Česká zemědělská univerzita v Praze
Provozně ekonomická fakulta
Katedra informačního inženýrství

Diplomová práce

Rozpoznávání a klasifikace obrazu konvoluční neuronovou sítí

autor: Bc. Jakub Pekárek
vedoucí práce: doc. Ing. Arnošt Veselý, CSc.

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Jakub Pekárek

Systémové inženýrství a informatika
Informatika

Název práce

Rozpoznávání a klasifikace obrazu konvoluční neuronovou sítí

Název anglicky

Image recognition and classification with convolutional network

---

Cíle práce

Diplomová práce je zaměřena na problematiku hlubokého učení v jazyce Python, za použití frameworku Keras a knihovny TensorFlow.

Cílem práce je navrhnout a implementovat konvoluční síť vhodné architektury a její trénink na testovacích open-source obrazových datech a dále její porovnání s dalšími konvolučními sítěmi z hlediska efektivity a chyby.

Metodika

Metodika řešené problematiky je založena na studiu odborných zdrojů o problematice a vlastní řešení je realizováno navržením konvoluční neuronové sítě v jazyce Python, určené k rozpoznávání daných obrazových záznamů,která je trénována na veřejně dostupných datových tréninkových sadách v knihovně TensorFlow.

Doporučený rozsah práce

60 – 80 stran

Klíčová slova

Hluboké učení, Python, TensorFlow, Keras, CUDA, CNN, strojové učení, multimédia

---

Doporučené zdroje informací

Francois Chollet, Deep learning v jazyku Python, knihovny Keras, TensorFlow ISBN 978-80-247-3100-1

Předběžný termín obhajoby

2021/22 ZS – PEF

Vedoucí práce

doc. Ing. Arnošt Veselý, CSc.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 22. 11. 2021

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 24. 11. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 24. 11. 2021

Čestné prohlášení

Prohlašuji, že svou diplomovou práci „Rozpoznávání a klasifikace obrazu konvoluční neuronovou sítí“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 28. listopadu 2021 Bc. Jakub Pekárek

Poděkování

Děkuji vedoucímu doc. Ing. Arnoštovi Veselému, CSc. za cenné rady a připomínky při tvorbě této diplomové práce a své rodině za podporu při tvoření této práce.

Rozpoznávání a klasifikace obrazu konvoluční neuronovou sítí

Abstrakt

Tato diplomová práce v teoretické části prezentuje základní principy, fungování a možnou aplikaci konvolučních neuronových sítí na frameworku Keras a knihovně TensorFlow konfigurovaných pomocí programovacího jazyka Python. Dále charakterizuje jejich přednosti a omezení při plnění různých úkolů jako je rozeznávání znaků, obrázků a jiných vizuálních dat.

V praktické části je testování a využití konvolučních neuronových sítí s různým určením na obvyklých datových souborech a následně jejich použití na jiných datových souborech, pro které nebyly trénovány a naučeny.

V závěru pak popisuje srovnání jejich výsledků dosažených na datových souborech, které nejsou nativně obsaženy v knihovnách Keras a TensorFlow.

Klíčová slova: Hluboké učení, Python, TensorFlow, Keras, CUDA, CNN, strojové učení, multimédia, transfer-learning.

Image recognition and classification with convolutional network

Abstract

This diploma thesis describes basic principles, function and possible application convolutional neural networks on the Keras framework and Tensorflow library in programming language Python. Also characterises their strengths and weaknesses in fullfiling various tasks like character or image recognition and other visual data classification.

In the practical chapter is testing and usage convolutional neural networks with different purpose on usual datasets and subsequently usage on different datasets, on which they have not been trained and taught.

In the conclusion describes achieved results on datasets which are not contained in the Keras and TensorFlow libraries.

Keywords: Deep learning, Python, TensorFlow, Keras, CUDA, CNN, machine learning, multimedia, transfer-learning.

Obsah

Seznam obrázků

Seznam tabulek

1 Úvod

V této práci se zabývám problematikou rozpoznávání obrazu počítačem. V současné době je nejpopulárnější technikou, jak tuto věc s velkou mírou úspešnosti řešit, využití konvoluční neuronové síťě. Trénování takové sítě je problematické z několika důvodů: Je velice pravděpodobné, že opatření datového souboru v dostatečné kvalitě a velikosti, bude samo o sobě velmi složitým problémem, který je třeba překonat. A dalším je náročnost takového procesu na čas a zdroje podle výkonnosti hardware na kterém běží. Zpracování obrazového materiálu s rozlišením několika pixelů je otázkou sekund nebo spíše tisicín sekundy, ovšem v případě obrazu s rozlišením v HD tedy 1920x1080 pixelů nebo 4K, které je v rozměru 3840×2160 pixelů, může jít o násobně delší dobu hodin nebo dnů. Také je nutno brát na zřetel velikost, kterou bude zabírat soubor dat k trénování sítě. V případě opakování a ladění sítě v případě nevyhovujících výsledků se doba učení dále prodlužuje.

Proto se nabízí využití sítě natrénované na velkém objemu trénovacího materiálu, které jsou dostupné volně na Internetu a jejich úprava pro využití pro řešení obdobného nebo docela jiného problému.

Toto téma považuji za důležité z několika pro mě významných důvodů. V současné době je již možné, aby každý, kdo se chce věnovat řešení různorodých úloh pomocí hlubokého učení, se tomuto věnoval na svém osobním počítači. To vše díky rozvoji vykonného hardware a vývoji jednodušše použitelných knihoven jako je Keras. Obecně platí, že je mnohem méně časově náročné použít předtrénovanou síť a dosáhnout tak výsledku efektivněji, než zdlouhavým tréninkem nenaučené sítě.

V této práci budou předvedeny techniky využití již naučených sítí obsažených v knihovně Keras na dostupných datových souborech a bude prověřeno, do jaké míry jsou konvoluční sítě opakovaně využitelné a jaké výsledky to přináší.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem této práce je, pomocí nabytých teoretických znalostí, použít natrénované konvoluční neuronové sítě k řešení problému, ke kterému nebyly přímo trénovány s jiným modelem konvoluční neuronové sítě, který byl trénovaný od začátku a tyto výsledky, kterých bylo dosaženo na stejném datovém souboru, porovnat.

2.2 Metodika

Pro přípravu a úpravu dat je použit programovací jazyk Python v IDE PyCharm a jeho knihovny, zejména NumPy pro matematické úlohy, matplotlib pro zobrazení výsledků grafy.

Samotná konfigurace a následná kompilace modelů konvolučních neuronových sítí je provedena na platformě Keras s využitím funkcí knihovny pro strojové učení Tensorflow.

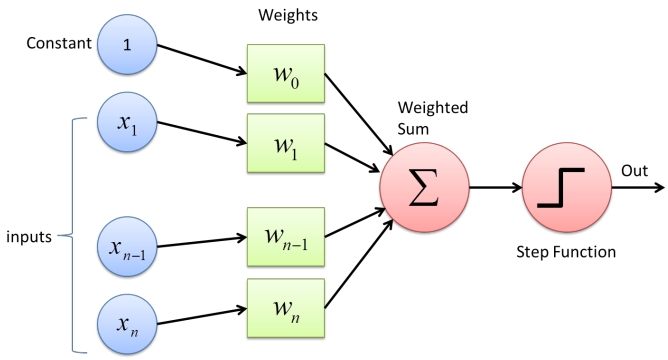
3 Teoretická východiska

3.1 Historie neuronových sítí a hlubokého učení

Myšlenky a ideje kolem strojového učení jdou desítky let do minulosti, ale až s rozvojem výpočetní techniky se tato technologie začíná široce rozvíjet.

V roce 1943 [12] dva vědci Warren McCulloch a Walter Pitts publikovali článek o tom jak by mohly neurony pracovat. K modelaci a znázornění použili součástky elektrického obvodu.[5]

Jeho funkci můžeme popsat takto: Skládá se ze čtyř částí[13]: Vstupní hodnoty X, váhy W Sumy všech vstupů vynásobených váhami $\sum$ Aktivační funkce upraví hodnotu získanou průchodem dat sítí, například v intervalu $< 0, 1 >$. [14]



Obrázek 1: Perceptron, Zdroj: https://deepai.org/machine-learning-glossary-and-terms/perceptron

V roce 1949 Donald Olding Hebb napsal práci The Organization of Behavior, která popsala metodu, jak se učí lidské mozky. Domníval se, že když jsou dva nervy stimulovány současně, jejich propojení se posílí. [6]. A v roce 1959 Bernard Widrow a Marcian Hoff ze Stanfordské univerzity vyvinuli model jménem "ADALINE", což je akronym pro Adaptive Linear Neuron, který vytvořili za použití jednoduchého elektrického obvodu s vlastní pamětí. [7]

Její následnice "MADALINE."[8] což je akronym pro "Multiple Adalines", tedy vznikla spojením obvodů ADALINE. Ta byla první neuronovou sítí použitou pro řešení problému z reálného světa, který se používá do dnešní doby. Fungovala jako adaptivní filtr datového

toku, byla schopna odhadnout další bit, který přijde a tím snižovala ozvěnu při telefonním hovoru. [8]

Bernard Widrow a Marcian Hoff dále pokračovali ve výzkumu a v roce 1962 [5] publikovali práci o učení perceptronu, které funguje podle pravidla: „Změna váhy závisí na hodnotě váhy před změnou, násobenou chybou, která je předtím dělena počtem vstupů“, což byl jeden z důležitých milníků ve vývoji strojového učení. Znamenalo to, že chyba v učení byla postupně snižována, až se nakonec srovnala. Toto pravidlo tvoří základ dnešního hlubokého a strojového učení obecně.[9]

První “konvoluční neuronová sít” na rozpoznávání obrazu jménem Neocognitron byla vytvořena v roce 1979 japonským vědcem Kunihiko Fukushima [15]. Ta dokázala rozpoznat geometrický obrazec bez ohledu na jeho umístění nebo nějakou jeho změnu. Zároveň byla schopna do jisté míry fungovat bez učitele.

V roce 1986 byl představen algoritmus back-propagation, což můžeme přeložit jako ”zpětné šíření chyby”. Jedná se o způsob, jak síť upravuje svoje parametry v průběhu učení [10]. Tento algoritmus vyřešil otázku využití Widrow-Hoffova pravidla s učení sitě s mnohonásobným opakováním cyklu.[11]

3.2 Současný stav neuronových sítí a hlubokého učení

Za rozšíření těchto technologií v poslední době vděčíme vědkyni [16] Fei-Fei Li, mimo jiné za vytvoření knihovny [17] ImageNet v roce 2009, která je tvořena velkým množstvím fotografií, přesněji zhruba 16 miliony obrázků pro realistické učení a vývoj neuronových sítí. [18]

A jako další z mnoha počinů z poslední doby, můžeme jmenovat AlphaGo od společnosti Google, která poprvé v roce 2016 [19] porazila člověka ve strategické hře GO. Byť se nejedná o prakticky využitelný úspěch, dává představu o tom, jak komplexní tento systém byl, že mohl porazit člověka.[19]

Celý tento vývoj provázela období bez výraznějšího pokroku, většinou z důvodu přehnaného očekávání a následného pesimismu. S tím, že jde o slepou vývojovou větev, nastaly v sedmdesátých a devadesátých letech dvě dlouhá období nezájmu.[10] S odstupem času je třeba konstatovat, že obavy byly liché a došlo k rozpoznání potenciálu i omezení neuronových sítí a strojového učení obecně a dnes zažívá velký rozmach a ukazuje se, že to vše je využitelné snad ve všech oborech lidského konání.

3.3 Budoucnost neuronových sítí a hlubokého učení

V budoucnosti nám neuronové sítě a strojové učení patrně přinesou plně autonomní řízení, vývoj nových léků, velké rozšíření v armádních aplikacích[20], chytrých domácnostech a ve spotřební elektronice, kde perfektně rozpoznají řeč uživatele a nebo poznají osobu, která se chce přihlásit do zařízení a budou přitom schopny odhadnout psychický stav, nebo například akutní zdravotní problém. Zvýší se využití těchto technologií u bezpečnostních složek v rámci jejich práce, rozpoznávání obličejů osob a jiných markantů[23] bude jen jedním z mnoha aspektů využití.[21] Diskuze o morálním dilematu využití neuronových sítí v kontextu omezování lidských práv nabydou na vážnosti a můžeme se jen domnívat, jaký bude jejich vliv na stav a využití těchto technologií v budoucnu.[22] O tom, jak lidský život ovlivní umělá inteligence, se můžeme jen dohadovat. Směr, kterým se ubírá, je ale již jasný a viditelný téměř pro každého: autonomně se pohybující vozidla, pokročilé robotické systémy, chytré domy nebo celá města řízená počítačem. Většina technologií je stále ve stádiu výzkumu a testování, je ale zřejmé, že rozmach těchto technologií bude velmi rychle pokračovat. Ve vojenském použití přináší tyto technologie již teď mnoho dilemat. V současné době je pravidlem, že poslední rozhodnutí vždycky učiní lidský operátor, zdali to tak zůstane, ukáže blízká budoucnost.

Stephen Hawking v rozhovoru, který poskytl BBC dne 2. prosince 2014, uvedl: "Vývoj plně soběstačné umělé inteligence může zapříčinit konec lidstva. Lidé, kteří jsou omezení pomalou biologickou evolucí, nemohou soutěžit a budou nahrazeni."[26]

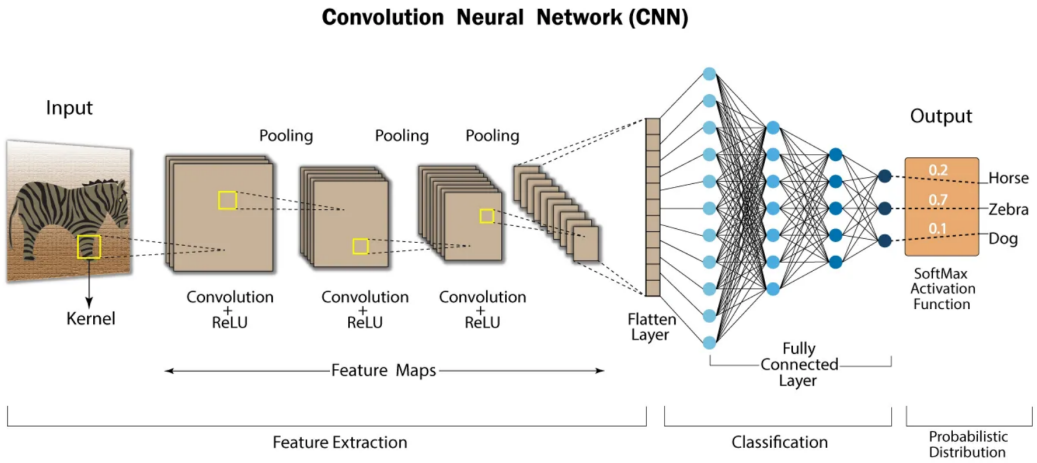
3.4 Základní části konvolučních neuronových sítí

Konvoluční neuronová síť je jednou z množiny tříd sítí hlubokého učení. Používá se převážně k počítačovému vidění, tedy k rozpoznávání a klasifikaci obrazových dat. Dále má uplatnění při rozpoznávání textu nebo řeči.

Velmi zjednoduššeně lze říci, že funguje na principu rozložení a následné transformaci vstupu, který dostává, dále opakovanému vyhledávání příznaků podle kterých určí, co by se na vstupním souboru, například obrázku zebry, jak je vidět na obrázku číslo 2. Jádrem a specifikem této sítě jsou konvoluční vrstvy. Obrazový materiál není pro konvoluční neuronovou síť nic jiného, než plocha pixelů s číselnými hodnotami. A v případě videa je to jen mnoho obrázků položených za sebou.

Proces konvoluce můžeme definovat jako aplikaci filtru na vstup, který je poté zopakován postupně na celou plochu vstupu a výsledkem je mapa příznaků, která zaznamenává

nejdříve přítomnost a poté umístění toho daného příznaku, který je charakteristický pro objekt, který je hledán, v tomto příkladu obrázku zebry stepní. Přičemž na konci celého procesu extrakce příznaků, jsou data transformována do skalární hodnoty. To provádí vrstva Flatten a plně propojené vrstvy síťě podle této hodnoty určí s hodnotou pravděpodobnosti, co se na daném obrázku nachází.



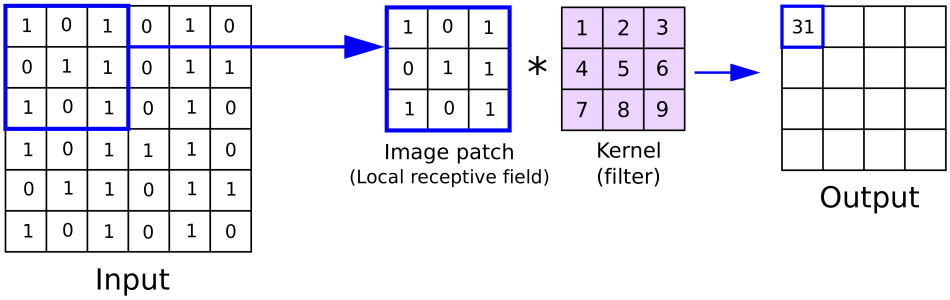
Obrázek 2: Diagram konvoluční sítě, Zdroj: https://i0.wp.com/developersbreach.com/wp-content/uploads/2020/08/cnn_banner.png?fit=1400%2C658&ssl=1

3.5 Vrstvy

Rozlišujeme několik základních typů vrstev, ze kterých se skládá celá konvoluční neuronová síť. Obecně se dá říci, že každá vrstva, je modulem, který nějakým způsobem zpracovává data na vstupu a poté poskytne nějaký výstup.[1]s.65

3.5.1 Konvoluční vrstva

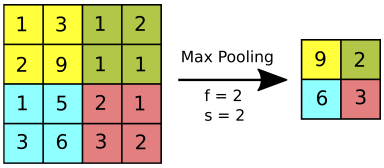
Funkcí této vrstvy je použití filtru, který z plochy obrazu extrahuje takzvané příznaky, tedy specifické znaky, které na obrázku jsou. Tento filtr, někdy také nazývaný Kernel, je menší než plocha kterou prochází, proto dochází k překrývání vstupu. Toto umožnuje filtru, tedy souboru vah, upravit váhy opakovaným násobením na jiných částech vstupu.



Obrázek 3: Diagram konvoluční vrstvy, Zdroj: https://anhreynolds.com/blogs/cnn.html

3.5.2 Pooling vrstva

Tato vrstva redukuje velikost vstupu a zachovává důležité příznaky pro další proces rozhodování. Z většího vstupu extrahuje příznaky, které jsou relevantní k rozpoznání a ty nedůležité potlačuje. Znázorněno na obrázku č.4.



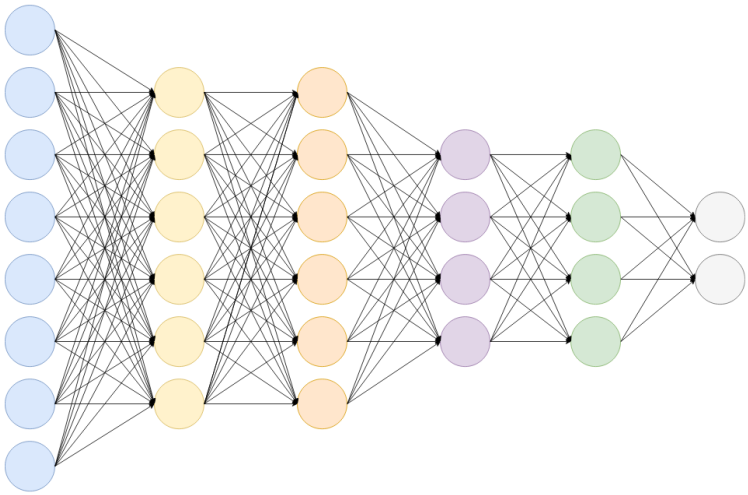
Obrázek 4: Diagram pooling vrstvy, Zdroj: https://anhreynolds.com/blogs/cnn.html

3.5.3 Plně propojená vrstva

Tato vrstva skládá získané příznaky dohromady a předkládá výstup pro rozhodování nebo kategorizaci. Zpracovává výstup předchozích vrstev, který nakonec vyhodnotí a určí, o jakou věc se jedná a pravděpodobnost s jakou se o daný předmět jedná. Schéma plně propojené vrstvy na obrázku č.5.

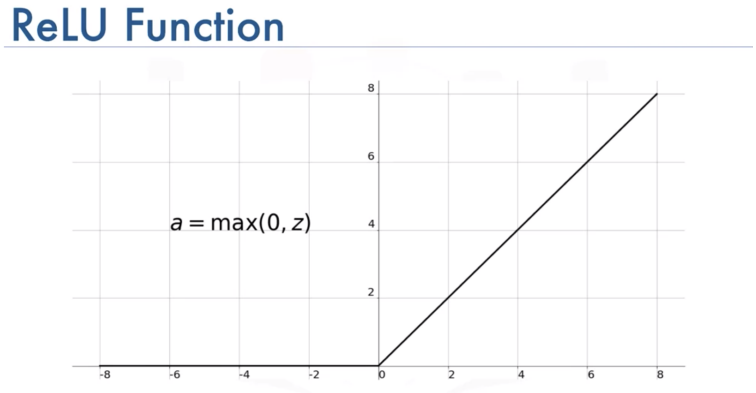
3.6 Aktivační funkce

Další důležitou částí jsou aktivační funkce, jsou součástí "neuronu"konvoluční sítě, a jejich funkce je, jak z názvu vyplývá, aktivovat nebo neaktivovat neuron. To činí na základě



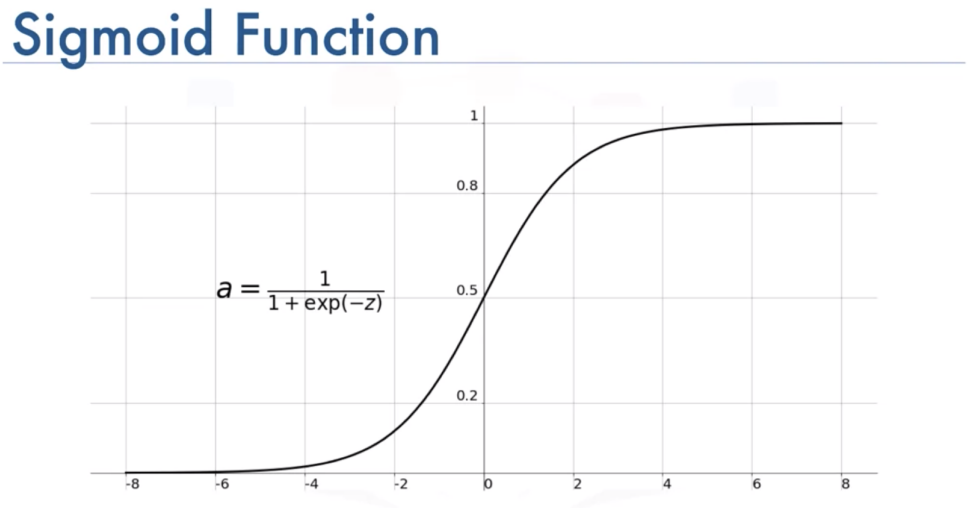
Obrázek 5: Diagram plně propojené vrstvy, Zdroj: https://towardsdatascience.com/convolutional-neural-network-17fb77e76c05

vstupu a s ním souvisejícího výpočtu. Tedy jejich hlavní funkcí je rozhodování a jejich další důležitou funkcí je transformace získané hodnoty na hodnotu v intervalu, dané průběhem funkce. Aktivačních funkcí je celá řada, často používané jsou, i v kombinaci, ReLU neboli Rectified Linear Activation Function a Sigmoid, každá pracuje hodnotami jinou metodou. ReLU, jak je z průběhu křivky patrné, generuje na výstupu kladné hodnoty. [1]s.75



Obrázek 6: Znázornění funkce ReLU, Zdroj: https://miro.medium.com/max/2400/1 × LiBZo_FcnKWqoU7M3GRKbA.png

Funkce Sigmoid oproti ReLU, poskytuje hodnoty v intervalu (0,1), které lze interpretovat jako pravděpodobnost, čím více se blíží k 1, tím je jistota vyšší a naopak.[1]s.75



Obrázek 7: Znázornění funkce Sigmoid, Zdroj: https://miro.medium.com/max/2400/1*a04iKNbchayCAJ7-0QlesA.png

3.7 Optimalizátory a ztrátové funkce

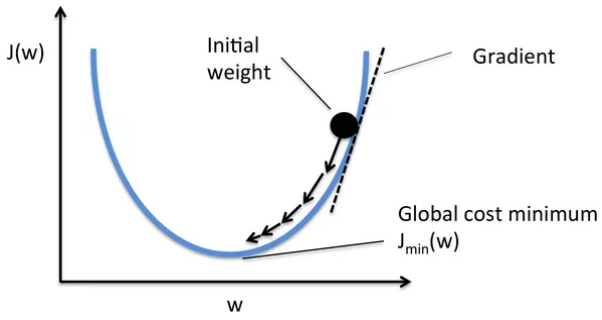
Optimalizátor zajišťuje to, jak bude model v průběhu tréninku změněn, jak rychle se bude učit, která konkrétní varianta stochastického gradientního sestupu bude použita. Základním problémem je vhodné nastavení kroku, tedy vzdálenosti, o kterou se po každém průchodu změní hodnota vah. Nesmí být příliš velký ani příliš malý, to by se síť zastavila v lokálním minimu a nepokračovala v tréninku, v případě velkého kroku zase může dojít k přeskočení hledaného minima [1]s.60.

Ztrátová funkce [1]s.58 vyhodnocuje úspěšnost modelu a kvalitu řešení, mezi hojně používané patří binární křížová entropie, kategorická křížová entropie apod. Tato funkce měří rozdíl mezi reálnými hodnotami a těmi, které síť předpověděla. Zde je třeba říci, že funguje na principu sestupu gradientu, který lze popsat jako hledání minima diferencovatelné funkce [1]s.58, před trénováním sítě jsou její hodnoty vah nastaveny na náhodnou hodnotu a při učebním procesu se hledá minimum funkce. Obě tyto části poté upravují průběh stochastického gradientního sestupu.

Algoritmus gradientního sestupu lze dle [1]s.59 popsat takto:

1. Máme dva soubory dat, trénovací X a cílové Y.

2. Na tréninkových datech spustíme síť a výsledkem bude predikce Y.
3. spočítáme rozdíl mezi Y a predikcí Y.
4. Vypočteme gradient ztráty podle parametrů sítě.
5. Posuneme parametry po křivce směrem k menší hodnotě, čímž se následně sníží ztráta. Obrázek č.8

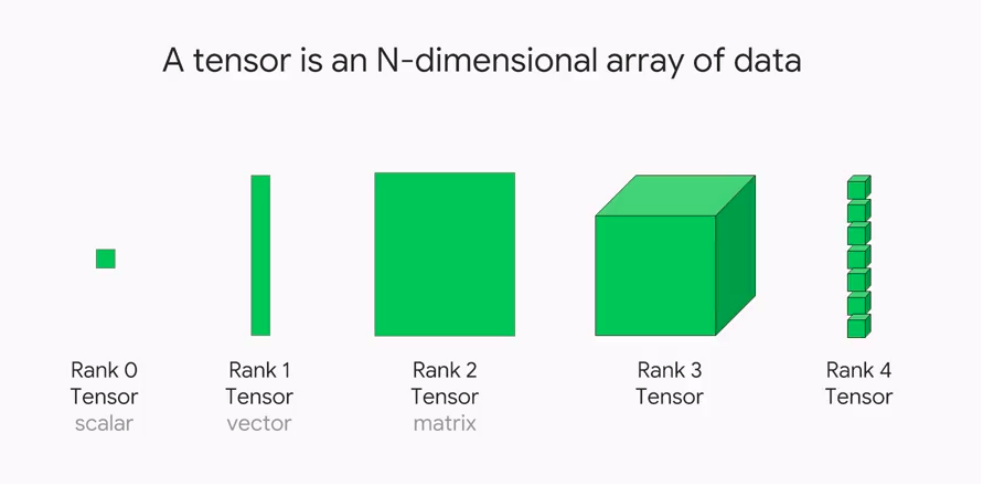


Obrázek 8: Sestup gradientu, Zdroj: https://miro.medium.com/max/602/1 × t6OiVIMKw3SBj Nzjlp_Fw.png

3.8 Data a jejich úprava, Tensory

Konvoluční neuronové sítě mohou pracovat s různorodými daty, lze využít neupravená data, fotografie nebo text a nebo vytvořit soubor hodnot například v tabulkovém formátu .csv. K tomu je možné využít vestavěné funkce v Kerasu, které byly vytvořeny k přípravě a úpravě tréninkových dat. Hojně využívané jsou load_data, extract_features, flow_from_directory, flow_from_dataframe. Jejich použití bude popsáno v praktické části. Tensory vytvořené těmito funkcemi pak označujeme jejich dimenzí[1]s.43, nebo osou[26]. Tensor nulté dimenze (OD) je skalární hodnota, tensor vektoru má dimenzi (1D), obrázek označujeme dimenzí (3D) má tři rozměry: výšku, šířku a hodnotu každého bodu, pokud obrázky označíme a dáme je za sebe, jedná se o vícedimenzionální matici, tensor o dimenzi (4D). Tímto způsobem lze slova, obrazy, video, transformovat na datový soubor pro neuronovou síť. [1]s.48

Rozměr tensoru se také někdy označuje jako stupeň (rank), jak je vidět na obrázku č.9.[26]



Obrázek 9: Tensor a jeho stupně, Zdroj: https://miro.medium.com/max/891/0*jGB1CGQ9HdeUwlgB

3.9 Hardware a Software

Následující část popisuje nutné kroky k instalaci a provozování Kerasu na lokálním počítačovém systému s vlastní GPU.

3.9.1 Software, konfigurace OS Ubuntu, Keras, Tensorflow

Při výběru operačního systému, na který bude Keras a další knihovny nainstalován, bylo zohledněno důrazné doporučení v části Příloha A [1]s.310, kde pro bezproblémový chod Kerasu je doporučována distribuce GNU/Linux Ubuntu. A to i pro uživatele operačního systému MS Windows, kterým je doporučena instalace do vedlejšího diskového oddílu tzv. dual-boot. Instalace a provoz linuxového OS Ubuntu verze 18.04 TLS Bionic Beaver nebo Ubuntu 20.04.2. LTS Focal Fossa jsou bezproblémové a proto byla tato doporučení vzata na vědomí. Za celou dobu provozu a testů nedošlo k jediné chybě nebo špatné funkcionalitě z hlediska operačního systému. Použití Ubuntu jako operačního systému pro hluboké učení a konkrétně pro trénování neuronových konvolučních sítí se jeví jako velmi vhodná volba. Nutno samozřejmě uvést, že i jiný klasický GNU/Linux operační systém bude fungovat. Veškerý kód byl spouštěn v IDE Pycharm, nicméně vzhledem k tomu, že většina manuálových stránek poskytuje kód do Notebook Jupyter, je občas nutné kód upravit, jedná se však o veskrze kosmetické změny. Kód v Jupyter Notebooku je narozdíl od IDE Pycharm, spouštěn po segmentech a lze ho lépe modifikovat. Výběr prostředí je spíše otázkou osobních preferencí a rozsahu úkolů, které chceme řešit. Dále je nutné uvážit, to

že grafické prostředí Linuxu, v tomto případě X server, zabírá část paměti grafické karty, jedná se o zhruba 300 MB. To není mnoho, ale v situacích, kdy je datový soubor již stejně velký jako paměť grafické karty může dojít ke zpomalení celého výpočtu. Nutno ale dodat, že znatelný rozdíl by nastal při dlouhodobém trénování, v délce například dnů. Alternativou by se jevilo použití počítače bez grafického rozhraní se vzdáleným připojením přes SSH nebo webový server Jupyter Notebooku. Další možností je využití cloudu pomocí Jupyter Notebooku, tyto jsou ovšem z valné většiny placené. Ceny se pohybují od 0.5Kč za hodinu při nízkém výkonu po 200Kč za hodinu v případě Colab od společnosti Google. [51] Pro malé datové soubory to ale není nutné, lze vystačit s osobním počítačem.

3.9.2 Python

Celá práce bude psána v interpretovaném jazyce Python verze 3.8, z důvodu jeho jednoduché syntaxe, velmi dobré přizpůsobitelnosti a efektivity pro tvorbu a učení neuronových sítí ve frameworku Keras. Dále bude pro větší komfort využito IDE PyCharm 2020.3 Community Edition. Pro práci bude využito virtuálních prostředí v jazyce Python, ta poskytují výhodu při práci s moduly pro hluboké učení, hlavně při problémech s kompatibilitou jednotlivých verzí Kerasu, Tensorflow a dalších. Vytvoření tohoto virtuálního prostředí je pak možné těmito příkazy:

V případě použití package manageru pro Python Pip3:

`python3 -m venv deeplearning`[43]

po aktivaci prostředí příkazem:

`source deeplearing-env/bin/activate`

`(deeplearning) $ python -m pip install keras`[43]

nebo takto v případě využití alternativního správce softwarových balíků Anaconda:

`conda create -n deeplearning python=3.8 anaconda`[44]

`conda install -n deeplearning [package]`[44]

Takto například můžeme nastavit virtuální prostředí pro využití Tensorflow a druhý pro Tensorflow-gpu pro porovnání výkonu při tréninku CPU mezi GPU. Práce v textovém terminálu by byla vhodná při práci na specializované stanici bez grafického prostředí pro využití veškerého dostupného výkonu. Mnohem rychlejší je vytvoření virtuálních prostředí přímo v IDE Pycharm, využití této funkcionality má výhodu v možnosti spravování prostředí přes možnosti nastavení IDE. Při řešení kompatibility softwarových balíků v rámci této práce, se neobjevil žádný výrazný problém s kompatibilitou. Jedná se o občasně se vyskytující problém, kdy jednotlivé moduly Pythonu závisí na jiných modulech různých verzí, přičemž nejde vyhovět kombinaci, kterou požadují.

Dá se říci, že při trénování sítě se objevilo pouze varování ohledně funkce:
`Model.fit_generator`

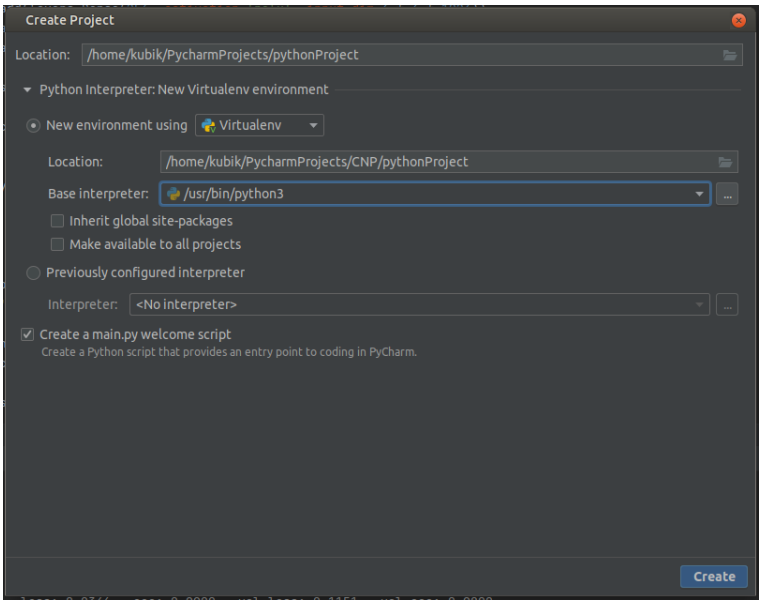
```
WARNING:tensorflow:From /home/peky/PycharmProjects/CNN/
Densenet121_zmraz_gener_finetun.py:109: Model.fit_generator
(from tensorflow.python.keras.engine.training) is deprecated
and will be removed in a future version. Instructions for updating:
Please use Model.fit, which supports generators.
```

V tomto konkrétním případě lze starší verzi metody použitou pro trénink modelu podle [1] `model.fit_generator` použít, ve verzi Kerasu 2.4.3 není třeba řešit kompatibilitu. Vše funguje i se starší funkcí. Jedná se však o potřebnou funkcionalitu, jelikož se různé verze nedají často nainstalovat na stejný operační systém a dá se takto vyhnout testování kompatibility neustálým instalovaním a odebíraním těchto knihoven.

Proto byla nahrazena novější verzí funkce `model.fit` pro případné použití modelů v této práci v budoucích verzích knihovny Keras.

Vzhledem k tomu, že pro práci bylo pro větší komfort využito prostředí Pycharm Community, je pak tvorba virtuálního prostředí znázorněna na obr. č.10:

Přičemž instalace a upgrade knihoven poté probíhá, jak je vidět na obr. č.11:



Obrázek 10: Vytvoření virtuálního prostředí v Pycharm, Zdroj: Vlastní

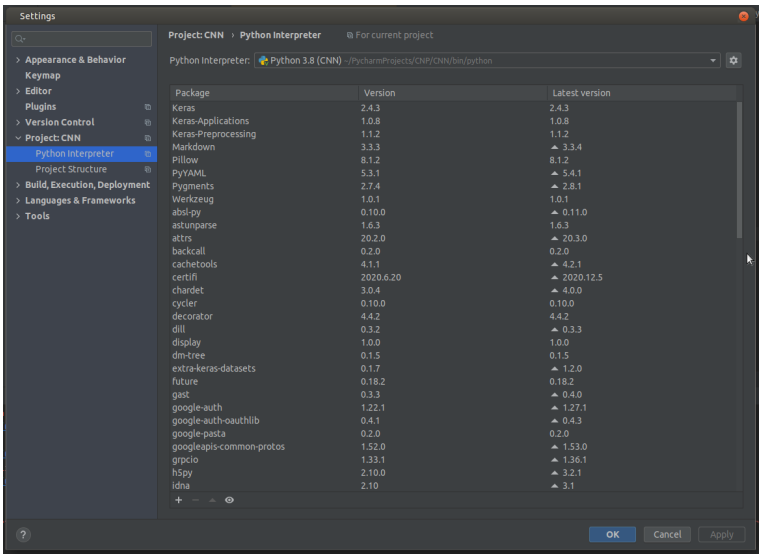
3.9.3 Keras a TensorFlow

Keras je API neboli aplikační programovací rozhraní pro hluboké učení, je napsáno v jazyce Python, klade důraz na přístupnost a relativní jednoduchost.[27], Důležitou vlastností je vysoký stupeň abstrakce pro jednoduché programování, přístupné i lidem bez větších zkušeností. Dále je modulární, podporuje různé knihovny pro hluboké učení, mezi nejpoužívanější patří TensorFlow od společnosti Google. Ta narozdíl od Kerasu poskytuje i nízkoúrovňové API pro průmyslové použití. Druhou vestavěnou knihovnou je Theano, vyvinuté akademiky z univerzit po celém světě[28]. Třetí vestavěnou knihovnou je The Microsoft Cognitive Toolkit (CNTK), pod licencí open-source, pro distribuované hluboké učení[29]. Schematicky zobrazeno na obrázku č.12.

Tensorflow je naopak určen pro velké datové soubory a obvykle se používá pro velmi výkonné modely na profesionální úrovni. Pro jeho použití je třeba zkušenost s hlubokým učením a dobré znalosti matematických operací.[33]

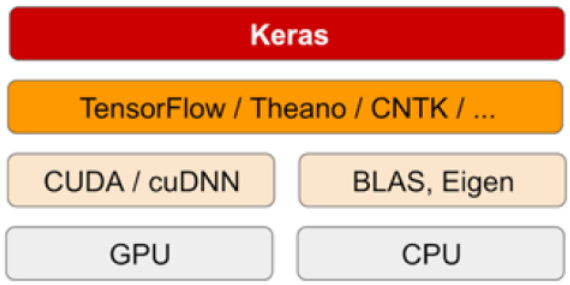
Dá se říci, že Keras zpřístupňuje funkce knihoven Tensorflow nebo Theano, přičemž uživateli ulehčuje vývoj a experimentování.

Trénování konvolučních neuronových sítí v Kerasu má několik praktických výhod, obsahuje vzorové datové soubory, které jsou velmi vhodné k seznámení se s tréninkem konvolučních neuronových sítí. Základní soubor ručně psaných číslic MNIST, pro klasifikaci číslic. Fashion MNIST, který obsahuje obrázky 10 různých druhů oblečení. Soubor Cifar100



Obrázek 11: Instalace knihoven Pycharm, Zdroj: Vlastní

zase obsahuje obrázky automobilů, zvířat a dalších předmětů, zařazených ve 100 kategoriích. Je tedy obsaženo vše potřebné pro seznámení s funkcemi a následnému trénování neuronových sítí. Tyto datové soubory jsou již součástí nainstalovaných knihoven. A jako nejdůležitější funkcionalitou je možnost využití předtrénovaných modelů konvolučních sítí, které jsou jednodušše přístupné a využitelné pro výzkum. Dále má velmi kvalitně napsanou dokumentaci dostupnou na webové adrese `https://keras.io/guides/`



Obrázek 12: Keras Diagram, Zdroj: https://miro.medium.com/max/504/1zumzj_ UJzenHYx0Gyyulyw.png

V této práci se zaměřím na využití knihovny TensorFlow v Kerasu, na GPU, na platformě CUDA s využitím knihovny cuDNN od společnosti NVIDIA.[1]s.311

3.9.4 Hardware, CUDA, GPU

Obecně lze říci, že pro trénink rozsáhlejších konvolučních sítí je třeba velmi výkonný hardware, roli výpočetního prostředku zastává grafická karta (GPU), ta je již ze své podstaty uzpůsobená pro vykonávání velmi velkého množství paralelních výpočtů, samozřejmě za předpokladu použití software, který tuto možnost dovede využít. Počítačové procesory tyto úlohy zastanou také, ale v čase neporovnatelně delším. Jak vyplývá z článku TensorFlow 2 - CPU vs GPU Performance Comparison[31] užití grafické karty pro trénování je zhruba 6x rychlejší než využití procesoru. Platí zde pravidlo, že čím více vrstev, parametrů, síť má, tím větší rozdíl v čase potřebném pro výpočet bude. Proto nutné využít knihovnu tensorflow-gpu, která užití grafické karty podporuje.

Využití výkonu grafické karty pro výpočty je možné díky technologii CUDA. Jedná se softwarovou platformu, která mimo jiné umožňuje provádět paralelní výpočty přímo, kódem napsaným v jazycích C nebo C++ a nebo prostřednictvím API.[30]

Pro učení a testování již naučených sítí na nových datech neboli "transfer learning"[32] byla použita tato sestava:

- Procesor Intel(R) Core(TM) i5-9400F CPU @ 2.90GHz
- Operační pamět RAM 32819MB DDR4
- Grafická karta GeForce GTX 1050 Ti 4GB RAM GDDR5
- Základní deska GIGABYTE Z390 GAMING X

Softwarová výbava počítače, tedy operační systém a verze ovladačů grafické karty a platformy pro paralelní výpočty CUDA byla tato.

- Operační systém Ubuntu 20.04.2 LTS
- Driver Version: 450.102.04
- CUDA Version: 11.0
- libcudnn7 verze 7.6.5.3

https://www.tensorflow.org/install/gpu
https://www.tutorialspoint.com/keras/keras_installation.htm
https://linoxide.com/ubuntu-how-to/install-cuda-ubuntu

Instalace při dodržení postupů na výše uvedených manuálových stránkách byla bezproblémová s jedinou výjimkou a to funkce výpočtů na GPU. V tomto případě nefungovala verze nvidia-driver-418. Tento problém se podařilo vyřešit instalací novější verze ovladače grafické karty Nvidia. Úspěšně byla otestována kombinace Nvidia driver 450.102.04, CUDA Version: 11.0 a knihovna libcudnn7 verze 7.6.5.3. V této kombinaci funguje již grafická akcelerace bezchybně bez dodatečného konfigurování nebo testování různých verzí. V novější verzi Ubuntu 20.04.2 LTS, již vše fungovalo bez nutnosti dodatečné konfigurace a testování kompatibility různých verzí. Grafická akcelerace fungovala bezchybně se všemi verzemi ovladače grafické karty Nvidia.

Příkazem v terminálu `watch -n 2 nvidia-smi` pak získáme tento výstup (obr. č. 13) čímž ověříme, že akcelerace skutečně funguje. Tento výpis ukazuje nainstalované a využívané ovladače. Dále verzi CUDA a v části s názvem CPU-Util vytížení grafické karty při trénování.

```
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 450.102.04   Driver Version: 450.102.04   CUDA Version: 11.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  GeForce GTX 105...  Off  | 00000000:01:00.0  On |                  N/A |
|  0%   52C    P0    N/A /  90W |   3925MiB /  4036MiB |     96%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1169      G   /usr/lib/xorg/Xorg                184MiB |
|    0   N/A  N/A      7956      G   /usr/lib/firefox/firefox            1MiB |
|    0   N/A  N/A     14212      C   ...ojects/CNP/CNN/bin/python     3735MiB |
+-----------------------------------------------------------------------------+
```

Obrázek 13: Monitoring zatížení GPU, Zdroj: vlastní

4 Vlastní práce

V této části budou prezentovány techniky pro snížení přeučení, tak jak je lze použít při konfiguraci neuronové konvoluční sítě v Kerasu. Poté bude uveden příklad skriptu, kterým lze připravit datový soubor k tréninku konvoluční neuronové sítě. A následně popis vytvoření jednoduchého referenčního modelu neuronové konvoluční sítě a jeho trénink na připraveném datovém souboru od nuly. Dále popis konfigurace již naučených sítí a testování jejich výkonu na malém tréninkovém datovém souboru. V závěru bude následovat vyhodnocení použitých technik a diskuse výsledků.

4.1 Techniky pro snížení přeučení

Tématem praktické části je použití předtrénované neuronové konvoluční sítě neboli „transfer learning“, jehož principem je použití sítě, která byla dříve natrénována na velké množině dat, v tomto případě na tisících nebo milionech obrázků různých kategorií. Konkrétně se jedná o trénink na souboru Imagenet, který obsahuje kategorie zvířat, plísní, automobilů, nářadí a celou řadu dalších [17]. V souhrnu se jedná o zhruba 14 000 000 obrázků[17]. V reálném světě je velmi obtížné získat takové množství materiálu pro trénink navržené sítě, tedy pokud ho již někdo jiný nevytvořil a nezveřejnil, takové případy ale nejsou moc časté. Záleží tedy na povaze problému, v případě rozpoznávání obecných objektů jako jsou automobily, lidské postavy, zvířata, je možno použít mnoho volně přístupných souborů dat. Jiná situace pravděpodobně nastane v případě rozpoznávání neobvyklých nebo specifických objektů, kdy žádný soubor dat neexistuje nebo není dostupný, jako například cirkusových klaunů nebo obrázků s typy balistických střel.

Pro vývoj a trénink konvolučních neuronových sítí jsou datové soubory nutné, a pokud možno, s co největším počtem exemplářů, které je třeba rozpoznávat nebo klasifikovat. V případě menšího počtu jsme znevýhodněni, ale existují způsoby, jak tuto nevýhodu do jisté míry eliminovat.

Dalším výhodným aspektem použití předtrénované sítě je úspora času, neboť síti trvá zlomek času, oproti učení od začátku, zpracovat nová data. Zároveň je tato technika využitelná, pokud není k dispozici výkonný hardware. K tomuto provozu pak obecně stačí procesor nebo grafická karta s menším výkonem. V době psaní této práce je na trhu nedostatek výkonných karet pro hluboké učení.

V případě tréninku na malém souboru dat velmi rychle dojde k jevu nazývanému přeučení, neboli "overfitting". V principu se jedná o stav kdy síť ukládá vlastnosti, vzory

specifické pro ta konkrétní data, na kterých je cvičena[1]s.105. Síť si velice rychle začne pamatovat konkrétní detaily. To způsobí propad ve výkonu v případě jejího vystavení jiným datům obsahujícím obdobné objekty, které ještě neviděla. Ve výsledcích bude tento jev znázorněn zvýšením hodnoty validační ztráty.

Technik pro snížení přeučení na přijatelnou úroveň, je několik. Ty nejčastěji používané jsou podle[1]s.106 tyto:

- Přidání výpadku, vrstva Dropout
- Redukce velikostí sítě.
- Použití ImageDataGenerator
- Přidání váhové regularizace.

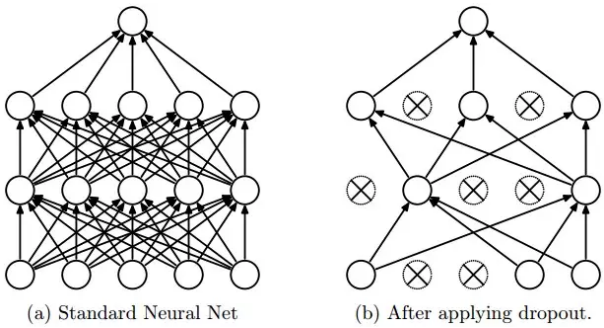
4.1.1 Přidání vrstvy výpadku

Vrstva výpadku (Dropout layer) funguje na principu náhodného vynulování části výstupních příznaků vrstvy při tréninku sítě, ty pak mají zmenšený vliv na rychlost učení a ztrátovou funkci[1]s.110.

Implementace pak vypadá následovně:

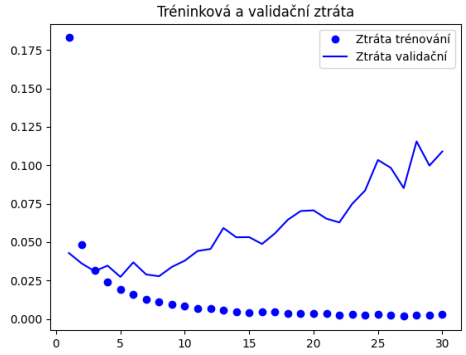
```
model.add(layers.Dropout(0.25))
```

Argument rozsahu nulování se zpravidla pohybuje mezi 0.2 až 0.5[1]. Vkládá se mezi výstup jedné a vstup následující vrstvy.

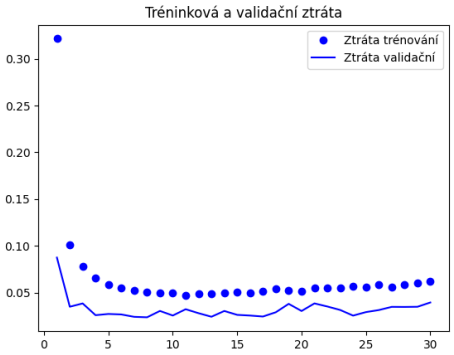


Obrázek 14: Vrstva Dropout, Zdroj: https://blog.christianperone.com/wp-content/uploads/ 62015/08/dropout.jpeg

Rozdíl na ztrátě s použitím vrstvy výpadku, na obrázku 15 je vidět, že se validační ztráta kolem 7. epochy začíná zvyšovat a v průběhu trénování dosahuje hodnota až 0.1, přičemž v případě použití dvou vrstev Dropout se hodnota ustálí na zhruba 0.05. Průběh hodnot v tréninkových epochách lze vidět na hodnotách grafů na obrázcích 15 a 16.



Obrázek 15: Ztráta na souboru mnist bez využití vrstvy výpadku, Zdroj: Vlastní



Obrázek 16: Ztráta na souboru mnist s vrstvou výpadku, Zdroj: Vlastní

4.1.2 Redukce velikosti sítě

Redukce velikosti sítě je další možností, jak vylepšit kvalitu trénované sítě, protože jednou z nejdůležitějších vlastností sítě je schopnost uchovávat informace. Extrahované záznamy o materiálu, který byl použit pro naučení, musí obsahovat dostatek znaků potřebných k

úspěšné identifikaci nového obdobného neznámého materiálu a zároveň nesmí přesáhnout hranici, kdy by se kapacita využila k přílišně podrobnému a neobecnému učení markantů. V tomto případě, pak dojde ke snížení výkonu celé sítě na testovacích datech. Dále dle [1]s.106 platí, že čím větší kapacita sítě, tím dříve dojde k přeučení, tedy síť se přizpůsobí na míru trénovacím datům a sníží svou schopnost zobecňovat a tedy odhadovat správně data testovací nebo jiná neznámá data.

Zároveň je třeba dbát na dostatečnou kapacitu, tak aby síť měla možnost zaznamenat dostatek markantů z tréninkového souboru dat [1]s.106. Tedy sítě se sníženou kapacitou se méně přeučí, ale také mohou dosahovat nižších výsledků z důvodu nedostatečné kapacity. Platí jednoduché pravidlo: pokud se síť v průběhu trénovacích epoch drží nastaveného průměru, je kapacita sítě nastavena správně,[1]s.107, pokud validace klesá a nebo pokud dochází k markantním výkyvům, je kapacita sítě nadhodnocena a je vhodné zmenšit počet prvků v plně propojených vrstvách.

Pokud plně propojená vrstva (Dense) má 64 prvků, a ztráta sítě po několika epochách tréninku stoupá, je vhodné s modelem experimentovat a upravit z 64:

```
model.add(layers.Dense(64, activation='relu'))
```

na 32 prvků ve vrstvě:

```
model.add(layers.Dense(32, activation='relu'))
```

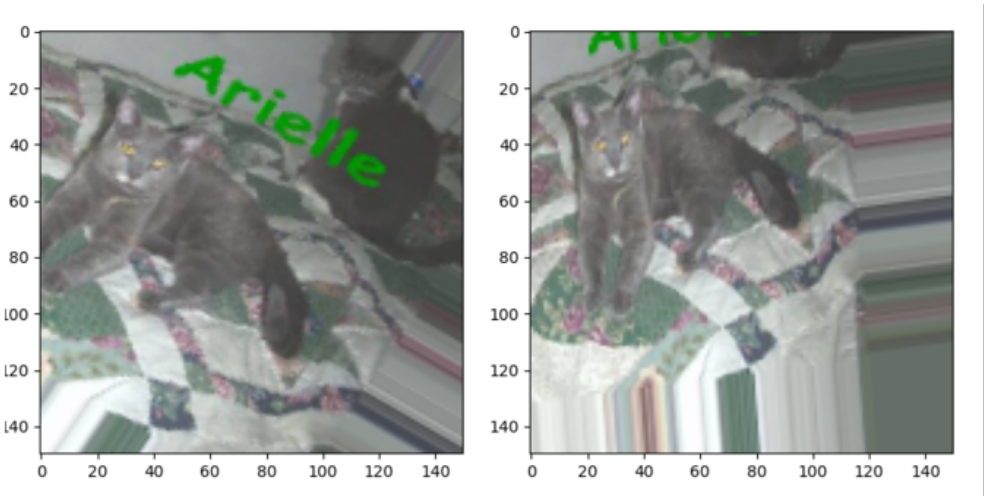
A nebo postupovat obráceně, tedy zvětšením kapacity, pokud je validační ztráta menší než trénovací.

4.1.3 Použití ImageDataGenerator

Tato třída rozšiřuje datový soubor dle zadaných parametrů, implementace začíná `import from keras.preprocessing.image import ImageDataGenerator`, poté probíhá definování toho, jakým způsobem jsou data změněna před použitím v trénovacím cyklu. U malých souborů je tato metoda hojně používaná, sníží míru přeučení. Nové obrázky pak vycházejí z konečného počtu těch, co jsou obsaženy v datovém souboru Dogs vs Cats.[1]s.134. Příklad rozšířeného obrazu je na obrázku č.17.

```
datagen = ImageDataGenerator(
```

```
rotation_range=50, #tento parametr určuje úhel rotace
width_shift_range=0.4, #posunutí do šířky
height_shift_range=0.3, #posun výškový
shear_range=0.3, #oříznutí
zoom_range=0.2, #zvětšení
horizontal_flip=True, #otočení podél vodorovné osy
fill_mode='nearest' #vyplnění volného místa po ořezu nejbližšími pixely
```



Obrázek 17: Výstup ImageDataGenerator, Zdroj: Vlastní

4.1.4 Přidání váhové regularizace.

Dle [1]s.108 jde o zmenšení vlivu vah v síti. Síť tedy nebude nárazově ovlivňována vzory, které zpracuje ale všem vzorům bude snížena hodnota, tedy zjištěný markant bude mít snížený vliv na úpravu vah při rozpoznávání nebo klasifikaci. Zvětší se schopnost sítě generalizovat, tedy ukládat obecnější vzory toho co „vidí“ Jinými slovy, snižuje složitost celé sítě, jelikož regularizace vah omezí velký vliv různých příznaků a síť si bude ukládat jen příznaky obecnějšího charakteru[34]

Rozlišujeme dva druhy regularizace, L1 a L2. První způsob reguluje váhy úměrou k absolutní hodnotě kooeficientů a druhý úměrou k druhé mocnině hodnot koeficientů [1]s.108. V Kerasu se implementuje přidáním parametru do plně propojené vrstvy například takto:

```
kernel_regularizer=regularizers.l1(0.001)
```

Tolik k teorii k základním technikám snížení přeučení při práci s malým datovým souborem.

4.2 Příprava datového souboru

Základním a prvním krokem v případě použití jiného datového souboru, než který je obsažen v knihovně Keras nebo v Tensorflow, by měla být vždy příprava datového souboru. Pokud budeme řešit úplně nový neobvyklý problém, budeme muset data i ručně označit, tedy předem je vyhodnotit, a v případě, že budou vyhovovat našemu úkolu, označit je popisky tříd (labels) co obsahují, abychom měli vůbec materiál na trénování sítě.

V této práci bude pro praktický úkol binární klasifikace použit již zmíněný datový soubor Dogs vs Cats, získaný mimo knihovnu Keras. Ten je dostupný ke stažení na komunitních stránkách Kaggle, na adrese: https://www.kaggle.com/c/dogs-vs-cats/data. Na těchto komunitních webových stránkách se členové zaměřují na soutěže v hlubokém učení, sdílí zdrojový kód a datové soubory. Soubor dogs vs cats je 854 MB velký, obsahuje 12500 testovacích neoznačených obrázků a dalších 25000 trénovacích obrázků, které využijeme v této práci. Tyto obrázky jsou označeny názvem toho, zda obsahují obraz kočky nebo psa. Jedná se o různá plemena těchto domácích zvířat, rozdělených do dvou skupin. Pro testování různých architektur využijeme a aplikujeme metody popsané v [1]s.142 na síti VGG16.

Po stažení souboru je třeba vytvořit trénovací, validační a testovací množinu malého souboru, tedy 2000 pro trénink, 1000 pro testování 1000 pro validaci. Toho lze dosáhnout dvěma způsoby, například kopírováním v souborovém manažeru, což je proveditelné v rámci malých souborů, nebo využitím Python modulů `os` a `shutil` . Modul os obsahuje funkce pro interakci Pythonu s operačním systémem a shutil pro práci s daty a adresáři. Příprava a rozdělení dat podle [1]s.128, pak vypadá pro tréninková data takto:

Vytvoříme soubor s koncovkou „.py“ a příhodným názvem, který poté spustíme.

```
import os
import shutil
```

Importujeme potřebné moduly, os obsahuje funkce pro práci s operačním systémem a shutil pro operace se soubory a jejich kolekcemi.

```
original_dataset_dir = '/home/peky/catdogs/train'
```

Nastavíme cestu ke složce s trénovacími obrázky a uložíme ji do proměnné.

```
# mensi soubor dat
base_dir = '/home/peky/PycharmProjects/CNN/DataCNN/catdogsmall'
os.mkdir(base_dir)
```

Vytvoříme základní adresáře malého souboru.

```
# vytvoreni adresare pro train
train_dir = os.path.join(base_dir, 'train') os.mkdir(train_dir)

   # trenovaci kocky
train_cats_dir = os.path.join(train_dir, 'cats') os.mkdir(train_cats_dir)

   # trenovaci psi
train_dogs_dir = os.path.join(train_dir, 'dogs')
os.mkdir(train_dogs_dir)
```

Vytvoříme cílové adresáře uvnitř složky.

```
fnames = ['cat..jpg'.format(i) for i in range(1000)]
for fname in fnames:
   src = os.path.join(original_dataset_dir, fname)
   dst = os.path.join(train_cats_dir, fname)
   shutil.copyfile(src, dst)


fnames = ['dog..jpg'.format(i) for i in range(1000)]
```

```
for fname in fnames:
    src = os.path.join(original_dataset_dir, fname)
    dst = os.path.join(train_dogs_dir, fname)
    shutil.copyfile(src, dst)
```

Pro samotné kopírování je pak využit cyklus for, který využívá toho, že obrázky psů a koček jsou očíslované od 0 do 12500 a vybere vždycky prvních 1000.

Data jsou tedy poté připravena ve třech adresářích train, test a validation, v každém z nich jsou dvě složky pojmenované dogs a cats, v nich je 1000 obrázků v případě tréninkové složky, 500 v případě validační a testovací. Tímto jsou data rozdělena a připravena pro použití. Skripty pro tvorbu datových souborů, tedy úpravu počtu souborů a složek, jsou v tomto ohledu velmi praktické, ve chvíli, kdy by datový soubor obsahoval množství obrázků v řádu desetitisíců, byla by jejich úprava jinou cestou, než tvorbou obdobného skriptu, přinejmenším komplikovaná.

4.3 Základní síť vytvořená v Kerasu a její popis

Při použití datového souboru, který je složen z obrázků typu .jpeg, .bmp, .png, je třeba několik jednoduchých přípravných kroků, předtím, než je možno přistoupit k tréninku. Rozdělení na trénovací, validační a testovací složku bylo provedeno v minulé podkapitole. Další potřebné kroky jsou popsány níže. Použitý dataset Dogs vs Cats, jak název napovídá, je soubor různých fotografií kočky nebo psa. V další části této práce tento datový soubor, využijeme k tréninku konvoluční neuronové sítě, která bude naším referenčním modelem. S ním pak budeme porovnávat již naučené sítě při řešení problému binární klasifikace. To vše s využitím technik pro snížení přeučení. Níže je popsána architektura referenční sítě pro dataset Dogs vs Cats, bylo postupováno dle metodiky od tvůrce knihovny Keras, François Chollet [1]s.130.

V prvním případě bude použit celý dataset 25000 obrázků, rozdělených do 3 skupin, 22000 jako trénovací, 2000 jako validační a 1000 jako testovací. Na tomto datasetu bude naučena malá konvoluční neuronová síť a na závěr bude otestována. Výsledky budou použity jako referenční základ pro porovnání s výsledky na zmenšeném datovém souboru.

V zásadě se dá tento kód rozdělit na několik segmentů, které lze popsat takto:

```
import os
import matplotlib.pyplot as plt
from keras import models
from tensorflow import optimizers
from tensorflow.python.keras.preprocessing.image_dataset //
import image_dataset_from_directory
from keras.layers import Conv2D, MaxPooling2D,//
Dropout, Flatten, Dense, BatchNormalization
```

V této části jsou importované moduly, které jsou dále využity a které obsahují funkce nutné k běhu, import datového souboru uloženého v knihovně, třídy modelů, vrstev a optimalizační funkce "binary_crossentropy", a grafická matematická knihovna matplotlib.pyplot slouží v tomto případě k vykreslení hodnot v průběhu trénování.

```
base_dir = '/home/peky/PycharmProjects/cnn/DataCNN/catdogs/'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')
```

Definování cest k datům, pomocí python knihovny `os`

```
train_ds = image_dataset_from_directory(
directory=train_dir,
labels='inferred',
label_mode='binary',
batch_size=16,
image_size=(150, 150),
color_mode="grayscale",
shuffle=True)
```

```
validation_ds = image_dataset_from_directory(
directory=validation_dir,
```

```
labels='inferred',
label_mode='binary',
batch_size=16,
image_size=(150, 150),
color_mode="grayscale",
shuffle=True)


test_ds = image_dataset_from_directory(
directory=test_dir,
labels='inferred',
label_mode='binary',
batch_size=16,
image_size=(150, 150),
color_mode="grayscale",
shuffle=True)
```

Zavolání funkce `image_dataset_from_directory()` s parametry adresář a nastavení popisků vrátí dávky obrázků z obou adresářů s popisky ve tvaru 0 pro první skupinu a 1 pro druhou skupinu, které jsou poté připraveny pro použití v modelu.

Co se týče nastavení parametrů, třída počítá s cestou k adresáři, označení popisků, tedy v případě `inferred` jde o převzetí označení z názvu složky, `label_mode` nastaven na `binary` vzhledem k tomu, že se jedná o binární klasifikaci.[45]

V této části se nastavuje typ sítě, v tomto případě Sekvenční, tedy vrstvy seřazené za sebou, přičemž první vrstva je vstupní konvoluční, s filtrem 3x3 pixely, obsahuje informaci o vstupním tvaru a je aktivována funkcí Relu, poté následuje vrstva MaxPooling, která vybírá maxima dané plochy. Také je využita normalizační vrstva BatchNormalization, která se stará o zmenšení odchylek vstupu, tedy že hodnoty jsou blíže 0 a v případě odchylky se blíží 1, dojde k odstranění extrémních hodnot.

Dále vrstva Výpadku, která vypíná části neuronů ve vrstvách, aby zmírnila přeučení a zmenšila validační ztrátu. Vrstva Flatten převede hodnoty do skalárního tvaru a závěrečné dvě plně propojené vrstvy, na základě zjištěných hodnot klasifikují obrázek, přičemž podle hodnoty v intervalu (0;1) funkce sigmoid, je zjištěno jaký objekt síť predikuje.

```
model = models.Sequential()
model.add(Conv2D(64,(3,3),activation='relu',input_shape=(150, 150, 3)))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))
model.add(Conv2D(128,(3,3),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))
model.add(Conv2D(256,(3,3),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))
model.add(Conv2D(64,(3,3),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))
model.add(Conv2D(512,(3,3),activation='relu'))
model.add(BatchNormalization())
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(1024,activation='relu'))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(1,activation='sigmoid'))
```

V této části se model zkompiluje, po zadání typu optimizeru ztrátové funkce a metody měření. V tomto případě optimalizátorem trénovacího pokroku je RMSprop, v případě klasifikace binárních objektů je vhodné užít ztrátovou funkci binary crossentropy.[1]s.114

```
model.compile(optimizer=optimizers.RMSprop(learning_rate=3e-5),
loss='binary_crossentropy', metrics=['acc'])
```

Zde se nastavuje trénovací cyklus, modelu se zadá soubory s daty a označením na kterém se učí, nastaví se počet trénovacích epoch, tedy to, kolikrát uvidí model celý datový soubor, a velikost dávky `batch_size`, která je definována v argumentu funkce `image_dataset_from_directory()` a znamená, po kolika příkladech bere trénovací data z celého souboru. [41]

```
history = model.fit(train_ds,
epochs=50,
validation_data=validation_ds,
validation_steps=20)
```

Tato část provádí testování kvality modelu na testovacích datech a poté vypíše výsledek.

```
test_loss, test_acc = model.evaluate(test_ds, steps=20)
print('test acc', test_acc)
```

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']
```

Definování hodnot validační a tréninkové přesnosti, tréninkové a validační ztráty z atributů objektu History, který v obsahuje hodnoty history.history získané za každou epochu. Výpis obsažených hodnost vypadá takto:

```
dict_keys(['loss', 'acc', 'val_loss', 'val_acc'])
```

V tomto kroku vytvoří funkce range číselnou řadu podle počtu epoch, která je poté využita jako osa grafu.

```
epochs = range(1, len(acc) + 1)
```

Grafové funkce z knihovny matplotlib.pyplot, umožňují graficky znázornit průběh tréninku

sítě. Tato část definuje hodnoty, které jsou zaznamenávané při tréninku a pro každou etapu. Ty poté zanese do dvou grafů, které na konci zobrazí. Trénovací přesnost a ztrátu pomocí bodů a validační přesnost a ztrátu pomocí linek.[42]

```
plt.plot(epochs, acc, 'bo', label='Přesnost trénování')
plt.plot(epochs, val_acc, 'b', label='Přesnost validace')
plt.title('Validační a tréninková přesnost')
plt.legend()

plt.figure()
plt.plot(epochs, loss, 'bo', label='Ztráta trénování')
plt.plot(epochs, val_loss, 'b', label='Ztráta validační')
plt.title('Tréninková a validační ztráta')
plt.legend()
plt.show()
```

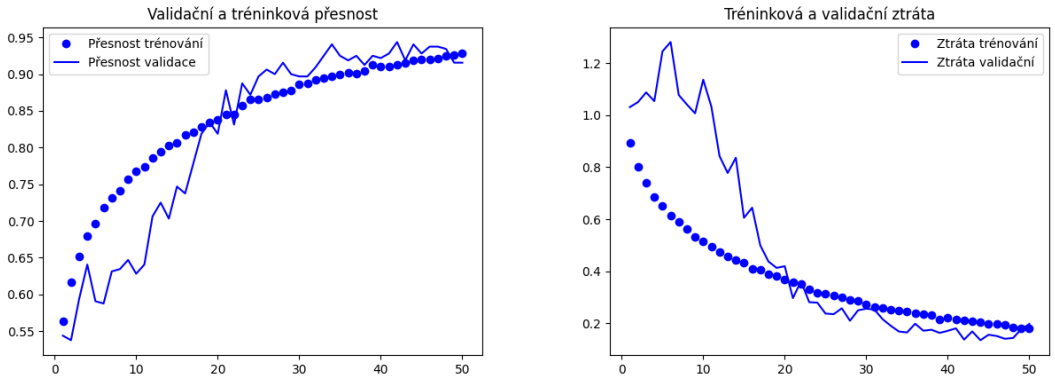
Výsledné grafy s vyznačenými výsledky poté ukazují jejich změny v průběhu tréninkových epoch.

4.3.1 Výsledky na celém datovém souboru

Od začátku trénovaná výše uvedená síť, jednoduchý sekvenční model, skládající se z 5 konvolučních a 2 plně propojených vrstev, dosáhl tréninkové přesnosti 92%, přičemž validační přesnost dosáhla 91% a ztráta dosahovala 19,7%. Tyto výsledky můžeme považovat za přijatelné, validační a tréninková přesnost se téměř rovnají, tedy model se nepřeučuje ani nepodučuje. Názorně je vše vidět v grafu na obr. č. 18, ve zhruba první polovině trénovací přesnost dosahuje vyšších výsledků než validace, načež kolem 25. epochy dojde ke vyrovnání. Testováním prokázal model 92,5% úspěšnost. Trénink trval zhruba 1,5 hodiny při 50 tréninkových epochách a délce trvání jedné epochy průměrně 115 sekund.

4.3.2 Výsledky tréninku na malém datasetu s třídou Imagedatagenerator

Pro účely druhého referečního modelu pro porovnání výsledků bude tento dataset zmenšen na 2000 trénovacích obrázků a 1000 pro testování a 1000 pro validaci modelu a bude použita



Obrázek 18: Výsledky tréninku na velkém datovém souboru, Zdroj: Vlastní

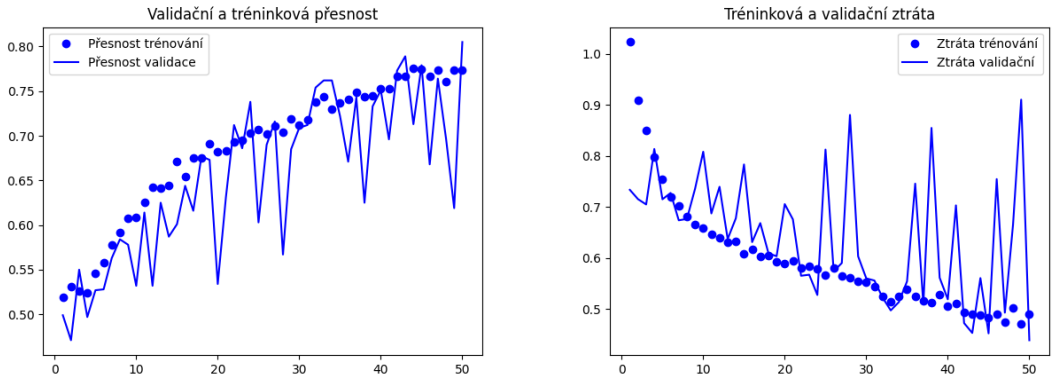
třída Imagedatagenerator pro rozšíření datového souboru, ke snížení přeučení.

Pro všechny tabulky výsledků tréninku modelů neuronových sítí bude hodnoty vysvětlovat tato legenda:

- trénovací ztráta - Tr-Z
- trénovací přesnost - Tr-Př
- validační ztráta - Val-Z
- validační přesnost - Val-P
- testovací přesnost - Te-P
- testovací ztráta - Te-Z

Velikost datového souboru	**Tr-Z**	**Tr-Př**	**Val-Z**	**Val-P**	**Te-P**	**Te-Z**
22000 tréninkových obrázků	0.1804	0.9283	0.1972	0.9156	0.9250	0.1806
2000 tréninkových obrázků	0.4814	0.7708	0.4385	0.8050	0.4780	0.7870

Tabulka 1: Tabulka rozdílu datových souborů



Obrázek 19: Výsledky na malém datovém souboru Cats vs Dogs, Zdroj: Vlastní

Výsledné grafy na obr. č.19 jsou daleko za výsledky tréninku na celém datovém souboru, dosahují 77%, s validační ztrátou 43% na dávce, testovou přesností 78% při 50 trénovacích epochách a době trénování 2 minut. Jak bylo očekáváno, jsou výsledky tréninku na 2000 obrázcích dramaticky horší, než ty kterých bylo dosaženo na 22000 exemplářích, tedy celém datasetu. Dalším laděním modelu by podle [1]s.138 šlo výkon posunout ještě výše, zhruba k 86% přesnosti. Takové ladění je ale bohužel z praktického hlediska zbytečné, validační ztráta, která ukazuje kvalitu modelu bude pořád výrazně vyšší, než u předtrénovaných modelů prezentovaných v další části této práce.

4.3.3 Reálný dopad technik proti přeučení

V rámci ladění referenčního modelu nad datasetem dogs vs cats, bylo ozkoušeno a zaznamenáno, jak různé změny kofigurace mohou ovlivnit výsledky. Jedná se o zjištění, jak velký je rozdíl v použití vrstvy Výpadku, vynechání vrstvy BatchNormalization nebo jak se kvalita změní při použití jiné rychlosti učení.

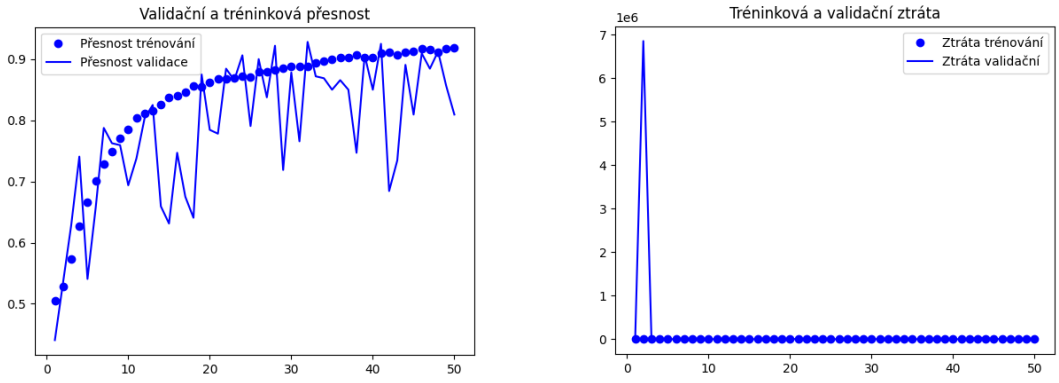
Natrénovaný model	**Tr-Z**	**Tr-Př**	**Val-Z**	**Val-P**	**Te-P**	**Te-Z**
Sekvenční model	0.1804	0.9283	0.1972	0.9156	0.9250	0.1806
Sek. model bez Dropout	0.0232	0.9919	0.8808	0.8625	0.8938	0.6282
Sek. mod. bez Batchnorm.	0.6244	0.7689	0.6807	0.5469	0.5719	0.6837

Tabulka 2: Tabulka výsledků úprav modelu

Jak je vidět v tabulce, vrstva Výpadku má zcela zásadní význam, model se okamžitě začne přizpůsobovat konkrétním datům v souboru, a validační ztráta stoupne na téměř 88%

oproti 19% při použití vrstvy Výpadku s nastavením na (0.25). Vrstva Batchnormalization ovlivní model relativně méně, než vynechání vrstvy Výpadku, ale i tak je kvalita modelu o 38% nižší.

Další věcí je zvolení správné rychlosti učení, v případě zadání extrémní hodnoty může jinak dobře nakonfigurovaný model dávat velice špatné výsledky jako na obrázku č.20. V tomto případě byla hodnota optimizeru RMSprop nastavena na 5e-2, jedná se o velmi vysokou hodnotu a model poté místo výsledků v tabulce 1 vykazuje velmi vysoké hodnoty validační ztráty 38.5013, testovací ztráty 88.4415. V případě podobných výsledků tréninku je třeba jako první věc zkontrolovat a snížit hodnotu `learning_rate`, nechat model znovu trénovat a teprve poté, co narazí na omezení a skončí v nějakém lokálním minimu gradientu a nebude se moci pohnout dál, můžeme znovu začít zvyšovat hodnotu.



Obrázek 20: Výsledek špatně nastavené learning_rate, Zdroj: Vlastní

4.4 Konfigurace sítí pro různá použití Transfer learning

Při použití předtrénované sítě je několik způsobů, jak tuto techniku provést, jedná se o: extrakce vah z předtrénované sítě, použití celé natrénované báze na trénovacím souboru dat a jemné ladění. Tedy následné trénování jen několika vrchních vrstev sítě. Podle [1]s.139 je lze charakterizovat takto:

- Extrakce příznaků z předtrénované sítě bez rozšíření dat.
  V této technice se využije uložených příznaků v natrénované konvoluční síti, princip je takový, že se zaznamenávají výstupy této natrénované sítě přitom jak prochází datovým souborem, poté se tyto výstupy předloží klasifikátoru, který je složen ze dvou plně propojených vrstev (Dense). A výstupem je hodnota pravděpodobnosti v intervalu (0,1) která určí, zdali se jedná o psa nebo kočku. Tato technika je velmi rychlá a nenáročná na výkon, neposkytuje ale možnosti ladění a úprav a využití rozšíření dat.

- Extrakce příznaků s rozšířením dat.
  Tato technika spočívá v použití naučené sítě, jako vrstvy v novém modelu, přičemž se odstraní poslední část, naučený klasifikátor, v tomto případě naučený na rozpoznání 1000 kategorií sítě Imagenet. Použije se nový, náhodně inicializovaný, který poté na základě výstupu hodnot a tréninku provede klasifikaci. Která spočívá v přiřazení získaných příznaků ke konkrétní entitě, také se zde použije zmrazení naučené sítě, aby se uchovaly již uložené hodnoty.

- Jemné doladění.
  Tato technika se používá, pokud je již natrénován nový klasifikátor, povolí se aktualizace několika vrchních vrstev. Současně dojde k nastavení velmi malé rychlosti učení optimalizátorem RMSprop, čímž dojde pouze k malým změnám uložených příznaků a síť se lépe přizpůsobí předkládaným datům.

Každá tato technika poskytuje různě kvalitní výsledky, stejně tak je jinak náročná na čas a výpočetní výkon.

V této části vycházíme z těchto předpokladů:

- Nemáme dostatečně velký datový soubor pro naučení sítě pro řešení problému.

- Nemáme výkonný hardware, zejména GPU schopnou vysokého výkonu při paralelních výpočtech.

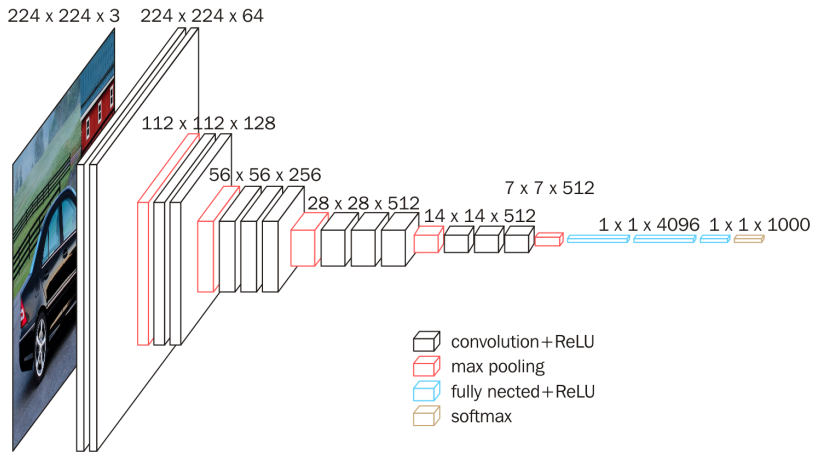
Bohužel se nejedná o předpoklady nereálné, v dnešní době (2020/2021) jsou výkonné grafické karty prakticky nedostupné, ve velkém množství skupovány překupníky, kteří profitují z nedostatečné nabídky a také lidmi, co je využívají na těžbu kryptoměn.[50]

Jako první využijeme naučenou síť VGG16. Tato síť, stejně jako ostatní, které budou použity, je obsažena v knihovně Keras a její použití pro transfer learning popisuje tvůrce Kerasu François Chollet [1]s.141. Popsaných metod bylo využito v této práci a při použití jiných předtrénovaných modelů.

4.4.1 VGG16

Síť VGG16 byla vytvořena pro soutěž ImageNet Large Scale Visual Recognition Challenge 2014 na datovém souboru Imagenet Karen Simonyan a Andrewem Zissermanem a skupinou Visual Geometry Group, University of Oxford[35] Z toho vychází název sítě a číslo 16 označuje počet konvolučních vrstev. [35] Ty jsou rozděleny do 5 bloků po dvou až třech vrstvách, s aktivační funkcí ReLu, mezi kterými jsou vrstvy Max-pooling a na konci je blok tří plně propojených vrstev, z nichž poslední provádí vyhodnocení vstupu s aktivační funkcí softmax. (obrázek č.21)

Tato síť v roce 2014 dosáhla druhého místa v soutěži za sítí GoogLeNet, zajímavostí také je, že na nejvýkonnější grafické kartě tehdejší doby Nvidia Titan trvalo trénování 2 až 3 týdny.[36]



Obrázek 21: Architektura VGG16, Zdroj: https://cdn-images-1.medium.com/max/1600/1* A_uro4dKSBUPWfng8jrPug.png

První technikou je extrakce příznaků, pro VGG16 předpokládáme úspěšné vytvoření malého datasetu podle kapitoly 4.2, přičemž je praktické uložit ho mimo kód s konfigurací sítě pro další využití a nebo pozdější úpravy datového souboru.

```
from keras.applications import VGG16
from keras import models
from keras import layers
from keras import optimizers
import os
import numpy as np
import matplotlib.pyplot as plt
from keras_preprocessing.image import ImageDataGenerator
```

Importujeme všechny potřebné moduly, předtrénovanou síť VGG16, třídy models, layers, optimizers, knihovnu os, knihovnu numpy pro vytvoření pole příznaků a matplotlib.pyplot k vytvoření grafu výsledků, stejným způsobem jako v první síti a na závěr třídu ImageDataGenerator. Jedná se o univerzální nástroj pro přípravu a úpravu datových souborů pro využití při tréninku konvolučních sítí.

```
conv_base = VGG16(weights='imagenet',
include_top=False,
input_shape=(150, 150, 3))
```

Zde definujeme conv_base, tedy instanci třídy VGG16, která obsahuje několik atributů. První je `weights`, ten určuje počáteční nastavení vah. Druhý `include_top` určuje, zdali bude použit klasifikátor z tréninku na souboru Imagenet a třetí `input_shape`, zadává vstupní tvar dat. Tento je ve formátu výška, šířka, obrazová hloubka.

```
base_dir = '/home/peky/PycharmProjects/CNN/DataCNN/catdogsmall/'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')
```

Tato část definuje kmenový soubor s daty a pro další postup jsou nastaveny proměnné s cestami pro validační, testovací a tréninkový soubor.

```
datagen = ImageDataGenerator(rescale=1. / 255)
batch_size = 20
```

Změna hodnot z rozsahu [0, 255] na [0, 1] třídou ImageDataGenerator a nastavení velikosti dávky, počtu obrázků, na 20.

```
def extract_features(directory, sample_count):
features = np.zeros(shape=(sample_count, 4, 4, 512))
labels = np.zeros(shape=(sample_count))
generator = datagen.flow_from_directory(
   directory,
   target_size=(150, 150),
   batch_size=batch_size,
   class_mode='binary')
i = 0
for inputs_batch, labels_batch in generator:
   features_batch = conv_base.predict(inputs_batch)
   features[i * batch_size : (i + 1) * batch_size] = features_batch
   labels[i * batch_size : (i + 1) * batch_size] = labels_batch
   i += 1
   if i * batch_size >= sample_count:
   break
return features, labels
```

Výše uvedená část se od předchozího modelu odlišuje, provádí extrakci příznaků z obrázků a numpy.zeros je ukládá do pole o velikosti (počet příkladů, 4, 4, 512), poté vytvoří pole pro popisky, deklaruje generátor, který použije třídu ImageDataGenerator s metodou `flow_from_directory`, té jsou předloženy argumenty v podobě adresáře s daty. Cílovou velikostí obrázků (150 * 150) pixelů. V jakém množství budou v jedné tréninkové dávce. Atribut `class_mode` nastavuje typ popisků na binární, vzhledem k tomu, že máme dvě kategorie. Následuje cyklus for, který postupně pro všechny trénovací dávky, použije metodu

predict, která z konvoluční báze získá odhad toho, co sítí prochází. Takto pokračuje až do splnění ukončující podmínky a pokračuje až s dalším obrázkem.[1]s.142.

```
train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)
```

Zde jsou určeny příznaky a popisky pro trénink, validaci a testování modelu.

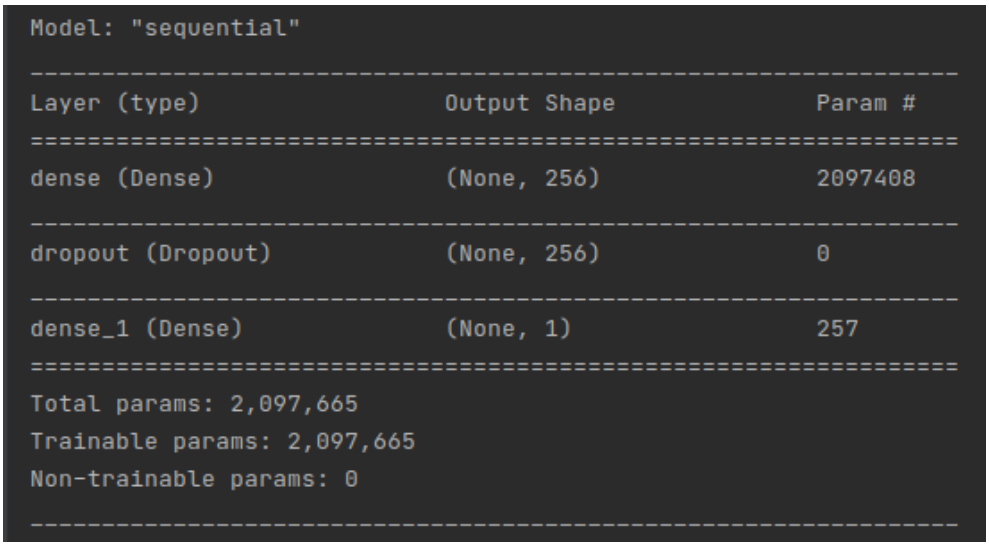
```
train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
test_features = np.reshape(test_features, (1000, 4 * 4 * 512))
```

Pro funkci klasifikátoru je třeba, aby se z polí staly skalární hodnoty, což provede metoda knihovny numpy reshape.

```
model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim= 4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))
```

Zde je definována instance modelu, v tomto případě se jedná o Sekvenční model. Následně je definován klasifikátor, který se skládá ze dvou plně propojených vrstev s vloženou vrstvou Výpadku, pro snížení míry přeučení klasifikátoru. Účinnost výpadku proti přeučení byla ověřena, v případě vynechání této vrstvy dosáhla přesnost 100%, ale hodnota validační ztráty se zdvojnásobila. Došlo tedy k výraznému zhoršení kvality modelu.

Parametry modelu lze zobrazit metodou summary `model.summary()`, jak je vidět na obrázku č.22, nový klasifikátor o dvou vrstvách `Dense` a vložené vrstvě `Dropout` obsahuje téměř 2 100 000 parametrů, které jsou trénovány.



Obrázek 22: Výpis metody summary, Zdroj: Vlastní

```
model.compile(optimizer=optimizers.RMSprop(lr=2e-5),
loss='binary_crossentropy',
metrics=['acc'])
```

Tato část určuje s jakými parametry se bude model trénovat, využívá optimalizátor `RMSprop`, ztrátovou funkci `binary_crossentropy` a bude měřit přesnost odhadu.

```
history = model.fit(train_features, train_labels,
epochs=30,
batch_size=20,
validation_data=(validation_features, validation_labels))
```

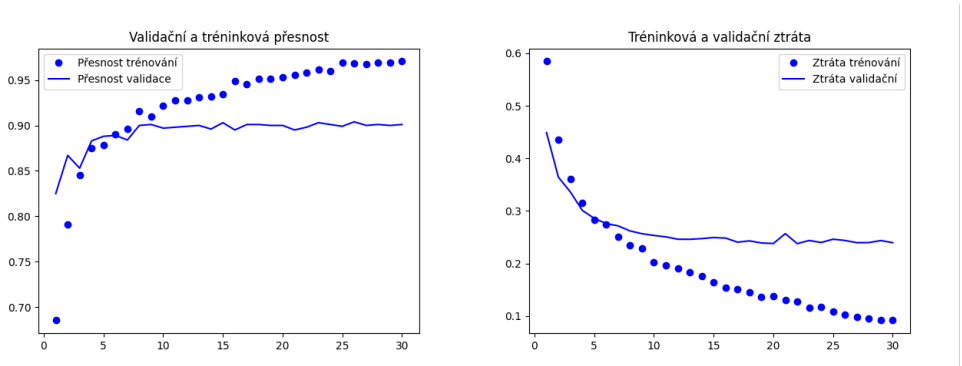
Využijeme třídy `history`, pro zaznamenání hodnot ztrátových funkcí a výkonu, nastavení počtu trénovacích epoch, velikost tréninkové dávky a cestu k validačním datům.

```
test_loss, test_acc = model.evaluate(test_features, test_labels, steps=50)
print('test acc', test_acc)
```

Vyhodnotíme model metodou `evaluate` na testovacích datech v 50 krocích s výpisem výsledků.

```
acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Přesnost trénování')
plt.plot(epochs, val_acc, 'b', label='Přesnost validace')
plt.title('Validační a tréninková přesnost')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Ztráta trénování')
plt.plot(epochs, val_loss, 'b', label='Ztráta validační')
plt.title('Tréninková a validační ztráta')
plt.legend()

plt.show()
```

Vytvoření grafů s výsledky, popsáno v kapitole 4.3



Obrázek 23: Výsledky VGG16 a extrakce příznaků, Zdroj: Vlastní

Po 30 epochách trénování dostáváme tyto hodnoty:

Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
VGG16 extr.	0.0920	0.9705	0.2397	0.9010	0.8870	0.3307

Tabulka 3: Tabulka extrakce příznaků VGG16

Zde je patrné, že validační přesnost se zastavila na 90% při 30 trénovacích epochách. Tréninková přesnost stoupala na 98%, což by v jiných případech naznačovalo velmi dobrý výsledek. Je ale nutno vzít v úvahu omezení této techniky a malý datový soubor, na kterém pracuje. Hodnota validační ztráty, která se pohybuje kolem 0.24, je v tomto případě mnohem více relevantní údaj pro hodnocení kvality modelu. Jedná se o velice rychlý proces, trénovací epocha i s validací trvala 4 milisekundy. Testování s více tréninkovými epochami bylo zbytečné, validační ztráta i přesnost se po 10 epochách zastavily na hodnotách, které potom zůstaly až do konce trénovacího cyklu stejné. Síť se velmi rychle adaptovala na konkrétní předkládaná data a zapamatovala si je, došlo k přeučení.

Druhá technika extrakce příznaků s rozšířením dat pomocí třídy `Imagedatagenerator`, pro VGG16 podle [1]s.142 přidává naučenou síť jako první vrstvu celého modelu. Zaměříme se pouze na změněný kód (identický vynecháme).

```
conv_base.trainable = False
```

Vzhledem k tomu, že smyslem této techniky je využít již naučené příznaky, je třeba „zmrazit“ celou použitou síť, nastavením parametru `trainable` na `False`, tím se zamezí změnám vah, ke kterým dochází v průběhu trénování.

```
model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))
```

V této části je přidán model jako vrstva `conv_base` a za ni je připojena vrstva Flatten pro upravení výstupu a dvě plně propojené vrstvy pro klasifikaci. Model vypadá potom takto: jediné parametry, kterým je umožněno trénování, jsou v posledních vrstvách. Spodní

část obsahuje počet parametrů, které jsou zmrazeny a poté počet 2097665 trénovatelných parametrů, který odpovídá počtu parametrů přidaného klasifikátoru, obr. č. 24.

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
vgg16 (Functional)           (None, 4, 4, 512)         14714688
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 256)               2097408
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 257
=================================================================
Total params: 16,812,353
Trainable params: 2,097,665
Non-trainable params: 14,714,688
_________________________________________________________________
```

Obrázek 24: Ověření zmrazení báze, Zdroj: Vlastní

```
train_datagen = ImageDataGenerator(rescale=1. / 255,
   rotation_range=40,
   width_shift_range=0.2,
   height_shift_range=0.2,
   shear_range=0.2,
   zoom_range=0.2,
   horizontal_flip=True,
   fill_mode='nearest')
```

Zde je využito vlastností třídy ImageDataGenerator pro rozšíření dat. Popsáno v podkapitole 4.1.

```
test_datagen = ImageDataGenerator(rescale=1. / 255)
```

Testovací data upravována nebudou, pouze jim budou normalizovány hodnoty do intervalu (0,1).

```
train_generator = train_datagen.flow_from_directory(train_dir,
target_size=(150, 150),
batch_size=20,
class_mode='binary')
```

Nastavení pro trénovací data spočívá v použití trénovacího generátoru s rozšířením obrázků a poté jeho spojení s metodou `flow_from_directory`, s nastavenými parametry, o cílových rozměrech velikostí jedné trénovací dávky a nastavením popisků.

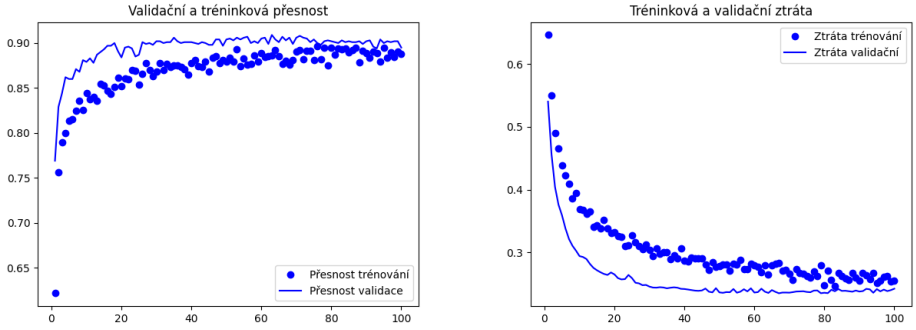
```
validation_generator = test_datagen.flow_from_directory(
validation_dir,
target_size=(150, 150),
batch_size=20,
class_mode='binary')



test_generator = test_datagen.flow_from_directory(
test_dir,
target_size=(150, 150),
batch_size=20,
class_mode='binary')
```

Data pro validaci a testování modelu, by neměla být upravená, je možné použít již definovaný `test_datagen` a použít ho pro validační i testovací data.

```
history = model.fit(train_generator,
steps_per_epoch=100,
epochs=100,
```

```
validation_data=validation_generator,
validation_steps=50)
```

Pro trénování modelu je nastavena třída `train_generator`, pro trénink na rozšířených datech, pro validaci na neupravených datech. Vzhledem k tomu že `batch_size` je nastavena na 20, potom nastavení `steps_per_epoch`, může být 100, čímž dostaneme 2000 obrázků, které máme v souboru a na těch provedeme 100 trénovacích cyklů.



Obrázek 25: Výsledky VGG16, zmrazená konvoluční báze, Zdroj: Vlastní

Po 100 epochách trénování dostáváme hodnoty uvedené v tabulce č. 4. Trénink jedné epochy trval 15 sekund, celý proces 25 minut.

V průběhu testování těchto konfigurací bylo zjištěno, že pro ověření možností dané architektury stačí 30 tréninkových epoch, další trénink je zbytečný protože kvalita modelu se nezlepšuje, a dochází jenom ke zvýšení hodnoty přeučení, což je vidět na křivkách validace, které oscilovaly na 0.925 v případě přesnosti a 0.24 v případě validační ztráty.

Pro úplnost, výkon modelu při 30 a 100 epochách v tabulce:

VGG16	**Tr-Z**	**Tr-Př**	**Val-Z**	**Val-P**	**Te-P**	**Te-Z**
100 trénovacích epoch	0.2551	0.8880	0.2424	0.8950	0.8970	0.2507
30 trénovacích epoch	0.2715	0.8900	0.2324	0.9090	0.8940	0.2449

Tabulka 4: Výsledky po 30 a 100 epochách

Jak je vidět, jsou téměř shodné, delší trénink je tedy zbytečný. Lepšího nebo stejného výsledku dosáhneme za 7,5 minuty místo za 25 minut.

Navazující technika jemného ladění mění hodnotu binárního parametru `trainable` na True. Tím je nastaveno, že vrstvy v naučené síti mohou měnit své hodnoty a za něj je přidán cyklus for s podmínkou, u které sítě jsou změny vah povoleny. V tomto případě jde o první konvoluční vrstvu v posledním konvolučním bloku.

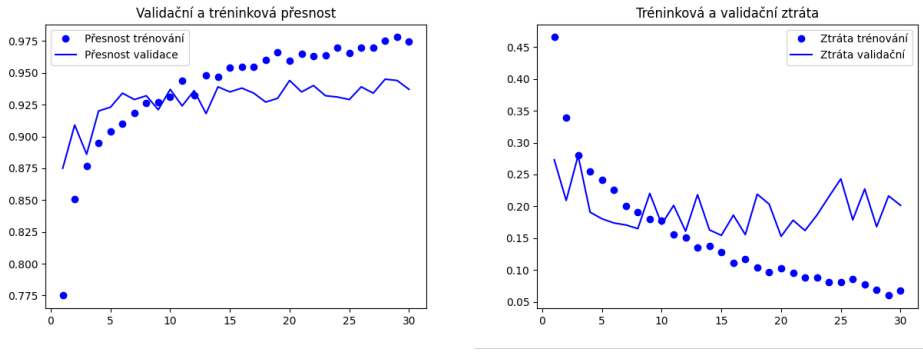
```
conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
   if layer.name == 'block5_conv1':
   set_trainable = True
   if set_trainable:
   layer.trainable = True
   else:
   layer.trainable = False
```

Doba jedné epochy se prodloužila na 19 vteřin, dochází ke změnám v 9,177,089 z celkových 16,812,353 parametrů, které síť obsahuje. Trénink trval 9,5 minuty a po 30 epochách trénování dostáváme tyto hodnoty:

Arch. a technika	**Tr-Z**	**Tr-Př**	**Val-Z**	**Val-P**	**Te-P**	**Te-Z**
VGG16 ladění	0.0637	0.9710	0.2305	0.9290	0.9350	0.2814

Tabulka 5: Tabulka jemného ladění VGG16

Díky rozmrazení horní vrstvy se podařilo zvýšit přesnost na 97%. Směrodatná je opět validační ztráta, ta je v tomto případě 0.2305, testovací přesnost, tedy na datech, která model neviděl, ta dosáhla 93.5% jedná se tedy opravdu o zlepšení, jak je vidět na obrázku 26.



Obrázek 26: Výsledky VGG16 jemné doladění, Zdroj: Vlastní

Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
VGG16 extr.	0.0920	0.9705	0.2397	0.9010	0.8870	0.3307
VGG16 extr. rozš.	0.2715	0.8900	0.2324	0.9090	0.8940	0.2449
VGG16 ladění	0.0637	0.9710	0.2305	0.9290	0.9350	0.2814

Tabulka 6: Tabulka srovnání výkonu VGG16

4.4.2 VGG19

Tato architektura, jak název naznačuje, která je téměř totožná s předchozí VGG16, Byla vybrána pro zjištění, zdali větší počet konvolučních vrstev, kterých je v tomto případě 19 místo 16, může znatelně ovlivňovat výsledky trénování na stejných datech. Jak je vidět na obr. č. 27, v předchozím případě šlo o konfiguraci „D“ a nyní využijeme konfiguraci „E“

Jak již bylo zaznačeno, jedná se o rozšíření architektury VGG16 a proto je jeho použití otázkou změny dvou parametrů:

```
from keras.applications import VGG19
from keras import models
from keras import layers
from keras import optimizers
import os
import numpy as np
import matplotlib.pyplot as plt
```

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Obrázek 27: Srovnání architektury VGG, Zdroj: https://arxiv.org/pdf/1409.1556.pdf

```
from keras_preprocessing.image import ImageDataGenerator
```

Importujeme VGG19 místo předchozí VGG16

```
conv_base = VGG19(weights='imagenet',
include_top=False, input_shape=(150, 150, 3))
```

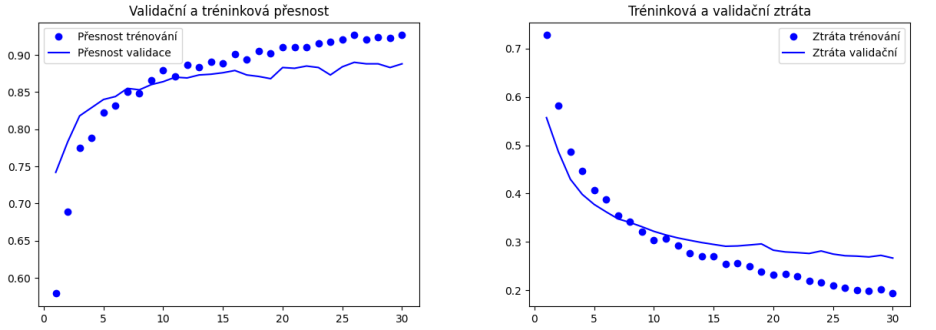
Zde upravíme stejným způsobem na VGG19, zbytek kódu je stejný jako u VGG16.

V případě extrakce příznaků VGG19 po 30 epochách trénování dostáváme tyto hodnoty:

Průběh učení lze vidět na obrázku č. 28

Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
VGG19 extr.	0.1773	0.9399	0.2755	0.8830	0.8720	0.3022

Tabulka 7: Tabulka extrakce příznaků VGG19



Obrázek 28: Graf výsledků VGG19, extrakce příznaků, Zdroj: Vlastní

Extrakce příznaků s rozšířením poskytla tyto výsledky:

Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
VGG19 extr.rozš.	0.3668	0.8183	0.2914	0.8850	0.8570	0.3335

Tabulka 8: Tabulka extrakce příznaků s rozšířením VGG19

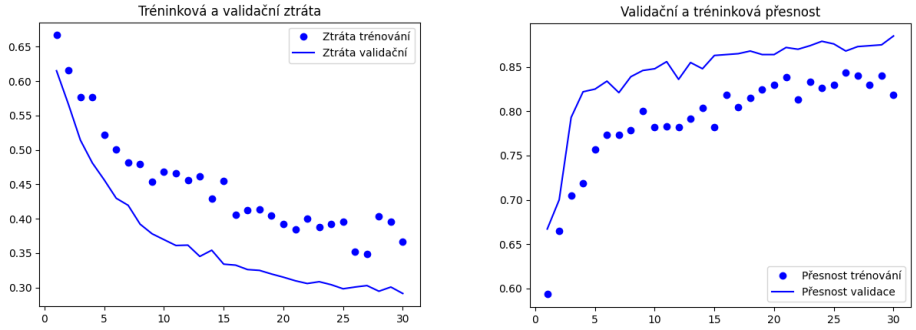
Trénink trval při 30 epochách 5 minut, Obrázek č. 29

Technika jemného ladění pak: Obrázek č. 30

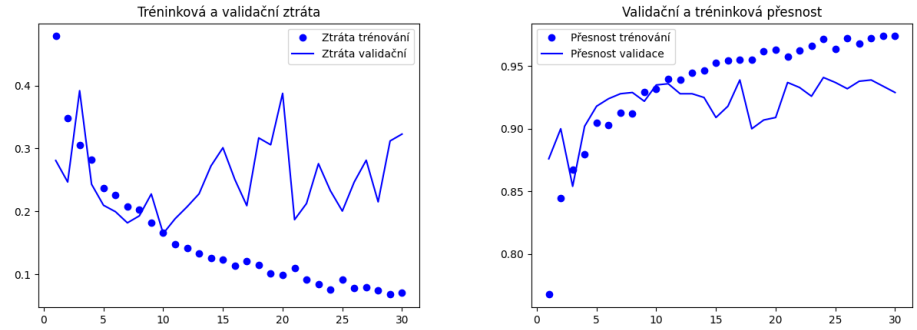
Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
VGG19 ladění	0.0717	0.9740	0.3229	0.9290	0.9260	0.4393

Tabulka 9: Tabulka jemného ladění VGG19

Pro přehlednost je vše uvedeno v tabulce č.10. Lze pozorovat, že i relativně zastaralá architektura může dosahovat výborných vysledků.



Obrázek 29: Graf výsledků VGG19, extrakce příznaků s rozšířením dat, Zdroj: Vlastní



Obrázek 30: Výsledky VGG19 jemné doladění, Zdroj: Vlastní

Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
VGG19 extr.	0.1773	0.9399	0.2755	0.8830	0.8720	0.3022
VGG19 extr.rozš.	0.3668	0.8183	0.2914	0.8850	0.8570	0.3335
VGG19 ladění	0.0717	0.9740	0.3229	0.9290	0.9260	0.4393

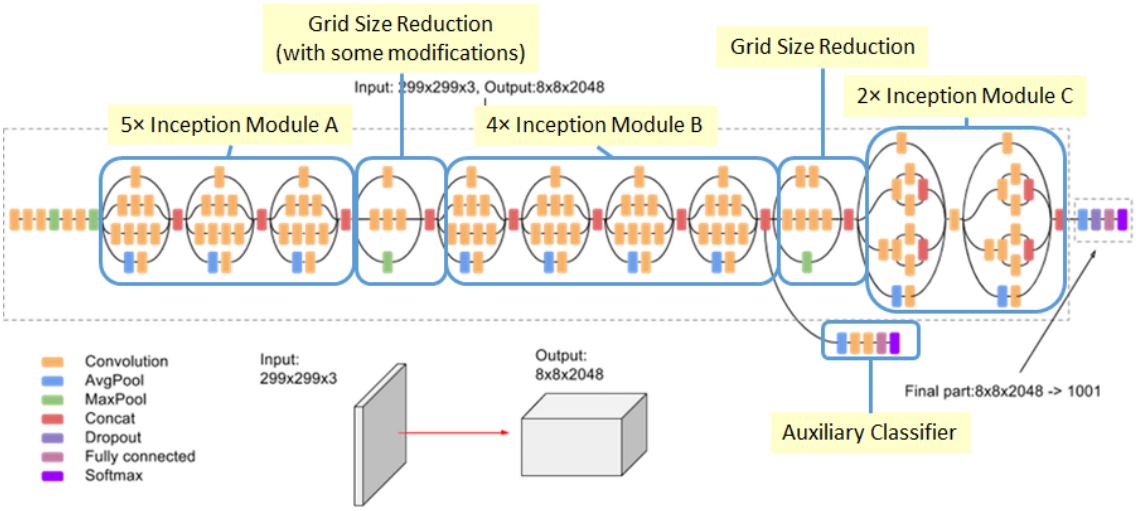
Tabulka 10: Tabulka srovnání výkonu VGG19

4.4.3 InceptionV3

Architektura Inception V3 vznikla v roce 2015[38] pokračováním a vylepšením architektury Inception V2. Tato architektura je poněkud odlišná od klasických sekvenčních modelů, její konvoluční bloky se nazývají Inception modules.

Principem těchto bloků je to, že používá současně několik jinak velkých filtrů v jedné vrstvě, jejichž výstup poté kombinuje a sloučí ve vrstvě MaxPooling.

Dále využívá toho, že počítání s filtry 3x3 je méně výpočetně náročné než s filtry 5x5,



Obrázek 31: Architektura InceptionV3, Zdroj: https://cdn-images-1.medium.com/max/1200 /1*gqKM5V-uo2sMFFPDS84yJw.png

proto je použito více menších. [38] a také užívá nesymetrické konvoluční filtry, v rozměru např. 1x3, 3x1, které opět snižují výpočetní náročnost.[38]

Tato síť má také přidané dodatečné klasifikátory a tak čelí problému mizejícího gradientu, který nastává v případě, že síť obsahuje velké množství vrstev, dochází pak při užití některých aktivačních funkcí jako například Sigmoid k tomu, že i velký vstup znamená velmi malý výstup, jelikož výstup této funkce je v intervalu od 0 do 1, tedy dochází ke ztrátě velkého množství dat. Tyto přídavné klasifikátory poté obsaženou ztrátu, kterou uchovaly, přičtou ke konečnému výsledku. Takto se zamezí ztrátě zpětné vazby sítě.[39]s.4

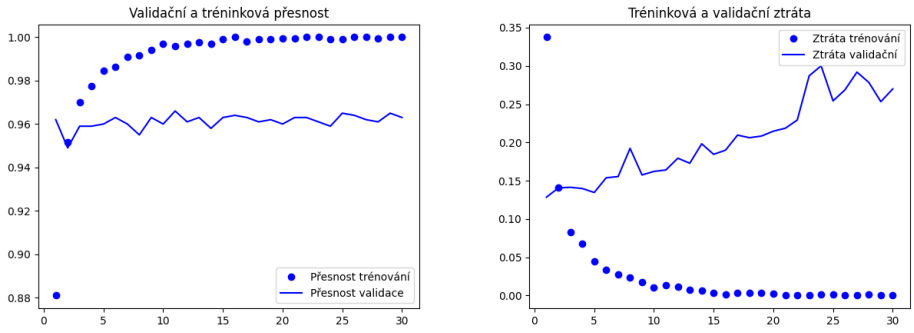
Technika extrakce příznaků

```
from keras.applications import InceptionV3
import os
import numpy as np
import matplotlib.pyplot as plt
from keras import models
from keras import layers
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers



conv_base = InceptionV3(weights='imagenet',
include_top=False,
input_shape=(150, 150, 3))


datagen = ImageDataGenerator(rescale=1. / 255)
batch_size = 20


def extract_features(directory, sample_count):
features = np.zeros(shape=(sample_count, 3, 3, 2048))
labels = np.zeros(shape=sample_count)
generator = datagen.flow_from_directory(
directory,
target_size=(150, 150),
batch_size=batch_size,
class_mode='binary')
i = 0 for inputs_batch, labels_batch in generator:
features_batch = conv_base.predict(inputs_batch)
features[i * batch_size: (i + 1) * batch_size] = features_batch
labels[i * batch_size: (i + 1) * batch_size] = labels_batch
```

```
i += 1
if i * batch_size >= sample_count:
break
return features, labels


train_features, train_labels = extract_features(train_dir, 2000)


validation_features, validation_labels = extract_features(validation_dir, 1000)


test_features, test_labels = extract_features(test_dir, 1000)


train_features = np.reshape(train_features, (2000, 3 * 3 * 2048))
validation_features = np.reshape(validation_features, (1000, 3 * 3 * 2048))
test_features = np.reshape(test_features, (1000, 3 * 3 * 2048))


model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=3 * 3 * 2048))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))


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


history = model.fit(train_features, train_labels,
epochs=30,
batch_size=20,
```

```
validation_data=(validation_features, validation_labels))
```

Jak je vidět v kódu uvedeném výše, je téměř identický s předchozím užitím architektur VGG16 a VGG19. S jednou výjimkou a to tvarem vstupního tensoru, definovaném v `input_dim`. Celková délka tréninku 1 minuta, obrázek č. 32.



Obrázek 32: Graf výsledků InceptionV3 extrakce příznaků Zdroj: Vlastní

Arch. a technika	**Tr-Z**	**Tr-Př**	**Val-Z**	**Val-P**	**Te-P**	**Te-Z**
InceptionV3 extr.	2.1942e-04	1.0000	0.2520	0.9660	0.9660	0.2429

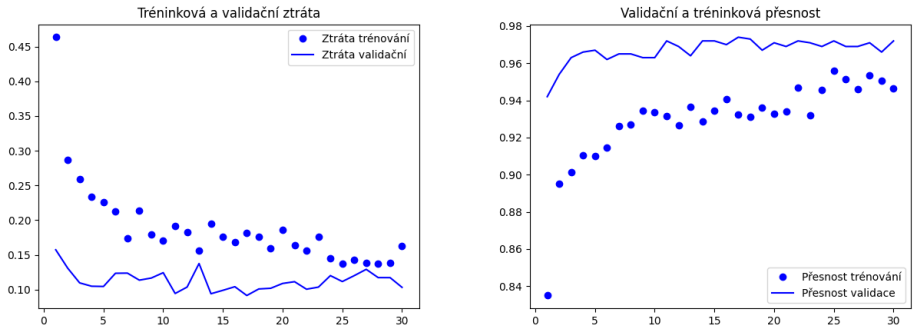
Tabulka 11: Tabulka extrakce příznaků InceptionV3

Extrakce příznaků s rozšířením dat, s celkovou dobou tréninku, 6 minut, obrázek č. 33

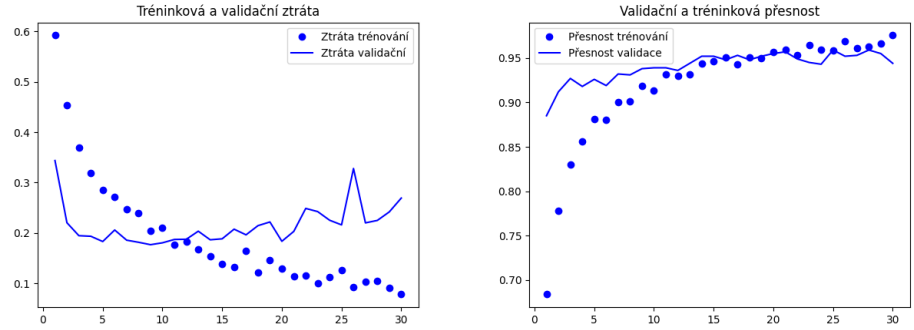
Arch. a technika	**Tr-Z**	**Tr-Př**	**Val-Z**	**Val-P**	**Te-P**	**Te-Z**
InceptionV3 extr. rozš.	0.1624	0.9465	0.1032	0.9720	0.9640	0.1182

Tabulka 12: Tabulka extrakce s rozšířením InceptionV3

Jemné ladění sítě InceptionV3, délka tréninku 10 minut.



Obrázek 33: Graf výsledků InceptionV3 extrakce s rozšířením, Zdroj: Vlastní



Obrázek 34: Výsledky InceptionV3 jemné doladění, Zdroj: Vlastní

Celková délka tréninku jemného ladění byla 10 minut, obrázek č. 34. Tabulka č.14 obsahuje výsledky všech tří použitých technik.

Arch. a technika	**Tr-Z**	**Tr-Př**	**Val-Z**	**Val-P**	**Te-P**	**Te-Z**
InceptionV3 ladění	0.0781	0.9755	0.2692	0.9440	0.9550	0.2078

Tabulka 13: Tabulka jemného ladění InceptionV3

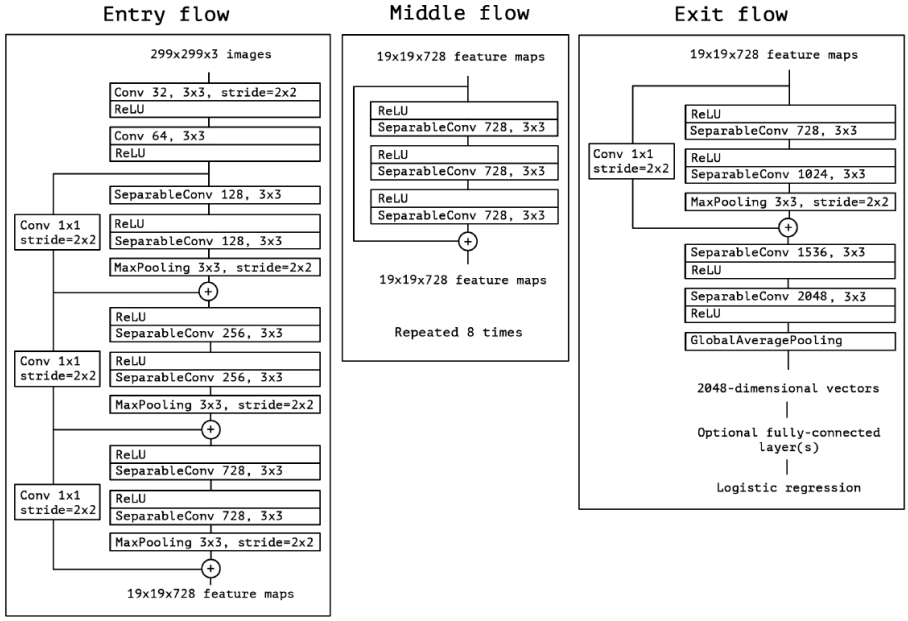
Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
InceptionV3 extr.	2.1942e-04	1.0000	0.2520	0.9660	0.9660	0.2429
InceptionV3 extr. rozš.	0.1624	0.9465	0.1032	0.9720	0.9640	0.1182
InceptionV3 ladění	0.0781	0.9755	0.2692	0.9440	0.9550	0.2078

Tabulka 14: Tabulka srovnání výkonu InceptionV3

4.4.4 Xception

Známá také jako Extreme Inception [47] vznikla jako vylepšení předchozí architektury InceptionV3 k práci nad velkými datovými soubory s důrazem zjednoduššení a zrychlení výpočtů použitím 36 hluboko rozdělitelných konvolučních vrstev (depthwise separable convolution layers) ve 14 propojených modulech. Ty narozdíl od klasických konvolučních vrstev, provádějí konvoluce ve dvou krocích[46]:

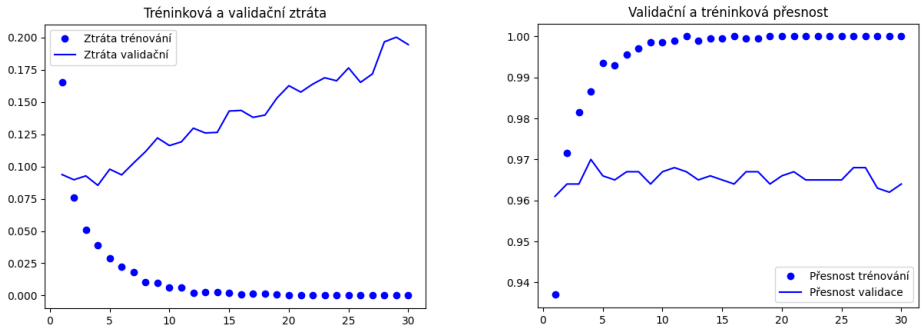
- Depthwise convolution, filtrační fáze
- Pointwise convolution, kombinační fáze



Obrázek 35: Xception, Zdroj: https://miro.medium.com/max/1400/1*hOcAEj9QzqgBXcwUzmEvSg.png

V první fázi hloubkové konvoluce dochází k tomu, že tensor dat, je rozdělen na části s hloubkou 1, tedy matice, na kterých jsou provedeny konvoluce filtrem 3x3. Druhá fáze spočívá v konvoluci filtrem 1x1, ten kombinuje všechny za sebou jdoucí hodnoty v tensoru. Myšlenka za tímto uspořádáním je taková: Fáze mapování příznaků je možno úplně oddělit. Tímto krokem dojde ke snížení množství výpočetního času, který je jinak třeba v klasické konvoluční síti. [47]

Dále je celý proces rozdělen, na tři části, přičemž prostřední (middle) je osmkrát opakována, jak je vidět z obrázku 35.



Obrázek 36: Graf výsledků Xception extrakce příznaků, Zdroj: Vlastní

Extrakce příznaků po 30 epochách trénování, při celkové době tréninku 1 minuty, průběh tréninku je vidět na obrázku č. 36, dostáváme hodnoty uvedené v tabulce č. 15.

Arch. a technika	**Tr-Z**	**Tr-Př**	**Val-Z**	**Val-P**	**Te-P**	**Te-Z**
Xception extr.	1.6504e-05	1.0000	0.1946	0.9640	0.9660	0.2369

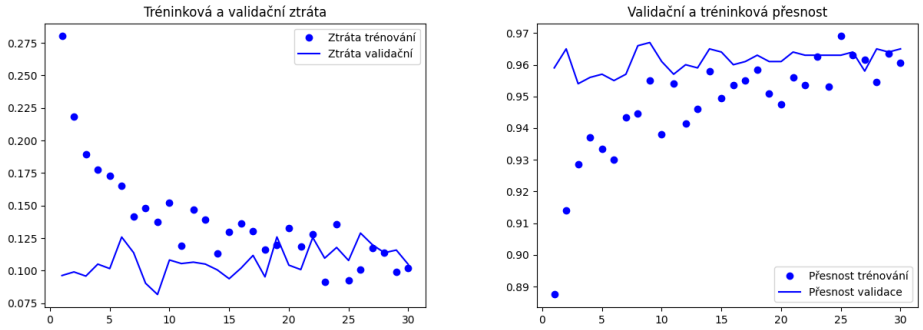
Tabulka 15: Tabulka extrakce příznaků Xception

Extrakce příznaků s rozšířením dat, po 30 epochách, celkem 7,5 minutách, trénování dostáváme hodnoty uvedené v tabulce č.16.

Čas tréninku s technikou jemného ladění trval opět 7,5 minuty, po 30 epochách trénování dostáváme hodnoty uvedené v tabulce č.17.

Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
Xception extr. rozš.	0.1023	0.9605	0.1048	0.9650	0.9710	0.1236

Tabulka 16: Tabulka extrakce s rozšířením Xception



Obrázek 37: Graf výsledků Xception extrakce s rozšířením, Zdroj: Vlastní

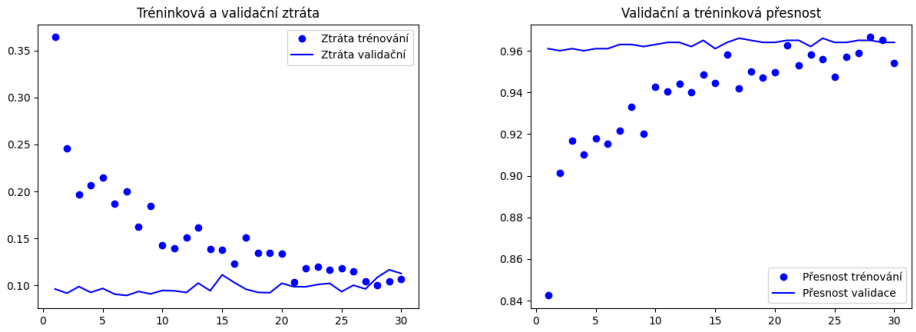
Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
Xception ladění	0.1065	0.9540	0.1127	0.9640	0.9630	0.1105

Tabulka 17: Tabulka výkonu jemného ladění Xception

Ze srovnání výkonu všech použitých technik je zřejmé, že se jedná o velice dobré výsledky, validační ztráta se blíží 0.1, kvalita tohoto modelu je téměř dvakrát vyšší, než tomu bylo u předchozích architektur.

Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
Xception extr.	1.6504e-05	1.0000	0.1946	0.9640	0.9660	0.2369
Xception extr. rozš.	0.1023	0.9605	0.1048	0.9650	0.9710	0.1236
Xception ladění	0.1065	0.9540	0.1127	0.9640	0.9630	0.1105

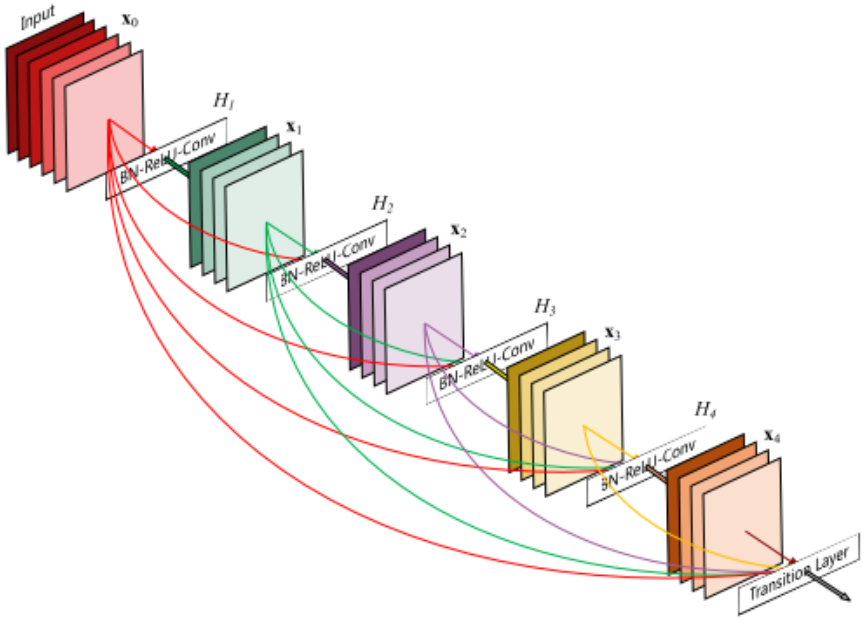
Tabulka 18: Tabulka srovnání výkonu Xception



Obrázek 38: Výsledky Xception jemné doladění, Zdroj: Vlastní

4.4.5 Densenet121

Architektura sítě Densenet vychází z myšlenky propojení všech vrstev se všemi dalšími, vstupy vrstvy jsou zřetězením všech předchozích vrstev a výstupy příznakových map jsou použity jako vstupy dalších vrstev.[48] Odtud její jméno, Densely Connected Convolutional Network, neboli hustě propojená konvoluční síť. Je to jeden ze způsobů jakým se předchází problému mizejicího gradientu. [49]



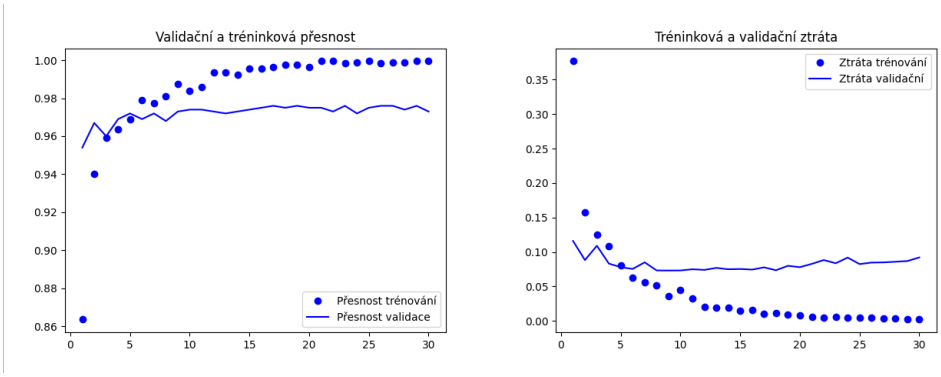
Obrázek 39: Znázornění Densenet, Zdroj: https://amaarora.github.io/images/densenet.png

V případě extrakce příznaků Densenet121 po 30 epochách trénování dostáváme tyto hodnoty:

Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
Densenet121 extr.	0.1773	0.9995	0.0920	0.9730	0.9650	0.0863

Tabulka 19: Tabulka srovnání výkonu Densenet121

Při době tréninku 35 vteřin můžeme vidět průběh tréninku na obrázku č. 40



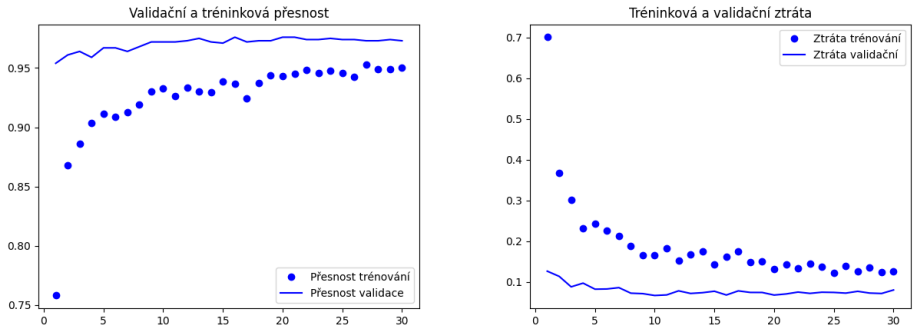
Obrázek 40: Výsledky Densenet121 extrakce příznaků, Zdroj: Vlastní

Extrakce příznaků s rozšířením dat třídou Imagedatagenerator, při užití Densenet121 po 30 epochách trénování, s délkou tréninku 7 minut, přináší tyto hodnoty:

Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
Densenet121 extr. rozš.	0.1254	0.9505	0.0772	0.9730	0.9730	0.0693

Tabulka 20: Tabulka extrakce příznaků s rozšířením Densenet121

Výsledky techniky jemného ladění sítě Densenet121 po 30 epochách tréninku v délce 13,5 minut, lze vidět na obrázku č. 42.



Obrázek 41: Výsledky Densenet121 extr. příz. s rozšířením, Zdroj: Vlastní

Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
Densenet121 ladění	0.0325	0.9875	0.0690	0.9800	0.9790	0.0803

Tabulka 21: Tabulka jemného ladění Densenet121

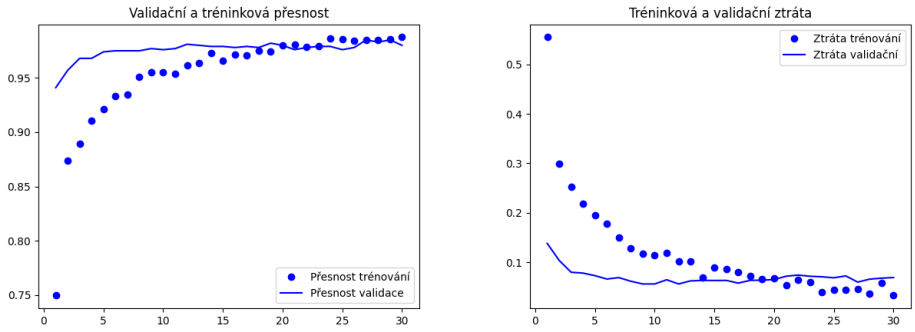
Arch. a technika	Tr-Z	Tr-Př	Val-Z	Val-P	Te-P	Te-Z
Densenet121 extr.	0.1773	0.9995	0.0920	0.9730	0.9650	0.0863
Densenet121 extr. rozš.	0.1254	0.9505	0.0772	0.9730	0.9730	0.0693
Densenet121 ladění	0.0325	0.9875	0.0690	0.9800	0.9790	0.0803

Tabulka 22: Tabulka srovnání výkonu Densenet121

5 Výsledky a diskuse

Předem je třeba říci, že srovnávání výsledků nově naučené sítě s použitím transfer learning, nelze tak docela ohodnotit a označit jedno za lepší než druhé. Při trénování sítě od začátku bylo zapotřebí větší znalosti problematiky a daleko více experimentování s parametry jako je velikost plně propojených vrstev, dále počet vrstev v celém modelu, posouvání velikosti kroku, tak aby bylo dosaženo nejlepšího výsledku. Někdy model ani při sebevětším ladění nedosáhne přesnosti nad 70%. Naproti tomu transfer learning obsahuje velmi málo možností nastavení ve srovnání s modelem trénovaným od nuly. V případě, že bychom chtěli vylepšovat výsledky některé z technik transfer learning, nemusíme uspět a tréninkem na malém datovém souboru můžeme paradoxně celkové výsledky zhoršit. Všechny použité techniky v této práci, extrakce příznaků, extrakce příznaků s rozšířením, a jemné ladění několika vrchních vrstev tedy fine- tuning mají svoje místo při praktickém používání technologie konvolučních sítí.

Obecně lze říci, že trénování sítě od nuly má největší potenciál v případě příhodných



Obrázek 42: Výsledky Densenet121 finetuning, Zdroj: Vlastní

podmínek a opravdové nutnosti získat ty nejlepší výsledky, s tím, že problém budeme řešit opakovaně a máme dostatek času připravit řešení na míru konkrétnímu problému. V opačném případě při řešení jedné úzce definované nebo specializované úlohy, pro příklad hledání jedné značky automobilu mezi ostatními, lze velmi dobře a rychle použít již naučenou síť a pomocí již známých technik dosáhnout výsledků mnohem rychleji při stejné nebo obdobné kvalitě.

Při práci na malém datovém souboru je třeba používat prostředky proti přeučení. Jak bylo zjištěno, rozdíl v použití nebo nepoužití vrstvy Dropout je obrovský, její použití zlepší kvalitu modelu, tím že omezí přeučení o polovinu nebo více. Naproti tomu tyto techniky nepomohou pokud je špatně zvolena kapacita modelu, tedy velikost plně propojených vrstev. Dalším poznatkem je, že na datových souborech o velikosti 2000 a 22000 obrázků, lze dosáhnout velice různých výsledků. Zdá se, že s 22000 trénovacích vzorků lze natrénovat velmi dobrý a použitelný model. Problematiku rozsahu datových souborů by bylo velice zajímavé dále prozkoumávat, jelikož výsledky nemusí vždy odpovídat očekávání.

Je pak na každém, aby rozhodl zdali na velkém datovém souboru bude delší dobu trénovat svou architekturu nebo nejprve využije naučenou síť a v případě, že ho výsledky neuspokojí, teprve poté začne trénovat síť od začátku. Nakonec to bude stejně otázka předchozí nabyté a aplikované zkušenosti.

Nyní ke konkrétním výsledkům, zobrazeným v tabulce 23. Jak je z přehledu patrné, transfer learning i při použití nejméně náročné techniky extrakce příznaků dosahuje velice dobrých výsledků. Za nejdůležitější kritérium považujme validační ztrátu. Kvalitativně nejlepším modelem je poté Densenet121 a hned za ním Xception s validační ztrátou blížící se nule a testovací přesností až 98%. Pokud však vezmeme v úvahu technologické rozdíly a pokročilou architekturu Densenetu121, nevede si mnohem starší VGG16 vůbec špatně.

Je ovšem zřetelně vidět, že na malých datových souborech je technika transfer learning velice efektivní, když porovnáme výsledky sekvenčního modelu s jakýmkoli jiným naučeným modelem, který byl otestován, dosahuje rozdíl až 20%.

Pro přehlednost závěrečný přehled všech dosažených výsledků:

Arch. a technika	**Tr-Z**	**Tr-Př**	**Val-Z**	**Val-P**	**Te-P**	**Te-Z**
VGG16 extr.	0.0920	0.9705	0.2397	0.9010	0.8870	0.3307
VGG16 extr. rozš.	0.2715	0.8900	0.2324	0.9090	0.8940	0.2449
VGG16 ladění	0.0637	0.9710	0.2305	0.9290	0.9350	0.2814
VGG19 extr.	0.1773	0.9399	0.2755	0.8830	0.8720	0.3022
VGG19 extr.rozš.	0.3668	0.8183	0.2914	0.8850	0.8570	0.3335
VGG19 ladění	0.0717	0.9740	0.3229	0.9290	0.9260	0.4393
InceptionV3 extr.	2.1942e-04	1.0000	0.2520	0.9660	0.9660	0.2429
InceptionV3 extr. rozš.	0.1624	0.9465	0.1032	0.9720	0.9640	0.1182
InceptionV3 ladění	0.0781	0.9755	0.2692	0.9440	0.9550	0.2078
Xception extr.	1.6504e-05	1.0000	0.1946	0.9640	0.9660	0.2369
Xception extr. rozš.	0.1023	0.9605	0.1048	0.9650	0.9710	0.1236
Xception ladění	0.1065	0.9540	0.1127	0.9640	0.9630	0.1105
Densenet121 extr.	0.1773	0.9995	0.0920	0.9730	0.9650	0.0863
Densenet121 extr. rozš.	0.1254	0.9505	0.0772	0.9730	0.9730	0.0693
Densenet121 ladění	0.0325	0.9875	0.0690	0.9800	0.9790	0.0803
Sekv. model, velký dataset	0.1804	0.9283	0.1972	0.9156	0.9250	0.1806
Sekv. model, malý dataset	0.4814	0.7708	0.4385	0.8050	0.4780	0.7870

Tabulka 23: Tabulka závěrečného přehledu výkonu použitých architektur

6 Závěr

Hlavní myšlenkou této práce a otázkou, na kterou bylo třeba odpovědět, je prozkoumání výhod a nevýhod transfer learning a také toho, jak je použitelné v praktických úlohách strojového učení při rozpoznávání objektů. V tomto případě odlišení kočky od psa, na obrazovém materiálu různé kvality a v malém množství, které není dostatečné pro trénink neuronové konvoluční sítě. Přičemž byly připraveny další překážky, které dosažení kvalitních výsledků ztížily. Nebyl k dispozici výkonný hardware a datový soubor byl dodatečně zmenšen, tak aby více odpovídal reálnému použití této technologie.

Cílem této práce bylo prověření opakované využitelnosti konvolučních sítí a zjištění výsledků, kterých dosahují. Po nastudování problematiky a přípravě pracovního prostředí, se tohoto cíle podařilo dosáhnout. Nebylo to bez občasných problémů, vytvoření kvalitního modelu konvoluční sítě není exaktní věda a lze se setkat s občasnými překvapivými výsledky, kdy jeden malý detail rozhoduje o kvalitě modelu. Hranice gradientního sestupu je někdy překvapivě tenká.

Pro mě osobně mělo toto téma veliký přínos a jen samotný úvod do této problematiky bude mít velké uplatnění v mém pracovním životě. Problematiku "vidění počítačů"považuji za fascinující s velkým potenciálem do budoucna. Ten se v poslední době rozvíjí překotným tempem. Je pravděpodobné že dojde i na další období stagnace, jako se již v historii několikrát ukázalo. Dosažené úspěchy již ale nelze ignorovat. Vzhledem k tomu, že problematika rozpoznávání statických obrazových dat, je již na pomyslném vrcholu a pravděpodobně tento stav nějakou dobu přetrvá. Je velice pravděpodobné, že dojde k dalšímu mezníku v této technologii, ale bude to nějakou dobu ještě trvat.

Pokračování této práce by mohlo řešit již skutečný problém, rozpoznávání fotografií kočky domácí od psa domácího by bylo užitečné leda pro myš nebo jiného hlodavce obývajícího nějaké lidské obydlí. Zajímavým problémem by mohla být klasifikace obrazového materiálu, zdali obsahuje obraz zvířete podléhající ochraně. Nebo posun k rekurentním sítím a využití jejich paměti k rozpoznávání aktivity ve videosouborech. Možnosti jsou v této disciplíně omezeny jen naší vlastní představivostí a výkonem grafické karty.

Seznam použité literatury

[1] CHOLLET, François: Deep learning v jazyku Python: knihovny Keras, Tensorflow. Praha: Grada Publishing, 2019. Knihovna programátora, ISBN 978-80-247-3100-1.

[2] Overview of the GNU System [online], [cit. 2019-12-19] Dostupné z WWW: http://www.gnu.org/gnu/gnu-history.html

[3] Comparison between conventional computers and neural networks. Standford Computer Science [online], [cit.2019-09-20].2019 © Stanford University. Dostupné z WWW :https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Comparison/comparison.html

[4] Machine learning explained. Big Data Made Simple [online], [cit.2020-01-19] Crayon Data©2019, Dostupné z WWW: https://bigdata-madesimple.com/machine-learning-explained-understanding-supervised-unsupervised-and-reinforcement-learning/

[5] Neural Networks: History [online], [cit. 2020-03-21] Dostupné z WWW: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html

[6] Donald Olding Hebb by Sarah Ferguson [online], [cit. 2020-03-10] Dostupné z WWW: https://can-acn.org/donald-olding-hebb/

[7] ADALINE [online], [cit. 2019-03-20], Dostupné z WWW: https://en.wikipedia.org/wiki/ADALINE

[8] MADALINE [online], [cit. 2019-05-30], Dostupné z WWW: https://en.wikipedia.org/wiki/ADALINE#MADALINE

[9] What is a Perceptron? [online], [cit. 2019-03-20], Dostupné z WWW: https://deepai.org/machine-learning-glossary-and-terms/perceptron

[10] A Brief History of Deep Learning By Keith D. Foote [online], [cit. 2019-03-20], Dostupné z WWW: https://www.dataversity.net/brief-history-deep-learning/

[11] Historie neuronových sítí 2 [online], [cit. 2019-03-20], Dostupné z WWW: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history2.html

[12] Historie neuronových sítí 1 [online], [cit. 2020-03-11], Dostupné z WWW: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/History/history1.html

[13] What is Perceptron: A Beginners Tutorial for Perceptron [online], [cit. 2019-03-20], Dostupné z WWW: https://www.simplilearn.com/what-is-perceptron-tutorial

[14] What the Hell is Perceptron? [online], [cit. 2020-08-20], Dostupné z WWW: https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53

[15] Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position [online], [cit. 2020-12-20], Dostupné z WWW: https://www.rctn.org/bruno/public/papers/Fukushima1980.pdf

[16] FeiFei Li [online], [cit. 2020-02-04], Dostupné z WWW: https://en.wikipedia.org/wiki/FeiFei_Li

[17] ImageNet is launched [online], [cit. 2020-05-25], Dostupné z WWW: https://machinelearningknowledge.ai/timeline/imagenet-is-launched/

[18] Timeline of machine learning, [online], [cit. 2019-03-20], Dostupné z WWW: https://en.wikipedia.org/wikiTimeline_of_machine_learning

[19] Google's AlphaGo Defeats Human Master of Ancient Game [online], [cit. 2019-03-20], Dostupné z WWW: https://www.technewsworld.com/story/83050.html

[20] Top 5 AI Achievements of 2020, M Umer Mirza publikováno: 26 Nov 2020 [online], [cit. 2020-12-12], Dostupné z WWW: https://thinkml.ai/top-5-ai-achievements-of-2020/

[21] Facial Recognition [online], [cit. 2019-03-20], Dostupné z WWW: https://www.interpol.int/How-we-work/Forensics/Facial-Recognition

[22] Facial recognition: Do you really control how your face is being used? [online], [cit. 2020-07-11], Dostupné z WWW: https://eu.usatoday.com/story/tech/2019/11/19/police-technology-and-surveillance-politics-of-facial-recognition/4203720002/

[23] Here's where the US government is using facial recognition technology to surveil Americans [online], [cit. 2020-02-02], Dostupné z WWW:

https://www.vox.com/recode/2019/7/18/20698307/facial-recognition-technology-us-government-fight-for-the-future

[24] To Neural Networks and Beyond! Neural Networks and Consciousness [online], [cit. 2019-09-20], Dostupné z WWW: https://cs.stanford.edu/people/eroberts/courses/soco/projects/neural-networks/Future/index.html

[25] Stephen Hawking warns artificial intelligence could end mankind, Rory Cellan-Jones Technology correspondent, publikováno: 2. prosince 2014 [online], [cit. 2020-07-21] Dostupné z WWW: https://www.bbc.com/news/technology-30290540

[26] Introduction to Tensors [online], [cit. 2020-03-13], Dostupné z WWW: https://www.tensorflow.org/guide/tensor

[27] Keras About [online], [cit. 2020-08-14.], Dostupné z WWW: https://keras.io/about/

[28] Theano developement team [online], [cit. 2019-03-20], Dostupné z WWW: http://arxiv.org/pdf/1605.02688.pdf

[29] The Microsoft Cognitive Toolkit [online], [cit. 2019-03-25], Dostupné z WWW: https://docs.microsoft.com/en-us/cognitive-toolkit/

[30] What Is CUDA? [online], [cit. 2019-03-25], Mark Ebersole, Dostupné z WWW: https://blogs.nvidia.com/blog/2012/09/10/what-is-cuda-2/

[31] TensorFlow 2 - CPU vs GPU Performance Comparison, [online], [cit. 2020-05-25], Dostupné z WWW: https://datamadness.github.io/TensorFlow2-CPU-vs-GPU

[32] What is Transfer Learning?, [online], [cit. 2020-03-25], Dostupné z WWW: https://towardsdatascience.com/what-is-transfer-learning-8b1a0fa42b4

[33] Keras vs Tensorflow: Must Know Differences!, [online], [cit. 2020-04-27] Dostupné z WWW: https://www.guru99.com/tensorflow-vs-keras.html

[34] Convolutional Neural Network and Regularization Techniques with TensorFlow and Keras, [online], [cit. 2020-07-21] Dostupné z WWW: https://medium.com/intelligentmachines/convolutional-neural-network-and-regularization-techniques-with-tensorflow-and-keras-5a09e6e65dc7

[35] Very Deep Convolutional Networks for Large-Scale Image Recognition [online], [cit. 2020-08-22], (PDF) Dostupné z: https://arxiv.org/pdf/1409.1556.pdf

[36] VGG-16 — CNN model [online], [cit. 2020-01-26] Dostupné z WWW: https://www.geeksforgeeks.org/vgg-16-cnn-model/

[37] Image Classification Architectures review, Prakash Jay [online], [cit. 2020-08-01], Dostupné z WWW: https://medium.com/@14prakash/image-classification-architectures-review-d8b95075998f

[38] Vishal Mishra, CNN Architecture: A Brief Introduction to Inception Network [online], [cit. 2020-11-26] Dostupné z WWW: https://medium.com/swlh/cnn-architecture-a-brief-introduction-to-inception-network-c94396157fba

[39] Christian Szegedy, Wei Liu, Yangqing Jia a kolektiv, Going deeper with convolutions, [online], (PDF), [cit. 2019-03-20], Dostupné z WWW: https://arxiv.org/pdf/1409.4842.pdf

[40] Sebastian Ruder, 19.1.2016, An overview of gradient descent optimization algorithms, [online], [cit. 2020-09-30], Dostupné z WWW: https://ruder.io/optimizing-gradient-descent/

[41] What is batch size, steps, iteration, and epoch in the neural network?, [online], [cit. 2020-06-21] Dostupné z WWW: https://androidkt.com/batch-size-step-iteration-epoch-neural-network/

[42] Pyplot tutorial, [online], [cit. 2020-03-16], Dostupné z WWW: https://matplotlib.org/stable/api/_as_gen/matplotlib.pyplot.plot.html#matplotlib.pyplot.plot

[43] Virtual Environments and Packages, [online], [cit. 2020-09-02], Dostupné z WWW: https://docs.python.org/3/tutorial/venv.html

[44] Create virtual environments for python with conda [online], [cit. 2020-10-20] Dostupné z WWW: https://uoa-eresearch.github.io/eresearch-cookbook/recipe/2014/11/20/conda/

[45] TensorFlow Core v2.6.0 dokumentace [online], [cit. 2021-10-21], Dostupné z WWW: https://www.tensorflow.org/api_docs/python/tf/keras/utils/image_dataset_from_directory

[46] XCeption Model and Depthwise Separable Convolutions, [online], [cit. 2020-10-21], Dostupné z WWW: https://maelfabien.github.io/deeplearning/xception/#how-does-xception-work

[47] Francois Chollet, Xception: Deep Learning with Depthwise Separable Convolutions, [online], (PDF), [cit. 2020-10-21] (PDF) Dostupné z WWW: https://openaccess.thecvf.com/content_cvpr_2017/papers/Chollet_Xception_Deep_Learning_CVPR_2017

[48] DenseNet Architecture Explained with PyTorch Implementation from TorchVision, [online], [cit. 2020-10-21] Dostupné z: https://amaarora.github.io/2020/08/02/densenets.html

[49] Gao Huang, Zhuang Liu, Laurens van der Maaten a kolektiv, Densely Connected Convolutional Networks, [online],(PDF), [cit. 2020-05-01] Dostupné z WWW: https://arxiv.org/abs/1608.06993

[50] Michael Kan a kolektiv, Inside the GPU Shortage: Why You Still Can't Buy a Graphics Card, [online], [cit. 2020-06-21], Dostupné z: https://www.pcmag.com/news/inside-the-gpu-shortage-why-you-still-cant-buy-a-graphics-card

[51] Nik Paushkin, Running Jupyter Notebook in cloud with GPU in 30 seconds, [online], [cit. 2020-03-25] Dostupné z: https://puzl.ee/blog/running-jupyter-notebook-in-cloud-with-gpu-in-30-seconds

7 Přílohy

Zdrojové kódy použitých neuronových konvolučních sítí.

7.1 VGG16

```
from keras.applications import VGG16

conv_base = VGG16(weights='imagenet', include_top=False, input_shape=(150, 150, 3))

import os
import numpy as np
from keras_preprocessing.image import ImageDataGenerator

base_dir = '/home/peky/PycharmProjects/CNN/DataCNN/catdogsmall/'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1. / 255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=(sample_count))
    generator = datagen.flow_from_directory(
    directory,
    target_size=(150, 150),
    batch_size=batch_size,
    class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
    features_batch = conv_base.predict(inputs_batch)
    features[i * batch_size : (i + 1) * batch_size] = features_batch
    labels[i * batch_size : (i + 1) * batch_size] = labels_batch
    i += 1
    if i * batch_size ¿= sample_count:
    break
    return features, labels

train_features, train_labels = extract_features(train_dir, 2000)
```

```
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)

train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
test_features = np.reshape(test_features, (1000, 4 * 4 * 512))

from keras import models
from keras import layers
from keras import optimizers

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim= 4 * 4 * 512))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(optimizer=optimizers.RMSprop(lr=2e-5),
loss='binary_crossentropy',
metrics=['acc'])
history = model.fit(train_features, train_labels,
epochs=30,
batch_size=20,
validation_data=(validation_features, validation_labels))

import matplotlib.pyplot as plt

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='training acc')
plt.plot(epochs, val_acc, 'b', label='validation acc')
plt.title('training and valid accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='training loss')
```

```
plt.plot(epochs, val_acc, 'b', label='validation loss')
plt.title('training and valid loss')
plt.legend()

plt.show()
```

7.2 VGG19

```
from keras.applications import VGG19
import os
import numpy as np
import matplotlib.pyplot as plt
from keras import models
from keras import layers
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers

conv_base = VGG19(weights='imagenet',
include_top=False,
input_shape=(150, 150, 3))

conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
    set_trainable = True
    if set_trainable:
    layer.trainable = True
    else:
    layer.trainable = False

conv_base.summary()

base_dir = '/home/peky/PycharmProjects/CNN/DataCNN/catdogsmall/'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1. / 255)
```

```
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 512))
    labels = np.zeros(shape=sample_count)
    generator = datagen.flow_from_directory(
    directory,
    target_size=(150, 150),
    batch_size=batch_size,
    class_mode='binary')
    i = 0
    for inputs_batch, labels_batch in generator:
    features_batch = conv_base.predict(inputs_batch)
    features[i * batch_size: (i + 1) * batch_size] = features_batch
    labels[i * batch_size: (i + 1) * batch_size] = labels_batch
    i += 1
    if i * batch_size ¿= sample_count:
    break
    return features, labels

train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)

train_features = np.reshape(train_features, (2000, 4 * 4 * 512))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 512))
test_features = np.reshape(test_features, (1000, 4 * 4 * 512))

model = models.Sequential()
model.add(conv_base)
model.add(layers.Flatten())
model.add(layers.Dense(256, activation='relu'))
model.add(layers.Dense(1, activation='sigmoid'))

model.summary()

train_datagen = ImageDataGenerator(rescale=1. / 255,
    rotation_range=40,
    width_shift_range=0.2,
    height_shift_range=0.2,
    shear_range=0.2,
```

```
    zoom_range=0.2,
    horizontal_flip=True,
    fill_mode='nearest')

test_datagen = ImageDataGenerator(rescale=1. / 255)

train_generator = train_datagen.flow_from_directory(train_dir,
target_size=(150, 150),
batch_size=20,
class_mode='binary')

validation_generator = test_datagen.flow_from_directory(validation_dir, target_size=(150, 150),
batch_size=20,
class_mode='binary')

test_generator = test_datagen.flow_from_directory(
test_dir,
target_size=(150, 150),
batch_size=20,
class_mode='binary')

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

history = model.fit_generator(train_generator, steps_per_epoch=100, epochs=30,
validation_data=validation_generator,
validation_steps=50)

test_loss, test_acc = model.evaluate(test_generator, steps=50)
print('test acc', test_acc)

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Přesnost trénování')
plt.plot(epochs, val_acc, 'b', label='Přesnost validace')
plt.title('Validační a tréninková přesnost')
```

```
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Ztráta trénování')
plt.plot(epochs, val_loss, 'b', label='Ztráta validační')
plt.title('Tréninková a validační ztráta')
plt.legend()

plt.show()
```

7.3 InceptionV3

```
from keras.applications import InceptionV3
import os
import numpy as np
import matplotlib.pyplot as plt
from keras import models
from keras import layers
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers

conv_base = InceptionV3(weights='imagenet',
include_top=False,
input_shape=(150, 150, 3))

conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
    if layer.name == 'conv2d_93' or 'conv2d_85':
    set_trainable = True
    if set_trainable:
    layer.trainable = True
    else:
    layer.trainable = False

conv_base.summary()

base_dir = '/home/peky/PycharmProjects/CNN/DataCNN/catdogsmall/'
```

```
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1. / 255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 3, 3, 2048))
    labels = np.zeros(shape=sample_count)
    generator = datagen.flow_from_directory(
    directory,
    target_size=(150, 150),
    batch_size=batch_size,
    class_mode='binary')
i = 0
for inputs_batch, labels_batch in generator:
    features_batch = conv_base.predict(inputs_batch)
    features[i * batch_size: (i + 1) * batch_size] = features_batch
    labels[i * batch_size: (i + 1) * batch_size] = labels_batch
    i += 1
    if i * batch_size ¿= sample_count:
    break
    return features, labels

train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)

train_features = np.reshape(train_features, (2000, 3 * 3 * 2048))
validation_features = np.reshape(validation_features, (1000, 3 * 3 * 2048))
test_features = np.reshape(test_features, (1000, 3 * 3 * 2048))

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=3 * 3 * 2048))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
optimizer=optimizers.RMSprop(lr=2e-5), metrics=['acc'])
```

```
history = model.fit(train_features, train_labels, batch_size=20, epochs=50,
validation_data=(validation_features, validation_labels))

test_loss, test_acc = model.evaluate(test_generator, steps=50)
print('test acc', test_acc)

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='training acc')
plt.plot(epochs, val_acc, 'b', label='validation acc')
plt.title('training and valid accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='training loss')
plt.plot(epochs, val_acc, 'b', label='validation loss')
plt.title('training and valid loss')
plt.legend()

plt.show()
```

7.4 Xception

```
from keras.applications import Xception
import os
import numpy as np
import matplotlib.pyplot as plt
from keras import models
from keras import layers
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers

conv_base = Xception(weights='imagenet',
include_top=False,
```

```
input_shape=(150, 150, 3))

conv_base.summary()

set_trainable = False

for layer in conv_base.layers:
    if layer.name == 'block5_conv1':
    set_trainable = True
    if set_trainable:
    layer.trainable = True
    else:
    layer.trainable = False

base_dir = '/home/peky/PycharmProjects/CNN/DataCNN/catdogsmall/'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1. / 255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 5, 5, 2048))
    labels = np.zeros(shape=sample_count)
    generator = datagen.flow_from_directory(
    directory,
    target_size=(150, 150),
    batch_size=batch_size,
    class_mode='binary')
i = 0
for inputs_batch, labels_batch in generator:
    features_batch = conv_base.predict(inputs_batch)
    features[i * batch_size: (i + 1) * batch_size] = features_batch
    labels[i * batch_size: (i + 1) * batch_size] = labels_batch
    i += 1
    if i * batch_size ¿= sample_count:
    break
return features, labels

train_features, train_labels = extract_features(train_dir, 2000)
```

```
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)

train_features = np.reshape(train_features, (2000, 5 * 5 * 2048))
validation_features = np.reshape(validation_features, (1000, 5 * 5 * 2048))
test_features = np.reshape(test_features, (1000, 5 * 5 * 2048))

model = models.Sequential()
model.add(layers.Dense(256, activation='relu', input_dim=5 * 5 * 2048))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

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

history = model.fit(train_features, train_labels, batch_size=20, epochs=30,
validation_data=(validation_features, validation_labels))

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='training acc')
plt.plot(epochs, val_acc, 'b', label='validation acc')
plt.title('training and valid accuracy')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='training loss')
plt.plot(epochs, val_acc, 'b', label='validation loss')
plt.title('training and valid loss')
plt.legend()

plt.show()
```

7.5 Densenet121

```
from keras.applications import DenseNet121
import os
import numpy as np
import matplotlib.pyplot as plt
from keras import models
from keras.layers import Flatten
from keras import layers
from keras.preprocessing.image import ImageDataGenerator
from keras import optimizers

conv_base = DenseNet121(weights='imagenet',
include_top=False,
input_shape=(150, 150, 3))

conv_base.trainable = True

set_trainable = False
for layer in conv_base.layers:
if layer.name == 'conv5_block16_1_conv' or 'conv5_block15_1_conv':
set_trainable = True
if set_trainable:
layer.trainable = True
else:
layer.trainable = False

conv_base.summary()

base_dir = '/home/peky/PycharmProjects/CNN/DataCNN/catdogsmall/'
train_dir = os.path.join(base_dir, 'train')
validation_dir = os.path.join(base_dir, 'validation')
test_dir = os.path.join(base_dir, 'test')

datagen = ImageDataGenerator(rescale=1. / 255)
batch_size = 20

def extract_features(directory, sample_count):
    features = np.zeros(shape=(sample_count, 4, 4, 1024))
    labels = np.zeros(shape=sample_count)
    generator = datagen.flow_from_directory(
    directory,
```

```
    target_size=(150, 150),
    batch_size=batch_size,
    class_mode='binary')
i = 0
for inputs_batch, labels_batch in generator:
    features_batch = conv_base.predict(inputs_batch)
    features[i * batch_size: (i + 1) * batch_size] = features_batch
    labels[i * batch_size: (i + 1) * batch_size] = labels_batch
    i += 1
    if i * batch_size ¿= sample_count:
    break
return features, labels

train_features, train_labels = extract_features(train_dir, 2000)
validation_features, validation_labels = extract_features(validation_dir, 1000)
test_features, test_labels = extract_features(test_dir, 1000)

train_features = np.reshape(train_features, (2000, 4 * 4 * 1024))
validation_features = np.reshape(validation_features, (1000, 4 * 4 * 1024))
test_features = np.reshape(test_features, (1000, 4 * 4 * 1024))

train_datagen = ImageDataGenerator(rescale=1. / 255,
rotation_range=40,
width_shift_range=0.2,
height_shift_range=0.2,
shear_range=0.2,
zoom_range=0.2,
horizontal_flip=True,
fill_mode='nearest')

test_datagen = ImageDataGenerator(rescale=1. / 255)

train_generator = train_datagen.flow_from_directory(train_dir,
target_size=(150, 150),
batch_size=20,
class_mode='binary')

validation_generator = test_datagen.flow_from_directory(validation_dir, target_size=(150, 150),
batch_size=20,
class_mode='binary')
```

```
test_generator = test_datagen.flow_from_directory(
test_dir,
target_size=(150, 150),
batch_size=20,
class_mode='binary')

model = models.Sequential()
model.add(conv_base)
model.add(Flatten())
model.add(layers.Dense(256, activation='relu', input_dim=4 * 4 * 1024))
model.add(layers.Dropout(0.5))
model.add(layers.Dense(1, activation='sigmoid'))

model.summary()

model.compile(loss='binary_crossentropy',
optimizer=optimizers.RMSprop(lr=1e-5), metrics=['acc'])

history = model.fit(train_generator, steps_per_epoch=100,
epochs=100,
validation_data=validation_generator,
validation_steps=50)

test_loss, test_acc = model.evaluate(test_generator, steps=50)
print('test acc', test_acc)

acc = history.history['acc']
val_acc = history.history['val_acc']
loss = history.history['loss']
val_loss = history.history['val_loss']

epochs = range(1, len(acc) + 1)

plt.plot(epochs, acc, 'bo', label='Přesnost trénování')
plt.plot(epochs, val_acc, 'b', label='Přesnost validace')
plt.title('Validační a tréninková přesnost')
plt.legend()

plt.figure()

plt.plot(epochs, loss, 'bo', label='Ztráta trénování')
```

```
plt.plot(epochs, val_loss, 'b', label='Ztráta validační')
plt.title('Tréninková a validační ztráta')
plt.legend()

plt.show()
```