ale popsat všechny tyto metody není účelem této práce.

4.1.2 Segmentace obrazu pomocí hlubokého učení

Metody hlubokého učení pronikly i do úlohy segmentace obrazu a přinesly velmi pozoruhodné výsledky, které v mnohém překonaly tradiční segmentační metody a posunuly tuto oblast dopředu.

Hloubkové učení je v současné době velmi populární v segmentaci obrazu, protože dokáže automaticky extrahovat relevantní příznaky z obrazových dat a naučit se složité reprezentace.

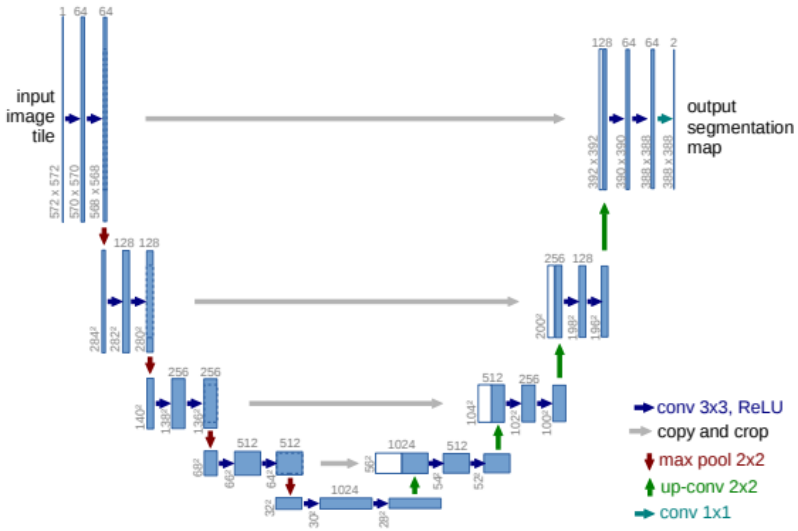
Pomocí hlubokého učení lze dosáhnout segmentace i velmi složitých a nehomogenních oblastí.

Většina segmentačních metod je založena na učení s učitelem, kdy pro učení modelu máme k dispozici obrazy, které chceme segmentovat a požadované segmentační masky.

Úkolem neuronové sítě je pak naučit se extrahovat takové příznaky, pomocí kterých dokáže provést segmentaci neboli klasifikaci jednotlivých pixelů do tříd (např. binární klasifikace pixelů na jeden objekt zájmu a pozadí).

Existuje spousta architektur, které se používají pro segmentaci. Nejčastěji jsou založeny na konvolučních vrstvách a reziduálních spojeních (vrstvách). [25,26]

Z pohledu medicínské segmentace se dnes již jako zlatý standard považuje architektura Unet (obr.4.2). Tato architektura v době svého vzniku v roce 2015 dosáhla s výrazným náskokem nejlepší přesnosti v ISBI Cell Tracking Challenge, což je mezinárodní soutěž v segmentaci buněk z mikroskopických snímků a od té doby se stala jednou z nejčastěji používaných architektur v této oblasti. [27]

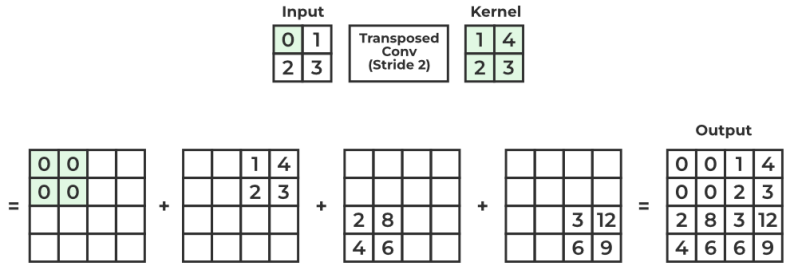


Obrázek 4.2: Unet architektura

U-Net vychází z plně konvolučních sítí a z encoder-decoder struktury. Skládá se ze dvou symetrických cest: kontrakční cesta (encoder) a expanzivní cesta.

Kontrakční cesta se skládá z několika vrstev konvoluce a Max-pooling vrstev, které slouží k redukci rozměrů vstupního obrazu a získání abstraktních příznaků. Tyto příznaky jsou následně využity v expanzivní části, která je podobná té kontrakční, ale nahrazuje Max-pooling operaci tzv. nadvzorkovací operací (up-sampling). Tato operace pomáhá zvýšit rozměry příznaků a vrátit se k původnímu rozměru vstupního obrazu. V každé vrstvě expanzivní cesty jsou kromě nadvzorkovací operace také transponované konvoluční vrstvy, které propojují informace z kontrakční cesty s expanzivní cestou.

Transponovaná konvoluce funguje tak, že každý prvek vstupní matice rozšíří na matici větší velikosti než konvoluční jádro, přičemž jednotlivé prvky v této matici jsou vynásobeny odpovídajícími prvky konvolučního jádra (viz. obr. 4.3). všechny nové matice jsou sečteny do jedné výstupní.



Obrázek 4.3: Transponovaná konvoluce

Nakonec v poslední vrstvě Unet je použita tzv. softmax aktivační funkce (případně sigmoida pro binární segmentaci), která přiřazuje každému pixelu vstupního obrazu pravděpodobnosti příslušnosti k jednotlivým třídám. Konkrétně se v této vrstvě vytvoří výstupní mapa příznaků o stejné velikosti jako vstupní obraz, kde každý pixel odpovídá vektoru pravděpodobností jednotlivých tříd. [27]

V současné době je Unet architekturou první volby, pokud se jedná o medicínskou segmentaci, nicméně existuje spousta architektur odvozených od klasické Unet jako je například Unet++, 3D Unet, Xception Unet či Attention Unet, které mohou být lepší volbou pro obtížnější úlohy.

My pro náš úkol budeme také vycházet z architektury Unet.

4.2 Popis datové množiny

Datová množina je organizována do dvou adresářů *in* a *out*. V adresáři *in* se nachází 26 podadresářů, přičemž každý podadresář obsahuje jeden CT sken ve formě jednotlivých řezů, kde každý řez je uložený v samostatném DICOM souboru. Adresář *out* má stejnou strukturu, jen místo DICOM souborů jsou řezy s označenými oblastmi uloženy v .png formátu.

Data jsou pořízena z CBF série pacientů získané mezi 2021-01-01 až 2021-02-01. K detekci a označení zájmových oblastí jednotlivých řezů byl využit *software InfarctionCoreDelineator* s doporučenými parametry:

InfarctionCoreDelineator.exe in out c 7 7 1 0.3 0

Kde c je CBV mapa (mapa cerebrálního krevního objemu), 7 je sousedství na ose x (7 pixelů),7 je sousedství na ose y (7 pixelů), 1 je sousedství na ose z (1 řez), 0.3 je úroveň prahu a 0 je polynomický stupeň aproximace. [28]

Skeny byly původně v jednom DICOM souboru, ale protože *InfarctionCoreDelineator* potřebuje na vstup sérii DICOM souborů, byl na transformaci 3D skenu na sérii 2D řezů použit software 3D Slicer, což je bezplatný software pro zpracování, vizualizaci a analýzu lékařských obrazových dat podporující vedle standardních formátů i mnoho formátů používaných v medicíně (např. DICOM, NIfTI či NRRD) [29].

V 3D Sliceru se načetla série a následně pomocí pluginu *Data->Export to DICOM->Scalar Volume* se vytvořila nová složka obsahující DICOM soubory jednotlivých řezů.

4.2.1 Formát DICOM

DICOM (Digital Imaging and Communications in Medicine) je standardizovaný formát pro ukládání a přenos medicínských obrazových dat, jako jsou CT snímky, MRI nebo rentgenové snímky.

DICOM formát obsahuje následující základní prvky:

1. Hlavička souboru - obsahuje informace o formátu souboru a o metadatach, která jsou uložena v souboru.

2. Metadata - obsahují informace o pacientovi (jako je například jméno, datum narození, pohlaví), o zobrazovaném obrazu (jako je například typ zobrazovaného orgánu nebo použitá technologie) a o vyšetření (jako jsou například datum vyšetření, použitá dávka záření).

3. Obrazová data - obsahují samotné obrazové informace, které jsou reprezentovány jako matice pixelů, kde každý pixel má svou hodnotu v závislosti na intenzitě světla.

DICOM formát umožňuje ukládat a přenášet medicínská data v standardizované podobě, což usnadňuje jejich výměnu a použití mezi různými zdravotnickými zařízeními a aplikacemi.

4.3 Implementace

Pro implementaci a experimentování s trénováním různých modelů je z důvodu vyšší výpočetní náročnosti využita platforma Kaggle, která umožňuje vytvářet a spouštět Jupyter notebooky a k trénování neuronových sítí používat GPU.[31]

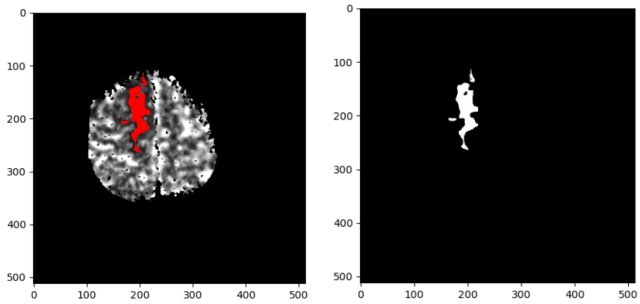
4.3.1 Načtení a předzpracování dat

Prvním krokem bude načtení dat do numpy tenzorů a jejich předzpracování.

Definujeme si funkce *read_and_transform_dicom_scans(path, img_size)* a *read_and_transform_png_masks(path, img_size)*, které na vstupu přebírají dva parametry: *path*, což je cesta k adresářům jednotlivých CT skenů a segmentačních masek a *img_size*, což je požadovaná velikost snímků. Funkce načítají jednotlivé snímky a masky jednoho skenu v konkrétním adresáři a vrací 3D tenzor s tvarem [počet řezů ve skenu, img_size, img_size], který se přidá do listu 3D tenzorů všech skenů.

Konkrétně funkce *read_and_transform_dicom_scans(path, img_size)* postupně načítá řezy z DICOM souborů pomocí knihovny pydicom, které následně přeškáluje na velikost *img_size* a převede na 2D numpy tenzor. Dále provede zvýšení kontrastu v řezech podle metody CLAHE (contrast limited adaptive histogram equalization), aby se zvýraznil rozdíl mezi oblastí zájmu a okolím.

Funkce *read_and_transform_png_masks(path, img_size)* načítá snímky z PNG souborů. Každý PNG snímek obsahuje konkrétní řez mozku (korespondující s konkrétním DICOM řezem) a barevně označenou oblast zájmu. Funkce ze snímku extrahuje pouze oblast zájmu (segmentační masku) a vytvoří nový binární snímek, kde jednotková hodnota pixelů znamená masku a nulová hodnota znamená pozadí. Na obrázek 4.4 můžeme vlevo vidět snímek s vyznačenou oblastí zájmu získanou softwarem *InfarctionCoreDelineator* a vpravo již vyextrahovanou binární masku, která bude použita pro učení neuronové sítě.



Obrázek 4.4: Vlevo snímek s vyznačenou oblastí zájmu, vpravo extrahovaná segmentační maska

Poté, co jsou všechny skeny a masky načtené v samostatných 3D tenzorech uložených ve dvou listech *scans* a *maskss*, předáme je do funkce *make_4D_tensors(scans, maskss)*, která tyto listy přetransformuje na dva plně předzpracované 4D tenzory tvaru [počet snímků v celé množině, img_size(výška), img_size(šířka), kanály].

Funkce dále provede normalizaci hodnot pixelů v tenzorech CT snímků. Tenzor korespondujících binárních masek není potřeba normalizovat (pixely obsahují jen 0 nebo 1).

CT snímky mají hodnoty uložené v jednotkách HU, které se řídí podle hustoty nasnímaných objektů v rozsahu od -1000 (a méně) do +2000 (a více), kde -1000 obvykle značí vzduch a +3000 kosti či jiné husté objekty.

Mozková tkáň (která nás primárně zajímá) má hodnoty okolo 20 až 50 HU. Protože funkce bude CT snímek normalizovat, vyfiltruje nejdříve pozadí obrázku (s hodnotami -1024 HU) a všechny extrémní hodnoty nezajímavé pro segmentaci a nastaví je na 0, protože jinak by jimi byly normalizované hodnoty silně zasaženy, což by negativní vliv na segmentaci mít mohlo. Potom se provede samotná normalizace hodnot.

Výstupem celého procesu načítání a předzpracování dat jsou tedy dva tenzory CT snímků a k nim odpovídajících segmentačních masek tvaru [počet snímků v celé množině, výška, šířka, kanály] = [1014, img_size, img_size, 1].

4.3.2 Augmentace dat

Dalším krokem je augmentace dat. Kvůli vysoké nevyváženosti v datech (dané charakterem problému), kdy velká většina snímků neobsahuje žádnou oblast zájmu neboli mají segmentační masku obsahující samé nuly, budeme augmentovat jenom snímky, které oblast zájmu obsahují.

Pokud bychom augmentovali všechny snímky, výsledná síť by se nejspíše

naučila v příliš vysoké míře predikovat nuly a nebyla by schopna správně identifikovat oblasti zájmu. Proto je lepší augmentovat pouze snímky, které obsahují oblasti zájmu, abychom zajistili, že se síť naučí nejobecnější a nejpotřebnější reprezentace pro správnou segmentaci obrazu.

Definujeme si tedy funkci *augment_positive_masked_slices(ct_tensors, mask_tensors, num_of_pos_pixels)*, která ve vstupních 4D tenzorech (získaných z předešlé části) vyhledá snímky, jejichž masky mají alespoň *num_of_pos_pixels* nenulových pixelů.

Poté tyto snímky s korespondujícími maskami augmentuje pomocí transformace rotace, translace a škálování a nově vzniklé snímky a masky přidá do vstupních tenzorů.

Výstup funkce tedy budou stejné tenzory, jako ty vstupní jen obohacené o augmentované pozitivní snímky.

Tímto jsme do určité míry narovnali nevyrovnanost v datech a neuronová síť se tak bude moci naučit potřebné reprezentace.

4.3.3 Ztrátová funkce a metriky

Pro sledování výkonu sítě použijeme dvě metriky: Diceho koeficient a Tverskyho koeficient.

Diceho koeficient je statistickým nástrojem, který se používá pro měření podobnosti dvou množin. V případě segmentace se používá pro porovnání segmentační masky učitele s maskou predikovanou sítí. Jedná se o jednu z nejpoužívanějších metrik pro segmentaci. Vzorec vypadá následovně:

$$D(X, Y) = \frac{2|X \cap Y|}{|X| + |Y|}$$

Kde X je segmentační maska učitele a Y segmentační maska predikovaná sítí.[32]

Tverskyho koeficient je generalizací Diceho koeficientu. Jeho vzorec vypadá takto:

$$T(X, Y) = \frac{|X \cap Y|}{|X \cap Y| + \alpha|X \backslash Y| + \beta|Y \backslash X|}$$

Zde α a β jsou váhy, které určují, jak moc se má brát v úvahu chybné označení pixelů. Platí $\alpha + \beta = 1$. [33]

Implementace obou metrik je přímočará a kopíruje uvedené vzorce.

```
# Diceho koeficient
def dice_coef(y_true, y_pred):
    y_true_f = K.flatten(y_true)
    y_pred_f = K.flatten(y_pred)
    intersection = K.sum(y_true_f * y_pred_f)
    return (2. * intersection + smooth) / (K.sum(y_true_f) +
   K.sum(y_pred_f) + smooth)

# Tverskyho koeficient
def tversky_coef(y_true, y_pred):
    y_true_pos = K.flatten(y_true)
    y_pred_pos = K.flatten(y_pred)
    intersection = K.sum(y_true_pos * y_pred_pos)
    diff1 = K.sum(y_true_pos * (1-y_pred_pos))
    diff2 = K.sum((1-y_true_pos)*y_pred_pos)
    alpha = 0.75
    beta = 1 - alpha
    return (intersection + smooth)/(intersection + alpha*
   diff1 + beta*diff2 + smooth)
```

Listing 4.1: Diceho a Tverskyho koeficient

Z obou koeficientů nyní dokážeme odvodit dvě ztrátové funkce, které vyzkoušíme při trénování sítě: Diceho ztrátu a Tverskyho ztrátu.

$$D_{loss} = 1 - D(X, Y)$$

$$T_{loss} = 1 - T(X, Y)$$

V rámci experimentů si definujeme ještě fokální variantu obou ztrátových funkcí.

$$D_{loss} = (1 - D(X, Y))^{\gamma}$$

$$T_{loss} = (1 - T(X, Y))^{\gamma}$$

V této variantě je do funkce přidán parametr $\gamma \in (0, 1)$, který zvyšuje hodnotu ztráty a nutí model naučit se lepší reprezentace. Hodnota parametru se volí ručně a je tudíž jedním z hyperparametrů sítě.

4.3.4 Definice a trénování modelu

Pro segmentaci byla v rámci experimentování s různými architekturami vybrána lehce modifikovaná architektura Unet (jejíž základní architektura byla popsána výše).

Modifikace spočívá v přidání reziduálních spojení realizující součet vrstev v rámci encodetu a decoderu, tím síť dokáže podchytit ještě více kontextu.

Implementace architektury je následující:

```
# Architektura
def Unet(input_shape):
    inputs = Input(shape=input_shape)

    # vstupni blok
    x = Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)

    #vystupy pro decoder
    #(residualni spoj mezi encoderem a decoderem)
    encoder_outputs = []
    #residualni spoj mezi vrstvami
    #v ramci samotneho encoderu a decoderu
    previous_block_activation = x

    # Encoder (kontrakcni cesta)
    ----------------------------------
    for filters in [64, 128]:
        for _ in range(2):
            x = SeparableConv2D(filters,
                                3,
                                padding="same")(x)
            x = BatchNormalization()(x)
            x = Activation("relu")(x)

        encoder_outputs.append(x)
        x = MaxPooling2D(3, strides=2, padding="same")(x)

        residual = Conv2D(filters, 1,
                          strides=2,
                          padding="same")
                          (previous_block_activation)

        # soucet residualnich spojeni
        x = add([x, residual])
        previous_block_activation = x

    # Bottleneck (prechodova cesta)
    ------------------------------------------
    x = MaxPooling2D(3, strides=2, padding="same")(x)
    for _ in range(2):
        x = SeparableConv2D(256, 3, padding="same")(x)
        x = BatchNormalization()(x)
        x = Activation("relu")(x)
    x = UpSampling2D(2)(x)
```

```
    # Decoder (expanzivni cesta)
    ----------------------------------
    for i, filters in enumerate([128, 64, 32]):
        for _ in range(2):
            x = Conv2DTranspose(filters, 3,
                                padding="same")(x)
            x = BatchNormalization()(x)
            x = Activation("relu")(x)

        # zvysovani rozmeru
        x = UpSampling2D(2)(x)

        residual = UpSampling2D(2)(previous_block_activation)
        residual = Conv2D(filters, 1, padding="same")
                          (residual)

        x = add([x, residual])
        previous_block_activation = x

        if i < len(encoder_outputs):
            enc_output = Conv2D(filters,
                                1,
                                padding="same")
                                (encoder_outputs[-i-1])
            # zretezeni residualnich spojeni
            x = Concatenate(axis=-1)([x, enc_output])

    # Vystupni vrstva
    outputs = Conv2D(1, 1, activation="sigmoid",
                     padding="same", name="outputs")(x)
```

Listing 4.2: Definice modelu

Ukázalo se, že dostačující počet epoch pro trénování je 35 s číslem dávky 32 a Tverskyho ztrátovou funkcí s hodnotou $\gamma = 0,75$.

Po zkompilování modelu si ještě vytvoříme callback *ModelCheckpoint*. Callback kód, který se spustí na různých místech během trénování modelu. To může být užitečné pro monitorování průběhu trénování, ukládání checkpointů modelu, přerušení trénování při dosažení určitého kritéria apod.

Callback *ModelCheckpoint* nám umožňuje monitorovat průběh trénování podle určitého kritéria a při dosažení kritéria uloží konkrétní parametry modelu, díky čemuž dokážeme zachytit nejlepší neučené reprezentace.

My v našem *ModelCheckpoint* callbacku jako kritérium zvolíme hodnotu ztrátové funkce na validačních datech, konkrétně zvolíme, že callback uloží parametry modelu, které v průběhu trénování zajistili nejnižší vali-

dační ztrátu.

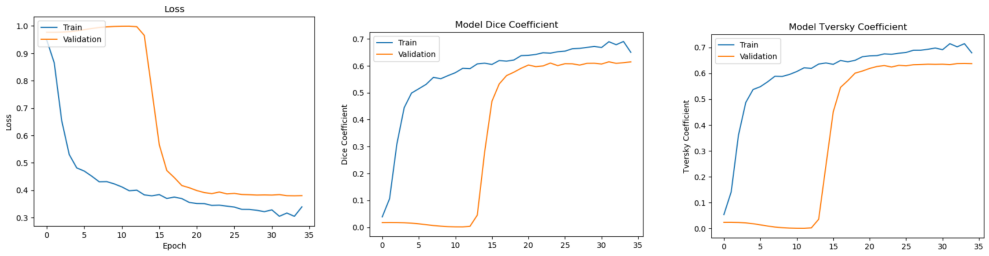
```
checkpoint = ModelCheckpoint('best_Unet_weights.hdf5' ,
                             monitor = 'val_loss',
                             verbose = 1,
                             save_best_only=True,
                             mode = 'min',
                             save_weights_only=True,
                             save_freq='epoch'
                            )
```

Listing 4.3: Callback ModelCheckpoint

Nakonec pomocí již známé funkce model.fit() začneme trénovat naší neuronovou síť.

4.3.5 Analýza a zhodnocení výsledků

Na následujícím obrázku můžeme vidět vizualizace Tverskyho ztráty a obou výše zmíněných koeficientů.



Obrázek 4.5: Vizualizace učení. Nalevo je průběh ztrátové funkce, uprostřed Diceho koeficient a napravo Tverskyho koeficient

Můžeme vidět, že na trénovací množinu (modré křivky) se začala síť přizpůsobovat poměrně rychle. Pro validační data se začal model přizpůsobovat zhruba v polovině trénování.

To může být způsobeno tím, že se model v počátečních epochách naučil nevhodné reprezentace charakteristické pouze pro trénovací data a v polovině trénování našel lepší, obecnější reprezentace společné všem datům.

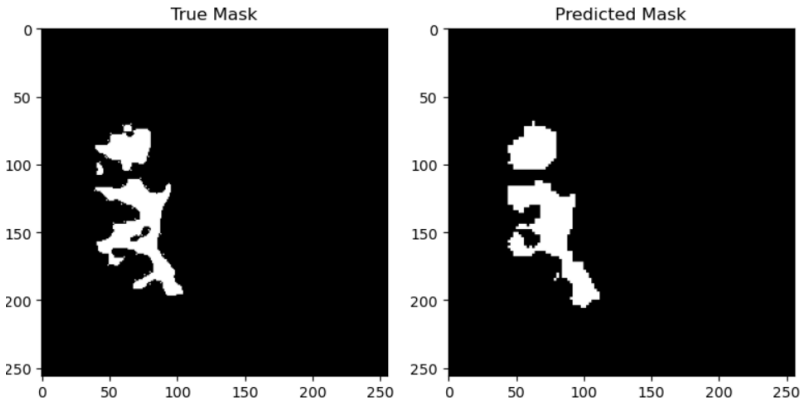
Nejlepší ztráta, které model dosáhl na validačních datech byla 0,37. Diceho a Tverskyho koeficient dosáhl na validačních datech hodnoty 0,61 a 0,64, což je nejlepší dosažený výsledek na těchto datech v rámci série experimentů, během kterých byly zkoušeny různé ztrátové funkce s různými parametry a různé neuronové segmentační architektury jako např. 3D Unet, klasická Unet, modifikovaná Unet s předtrénovanou VGG16 v encoderu atd.

Žádná ze zmiňovaných architektur nedosáhla hodnoty validační ztráty ani 0,6.

Důvodů, proč tomu tak je může být mnoho, od příliš komplexních architektur, po nevhodnost 3D architektury pro tato data (pro velmi řídký počet řezů s oblastí zájmu nemusí být použití 3D U-Net tou nejlepší volbou, protože nemusí mít dostatek informací, ze kterých by se naučila, a prostorový vztah mezi řezy nemusí být v tomto případě tak důležitý).

Celkově se ukázalo, že na těchto datech fungují pro lepší generalizaci segmentační architektury s menším množstvím parametrů (do 1 milionu).

Na obrázku 4.6 je srovnání segmentace snímku provedené natrénovaným modelem (napravo) s referenční segmentací provedenou učitelem (tedy softwarem *InfarctionCoreDelineator*).



Obrázek 4.6: Porovnání výsledku segmentace modelem. Nalevo je referenční segmentace a napravo predikce.

Vidíme, že predikce není úplně dokonalá, ale zhruba obsáhla požadovanou oblast.

Segmentace by se dala zlepšit například zvětšením datové množiny nebo velikosti snímků z 256x256, na kterých byl model natrénován (z důvodu limitace dostupných prostředků) na 512x512.

5 Závěr

V rámci této bakalářské práce byly popsány základní teoretické koncepty současného hlubokého učení s důrazem na důkladné vysvětlení umělých neuronových sítí. Následně byly představeny základní třídy a architektury, jako jsou konvoluční neuronové sítě (CNN), rekurentní neuronové sítě (RNN) a neuronové sítě typu transformer, spolu s popisem jejich využití v medicíně a demonstračními příklady.

V kontextu konvolučních neuronových sítí byla zdůrazněna jejich efektivita v oblasti zpracování obrazu a počítačového vidění a jejich uplatnění ve zpracování a analýze medicínských snímků.

Jako první praktický příklad byl implementován CNN klasifikátor, který dosáhl přesnosti 94 % při binární klasifikaci rentgenových snímků hrudníku na základě přítomnosti zápalu plic.

Dále byly představeny rekurentní neuronové sítě, zejména rekurentní architektura Long Short-Term Memory (LSTM), které jsou v medicíně využívány pro zpracování a analýzu sekvenčních dat, například EKG a EEG signálů.

Jako druhý praktický příklad byl natrénován LSTM klasifikátor, který dosáhl přesnosti 96 % při klasifikaci EKG signálů do 5 tříd podle různých srdečních stavů.

Poté byly představeny neuronové sítě typu transformer, vysvětlen princip jejich fungování a zdůrazněno, že jsou v současnosti nejlepší třídou neuronových sítí pro zpracování dlouhých textů. Bylo uvedeno, jak se mohou v medicíně využívat v různých směrech, včetně tvorby medicínských chatbotů či sumarizace rozsáhlých klinických dokumentů.

V hlavní praktické části, byl po nastínění problematiky segmentace obrazu implementován a natrénován model pro segmentaci dodaných CT skenů s referenční segmentací získanou použitím softwaru *InfarctionCoreDelineator*. Pro segmentaci byla po mnoha experimentech využitá lehce modifikovaná Unet architektura s Tverskyho ztrátovou funkcí. Výsledný model na těchto datech dosáhl hodnoty 0,61 Diceho koeficientu a 0,67 Tverskyho koeficientu, což byl nejlepší výsledek ze všech provedených experimentů. Výsledné predikované segmentace většinou dokáží obsáhnout oblast zájmu, ale už moc nedokáží kopírovat přesný tvar podle referenční oblasti.

Přesnost segmentace by se dala zvýšit například zvětšením datové množiny nebo zvýšením velikosti snímků z použitých 256x256 na 512x512.

A Příloha: Seznam zkratek

CT Computed tomography. 9, 31, 62, 67, 68, 75
GPU Graphics processing unit. 9, 38
TPU Tensor processing unit. 9
ReLU Rectified linear unit. 19, 20
PReLU Parametric rectified linear unit. 20
MSE Mean squared error. 21
CNN Convolutional neural networks. 28, 29, 30, 34, 36, 37, 38, 46, 75
MRI Magnetic resonance imaging. 31, 33, 66
DeepCCI Deep Chemical-Chemical interaction. 37
SMILES Simplified molecular input line entry specification. 37
CPU Central processing unit. 38
API Application programming interface. 38
RNN Recurrent neural networks. 46, 47, 48, 50, 51, 56, 75
vec2seq Vector-to-sequence. 47
seq2vec Sequence-to-vector. 47, 48
seq2seq Sequence-ti-sequence. 47, 48, 56
LSTM Long short-term memory. 48, 49, 50, 51, 54, 75
EEG Electroencephalograph. 50, 51, 75
EKG Electrocardiograph. 50, 51, 52, 75
EHR Electronic health records. 51
ViT Visual transformer. 61
ISBI International symposium on biomedical imaging. 64
DICOM Digital imaging and communications in medicine. 66, 67
CBF Cerebral blood flow. 66
CBV Cerebral blood volume. 66
PNG Portable network graphics. 67
CLAHE Contrast limited adaptive histogram equalization. 67

B Příloha: Elektronická příloha

Adresář **A20B0541P_prilohy** obsahuje 4 podadresáře.

1. Adresář **A20B0541P_prilohy/Text_prace** obsahuje PDF soubor s textem bakalářské práce, adresář s TeX zdrojovým dokumentem a adresář s obrázky použitými v textu.

2. Adresář **A20B0541P_prilohy/Aplikace_a_knihovny** obsahuje Jupyter notebooky s příklady popsanými v textu BP.

3. Adresář **A20B0541P_prilohy/Vstupni_data** obsahuje datové množiny použité v příkladech.

4. Adresář **A20B0541P_prilohy/vysledky** obsahuje výsledné parametry natrenovaných modelů v příkladech.

Každý z těchto adresářů obsahuje také Readme.txt soubor s detailnějšími informacemi.

Literatura

[1] KELLEHER, John D. *Deep learning*. Cambridge, MA: The MIT Press, 2019. ISBN 9780262537551.

[2] CHOLLET, Francois. *Deep learning with python*. New York: Manning publications, 2017. ISBN 9781617294433.

[3] *Unsupervised learning* [online]. Data science course, [cit. 2023/02/25]. Dostupné z:
`https://www.datasciencecourse.org/notes/unsupervised/`

[4] KUTYNIOK, Gitta. *An introduction to the mathematics of deep learning*. Notices of the American Mathematical Society, 2019, vol. 66, s. 776-784.

[5] RAO, Delip; MCMAHAN, Brian. *Natural language processing with PyTorch, build intelligent language applications using deep learning*. Sebastopol: O'Reilly Media, inc, 2019. ISBN 9781491978238

[6] KURENKOW, Andrey. *A brief history of neural nets and deep learning* [online]. Skznettoday, [cit. 2023/02/26]. Dostupné z:
`https://www.skynettoday.com/overviews/neural-net-history`

[7] *Backpropagation explained* [online]. Deeplizard, [cit. 2023/02/28]. Dostupné z:
`https://deeplizard.com/learn/video/Zr5viAZGndE`

[8] WOOD, Thomas. *Convolutional neural network* [online]. Deepai, [cit. 2023/02/28]. Dostupné z:
`https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network`

[9] SOJKA, Eduard; GAURA, Jan; KRUMNIKL, Michal. *Matematické základy digitálního zpracování obrazu* [online]. mi21.vsb, [cit. 2023/03/15]. Dostupné z:
`https://mi21.vsb.cz/sites/mi21.vsb.cz/files/unit/digitalni_zpracovani_obrazu.pdf`

[10] ŠIKUDOVÁ, Elena; ČERNEKOVÁ, Zuzana; BENEŠOVÁ, Wanda; HALADOVÁ, Zuzana; KUČEROVÁ, Júlia. *Počítačové videnie - detekcia a rozpoznávanie objektov*. Praha: Wikina, 2011. ISBN 978-80-87925-06-5

[11] QUOC, V. Le. *A Tutorial on Deep Learning Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks* [online]. cs.stanford.edu, [cit. 2023/03/20]. Dostupné z: https://cs.stanford.edu/~quocle/tutorial2.pdf

[12] ANAYA-ISAZA, Andres; MERA-JIMENEZ, Leonel; ZEQUERA-DIAZ, Martha. *An overview of deep learning in medical imaging.* Informatics in Medicine Unlocked, 2021, vol. 38.

[13] KWON, Sunyoung; YOON, Sungroh. *DeepCCI: End-to-end Deep Learning for Chemical-chemical interaction prediction* [online]. arxiv, [cit. 2023/03/16]. Dostupné z: https://arxiv.org/abs/1704.08432v1

[14] BRESSEM, K. Keno; Adams, C. Lisa; Erxleben, Christoph. et al. *Comparing different deep learning architectures for classification of chest radiographs* [online]. Sci Rep 10, [cit. 2023/03/25]. Dostupné z: https://doi.org/10.1038/s41598-020-70479-z

[15] HUANG, Gao; LIU, Zhuang; VAN DER MAATEN, Laurens; Weinberger, Q. Kilian. *Densely connected convolutional networks.* [online]. arxiv, [cit. 2023/03/25]. Dostupné z: https://arxiv.org/pdf/1608.06993.pdfhttps://arxiv.org/pdf/1608.06993.pdf

[16] ROHINI, G. *Everything you need to know about VGG16* [online]. medium, [cit. 2023/03/25]. Dostupné z: https://medium.com/@mygreatlearning/everything-you-need-to-know-about-vgg16-7315defb5

[17] YEHOSHUA, Roi. {Recurrent neural networks. [online]. cmu, [cit. 2023/03/26]. Dostupné z: http://www.cs.cmu.edu/~mgormley/courses/10601-s18/slides/lecture17-rnn.pdf

[18] HOCHREITER, Sepp; SCHMIDHUBER; Jurgen. *Long Short-Term memory.* Neural Computation 9, 1997, vol. 8.

[19] COLAH, Christopher. *Understanding LSTM networks.* [online]. colahsblock, [cit. 2023/03/26]. Dostupné z: https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[20] ABHYUDAY, N. Jagannatha; HONG Yu. *Bidirectional RNN for Medical Event Detection in Electronic Health Records*. IEEE International Conference on Bioinformatics and Biomedicine, 2018, s. 951-954.

[21] DEY, Victor. *When to Use One-Hot Encoding in Deep Learning?*. [online]. analyticsindiamag, [cit. 2023/04/10]. Dostupné z:
https://analyticsindiamag.com/when-to-use-one-hot-encoding-in-deep-learning/

[22] VASWANI, Ashish, et al. *Attention is All You Need*. Advances in Neural Information Processing Systems. 2017, s. 6000-6010. [cit. 2023/04/11].

[23] WOOD, Thomas. *Transformer neural network* [online]. Deepai, [cit. 2023/04/11]. Dostupné z:
https://deepai.org/machine-learning-glossary-and-terms/transformer-neural-networ

[24] ŽELEYNÝ, Miloš. *Zpracování digitalizovaného obrazu* [online]. kky, [cit. 2023/04/15]. Dostupné z:
http://www.kky.zcu.cz/cs/courses/zdo

[25] MINAEE, Shervin, et al. *Image Segmentation Using Deep Learning: A Survey* [online]. arXiv, [cit. 2023/04/15]. Dostupné z:
https://arxiv.org/pdf/2001.05566.pdf

[26] POTTER, Ryan. *Image Segmentation: The Deep Learning Approach* [online]. indiaai, [cit. 2023/04/15]. Dostupné z:
https://indiaai.gov.in/article/image-segmentation-the-deep-learning-approach

[27] RONNEBERGER, Olaf; FISCHER, Philipp; BROX, Thomas. *U-Net: Convolutional Networks for Biomedical Image Segmentation* [online]. arXiv, [cit. 2023/04/15]. Dostupné z:
https://arxiv.org/pdf/1505.04597.pdf

[28] MAULE, Petr. *Automated infarction core delineation* diplomová práce, Západočeská univerzita v Plzni - Fakulta aplikovaných věd 2012

[29] *3D Slicer image computing platform*. [online]. slicer. Dostupný z:
https://www.slicer.org/

[30] *DICOM*. [online]. dicomstandard. Dostupný z:

https://www.dicomstandard.org/

[31] *Notebooks documentation - Kaggle*. [online]. Kaggle. Dostupný z:
https://www.kaggle.com/docs/notebooks

[32] *Sørensen–Dice coefficient*. [online]. Wikipedia. Dostupný z:
https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient

[33] *Tversky index*. [online]. Wikipedia. Dostupný z:
https://en.wikipedia.org/wiki/Tversky_index