Vysoká škola ekonomická v Praze
Fakulta informatiky a statistiky

Použití konvoluční neuronové sítě pro klasifikační úlohu

BAKALÁŘSKÁ PRÁCE

Studijní program: Aplikovaná informatika
Studijní obor: Aplikovaná informatika

Autor: Martin Petráň
Vedoucí práce: prof. Ing. Petr Berka, CSc.

Praha, Květen 2019

Prohlášení

Prohlašuji, že jsem bakalářskou práci *Použití konvoluční neuronové sítě pro klasifikační úlohu* vypracoval samostatně za použití v práci uvedených pramenů a literatury.

V Praze dne 6. května 2019 .....

Podpis studenta

Poděkování

Děkuji panu prof. Ing. Petru Berkovi, CSc. trpělivost a cenné rady při vedení této práce.

Abstrakt

Cílem této bakalářské práce je vytvořit a optimalizovat model konvoluční neuronové sítě, který bude schopen klasifikovat vybrané druhy motýlů s celkovou správností $\geq 95\%$ na testovací množině dat.

V teoretické části jsou nejdříve vymezeny základní pojmy z oblasti strojového učení a umělé inteligence. Dále je představena použitá metoda přenosu učení a vybraná aplikační oblast. Praktická část se věnuje samotné tvorbě modelu konvoluční neuronové sítě s pomocí Python knihovny Keras a následné optimalizaci modelu. V této části jsou také popsány problémy, které bylo nutno překonat, a způsob jejich řešení. Jednalo se zejména o neúčinnost a neefektivitu učícího algoritmu pro zvolená nastavení hyperparametr. V několika experimentech se také vyskytl problém přeučení. Na závěr je provedeno vyhodnocení modelu včetně analýzy špatně klasifikovaných případů.

Výsledný model dosáhl celkové správnosti 97,77%, ale přesto byly identifikovány určité nedostatky vyplývající z podstaty použité architektury, jako je například neschopnost modelu rozeznat více motýlů na vstupním obrázku.

Klíčová slova

počítačové vidění, konvoluční neuronová síť, klasifikace obrazu, přenos učení, Keras

Abstract

The aim of this bachelor thesis is to create and optimize a convolutional neural network model, which will be able to classify selected species of butterflies with the overall accuracy of $\geq 95\%$ on the test dataset.

In the theoretical part, the basic concepts of machine learning and artificial intelligence are defined. Next, the transfer learning method and the selected application area are presented. The practical part deals with the creation of the convolutional neural network model with the use of the Python library Keras and subsequent optimization of the model. This section also describes the problems that had to be overcome and the means used to do so. In particular, it was the ineffectiveness and inefficiency of the learning algorithm for selected hyperparameter settings. The problem of overfitting also occurred in several experiments. Finally, the model is evaluated including the analysis of misclassified cases.

The final model achieved overall accuracy of 97,77%, but some weaknesses were identified. Due to the nature of used architecture, the model is unable to recognize multiple butterflies in one input image.

Keywords

computer vision, convolutional neural network, image classification, transfer learning, Keras

Obsah

Úvod

Neuplyne měsíc, kdy by prestižními vědeckými magazíny neproběhla zpráva o novém objevu v oblasti strojového učení a umělé inteligence. Přestože základy těchto oborů informatiky byly položeny již v půlce minulého století, zažívají v posledních letech bezprecedentní růst a mění způsob, jakým pracujeme a jakým žijeme. A co víc, tyto změny probíhají stále rychleji a rychleji.

Praktické aplikace umělé inteligence jsou dnes již běžnou součástí našich životů: Umělá inteligence nám na základě rozpoznání obličeje odemkne telefon, vybere nejvhodnější cestu do práce, doporučí dárek pro partnerku, ale může nám také spravovat investiční portfolio. Je jen otázkou času, kdy budou stroje z rentgenových snímků diagnostikovat rakovinu, řídit prostředky osobní i veřejné přepravy, nebo pomáhat právníkům se psaním zákonů.

Strojové učení se pomalu ale jistě začíná prosazovat ve všech odvětvích lidské činnosti od výroby po lidské zdroje a komplexita úkolů, které lze s vysokou spolehlivostí automatizovat, se neustále zvyšuje. Společnost McKinsey Global Institute ve svém reportu uvádí, že do roku 2030 by mohlo vlivem automatizace pomocí strojového učení dojít ke zrušení 400 - 800 milionů pracovních pozic po celém světě (Manyika et al., 2017). Zároveň však předpokládá vytvoření 555 - 890 milionů nových pracovních míst. Stejně jako každá jiná průlomová technologie je i strojové učení pro současnou společnost velikou hrozbou, ale zároveň i velikou příležitostí.

Jedním z oborů, kde strojové učení slaví největší úspěch, je počítačové vidění. Historickým milníkem se stal rok 2012, kdy tým z Torontské univerzity jako první použil technologii konvolučních neuronových sítí v soutěži *ImageNet Large Scale Visual Recognition Competition* (ILSVRC) a tuto soutěž s fenomenálním výsledkem vyhrál (Krizhevsky et al., 2012). Tímto okamžikem nastala na poli rozpoznání a klasifikace obrazu éra konvolučních neuronových sítí a v této oblasti se již prakticky nesetkáme s jinou technologií.

Konvolučním neuronovým sítím se věnuje i tato práce, jejíž tématem je použití konvoluční neuronové sítě pro klasifikační úlohu. O problematiku umělé inteligence se zajímám již delší dobu, ale teprve návštěva výstavy motýlů v tropickém skleníku Fata Morgana v Praze mě inspirovala ke zvolení tohoto tématu. Jedná se výbornou příležitost, jak mohu prohloubit své znalosti této problematiky a zároveň načerpat zkušenosti, které mohou sloužit jako odrazový můstek ke kariérnímu postupu, či ke tvorbě komerčně využitelného produktu.

Cílem této bakalářské práce je vytvořit a optimalizovat model konvoluční neuronové sítě, který bude schopen klasifikovat vybrané druhy motýlů s celkovou správností $\geq 95\%$. Model bude vytvořen metodou přenosu učení (transfer learning) v programovacím jazyce Python a to s pomocí softwarových knihoven Keras a TensorFlow. Optimalizace pak bude probíhat metodou experimentu.

Práce je strukturována do dvou hlavních částí, teoretické a praktické. V teoretické části

jsou nejdříve vymezeny základní pojmy z oblasti strojového učení a umělé inteligence, které poskytnou vhled do problematiky i neznalému čtenáři. Dále je představena použitá metoda přenosu učení a vybraná aplikační oblast.

Náplní praktické části je popis použitých technologií a tvorby modelu konvoluční neuronové sítě. Následně je popsán proces optimalizace vytvořeného modelu a problémy, které bylo nutno překonat. Na závěr je uvedeno vyhodnocení modelu, jehož součástí je i analýza špatně klasifikovaných případů. Ačkoliv praktická část obsahuje výpisy kódu, není pro čtenáře znalost programovacích jazyků nutná.

1. Strojové učení

Pojem strojové učení poprvé použil americký vědec Arthur Samuel z IBM v roce 1959. Strojové učení definuje jako obor, který dává počítačovým systémům schopnost „učit se“, aniž by k tomu byly explicitně programovány (Samuel, 1959).

Nejčastěji se setkáme s definicí počítačového vědce a profesora Toma Mitchella, který na Samuelovou definici navázal a dále ji rozvedl:

> *„Počítačový program se **učí** ze zkušenosti E s ohledem na třídu úloh T a výkonnostní metriku P, pokud platí, že výkon v úloze T, měřený pomocí P, se zlepšuje se zkušeností E.“* [1] (Mitchell, 1997)

Jako příklad může posloužit aplikace třídící nevyžádanou poštu. Samotné třídění je v tomto případě úloha T, příklady nevyžádané pošty jsou zkušeností E a procento správně klasifikovaných emailů je metrika P.

Strojové učení je multidisciplinární obor, který je často považován za podoblast umělé inteligence. Stephen Marsland na něj ve své knize poskytuje zajímavý náhled:

> *„Jedním z nejzajímavějších rysů strojového učení je to, že leží na hranici několika různých akademických disciplín, především počítačové vědy, statistiky, matematiky a inženýrství. Strojové učení je obvykle studováno jako součást umělé inteligence, což ho pevně zařazuje mezi počítačové vědy. Pochopení toho, proč tyto algoritmy fungují, vyžaduje určitou statistickou a matematickou sečtělost, která často u studentů počítačových věd chybí.“* [2] (Marsland, 2009)

Marsland zdůrazňuje multidisciplinární charakter tohoto oboru, jelikož strojové učení čerpá ze všech oblastí informačních věd. Zároveň ale zdůrazňuje i nebezpečí, které hrozí při náhledu na strojové učení pouze z jedné perspektivy. Zejména pak v případě, kdy se vyhýbáme vnitřním matematickým základům, na kterých je strojové učení postaveno. Stejně limitující je ale i pohled statistika, který se nezajímá o reálnou implementaci a nasazení algoritmů strojového učení.

[1] *„A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E.“*

[2] *„One of the most interesting features of machine learning is that it lies on the boundary of several different academic disciplines, principally computer science, statistics, mathematics, and engineering. Machine learning is usually studied as part of artificial intelligence, which puts it firmly into computer science. Understanding why these algorithms work requires a certain amount of statistical and mathematical sophistication that is often missing from computer science undergraduates.“*

1.1 Úloha *T*

Strojové učení nám umožňuje vyřešit úlohy, které by byly logickými a procedurálními metodami neřešitelné, nebo řešitelné jen velmi obtížně. Zatímco vytvořit s použitím umělé inteligence aplikaci pro rozpoznání obličeje je dnes běžnou praxí, naprogramovat stejnou aplikaci konvenčními postupy není reálné.

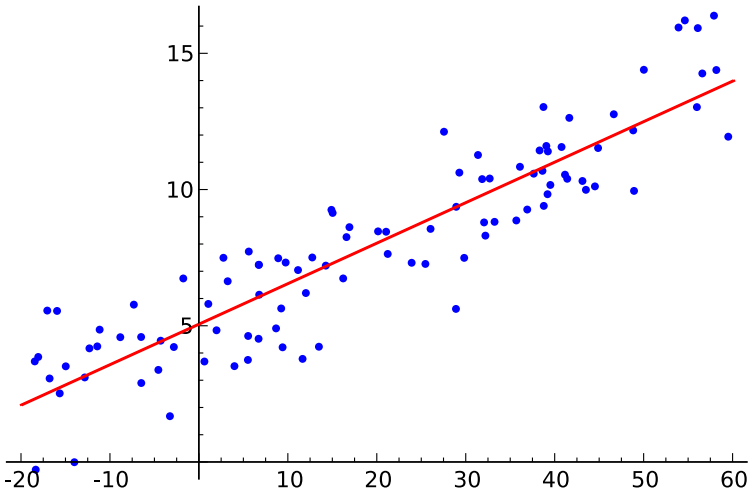
Jedním ze základních typů úloh je regrese. V případě regresní úlohy je úkolem počítačového systému vytvořit funkci $f : \mathbb{R}^n \rightarrow \mathbb{R}$, tedy funkci, která mapuje reálný vstup na reálný výstup. Tato funkce se obecně nazývá hypotézou a pro lineární regresi s jednou proměnnou má následující tvar:

$$h_\theta(x) = \theta_0 + \theta_1 x \tag{1.1}$$

Jelikož lineární regresi můžeme definovat jako proložení souboru bodů v grafu přímkou (viz obrázek 1.1), má i naše hypotéza $h(x)$ tvar přímky, kde x představuje vstupní data ve formě **vektoru příznaků** (feature vector). Příznaky si můžeme představit jako určité vlastnosti, které mají všechna vstupní data. Příznakem tedy může být např. výška, váha, rozměry, cena aj. Koeficienty dané rovnice se nazývají **váhy** a značí se řeckým písmenem θ.

Úkolem učícího algoritmu je nalézt takovou hodnotu vah θ, pro kterou existuje nejlepší hodnota metriky P (viz kapitola 1.3).

Typickým příkladem použití lineární regrese je prediktivní analýza (predikce vývoje poptávky, predikce ceny aktiva), kdy je na základě vstupních dat vytvořena učícím algoritmem funkce, kterou můžeme použít k aproximaci hodnot i pro příklady, které nejsou součástí vstupních dat.



Obrázek 1.1: Lineární regrese s jednou proměnnou

Mezi další významné typy úloh patří:

- **Klasifikace**: V tomto typu úlohy je úkolem počítačové systému určit, ke které z k tříd patří určitý vstup x.
- **Shlukování**: Cílem je rozdělit vstupní data do shluků (clusterů) na základě skrytého vzoru.
- **Strojový překlad**: V případě strojového překladu je vstupem pro učící algoritmus sekvence znaků jedné řeči a úkolem je převést tyto znaky do řeči jiné.
- **Transkripce**: V úlohách transkripce chceme po počítačovém systému, aby z relativně nestrukturovaného vstupu získal informace ve strukturované textové podobě. Příkladem může být čtení značek SPZ nebo domovních čísel z fotografií.
- **Detekce anomálií**: Vzhledem k tomu, že neuronové sítě jsou výjimečně dobré v rozpoznávání vzorů, mohou být naučeny k tomu, aby označily případy, které nějakým způsobem z daného vzoru vybočují. Detekce anomálií se využívá kupříkladu při kontrole finančních operací a transakcí.

1.2 Zkušenost *E*

V kontextu strojového učení rozumíme pojmem „zkušenost“ vstupní data, která algoritmus využívá v procesu učení. Dle typu zkušenosti rozlišujeme tři základní paradigmata: učení s učitelem, učení bez učitele a posilované učení (Burkov, 2019).

Učení s učitelem (supervised learning) je strategie, která zahrnuje učitele, který je „chytřejší“ než učený model. Vstupem je totiž kromě příznakových vektorů x i očekávaný výstup nazývaný anglickým termínem **ground truth**. Učící algoritmus má tedy k dispozici výstup, kterého by měl v ideálním případě dosáhnout. Proto se tento typ učení nazývá učení s učitelem.

Jedná se o nejrozšířenější způsobem učení a takřka všechny mediálně zajímavé aplikace strojového učení jsou postaveny právě na této strategii. I tato práce je psána především v kontextu učení s učitelem.

V případě **učení bez učitele** (unsupervised learning) nemáme k dispozici očekávané výstupy a vstupem je pouze kolekce příznakových vektorů x. Model vytvořený na základě těchto dat hledá skryté vzory a data dle těchto vzorů rozřazuje (Goodfellow et al., 2016).

Aktuálně se v této oblasti čeká na průlom, který by umožnil využívat učení bez učitele s podobným úspěchem jako učení s učitelem.

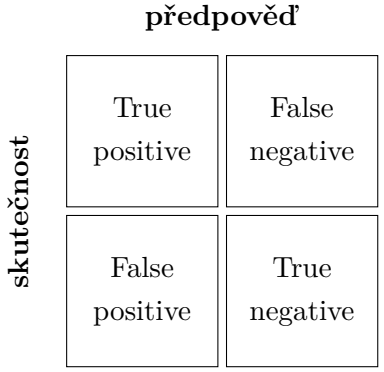
Posilované učení (reinforcement learning) je speciálním typem učení, kde počítačový systém existuje v určitém prostředí a je schopen stav tohoto prostředí vnímat jako vstup x. V jednotlivých stavech daného prostředí může systém vykonávat různé akce, přičemž každá akce je buď odměněna, nebo potrestána. Systém tedy dostává **zpětnou vazbu**, na základě které se učí (Burkov, 2019).

Cílem posilovaného algoritmu je naučit se určitému chování a pravidlům. Použijeme ho v případech, kdy je zapotřebí, aby systém činil komplexní rozhodování. Např. již v roce 2006 bylo posilované učení využito na Standfordově univerzitě k vytvoření samořídícího modelu helikoptéry (Ng et al., 2006).

1.3 Metrika *P*

Aby bylo možné vyhodnotit chování modelu vytvořeného učícím algoritmem, je nutné zavést měřitelnou metriku výkonnosti. Tato metrika P je zpravidla specifická pro konkrétní úlohu, nebo typ úloh T.

V případě klasifikace existuje několik základních metrik, které můžeme sledovat. Tyto metriky jsou správnost, přesnost, úplnost a F1-score. Vycházejí z matice záměn, což je kontingenční matice obsahující kombinace předpovězených a skutečných hodnot pro danou třídu (viz obrázek 1.2).



Obrázek 1.2: Matice záměn

Správnost modelu vyjadřuje, jakou část případů model klasifikoval správně. Zajímá nás tedy, kolik procent tvoří skutečně pozitivní (TP) a skutečně negativní (TN) případy na celkovém počtu.

$$\text{správnost (accuracy)} = \frac{TP + TN}{TP + FP + TN + FN} \tag{1.2}$$

Přesnost nám říká, kolik z případů označených jako pozitivní bylo **skutečně** pozitivních. Přesnost je vhodná metrika především v případech, kdy cena za falešně pozitivní klasifikaci je velmi vysoká. Opět se hodí příklad s filtrováním nevyžádané pošty; zde je logické požadovat co nejvyšší přesnost, jelikož falešně pozitivní případ znamená, že normální emailová zpráva je označena jako nevyžádaná pošta.

$$\text{přesnost (precision)} = \frac{TP}{TP + FP} \tag{1.3}$$

Úplnost modelu vyjadřuje, jak velkou část pozitivních případů model reálně odhalil. Příkladem, kdy bude vyžadována co nejvyšší úplnost, může být detekce podvodů nebo diagnóza nemocí. Zde by při nízké úplnosti mohlo dojít až k fatálním důsledkům pro pacienta, který

byl označen jako falešně negativní. Pokud by se jednalo o nakažlivou nemoc, tak cena za falešně negativní predikci je extrémně vysoká.

$$\text{úplnost (recall)} = \frac{TP}{TP + FN} \tag{1.4}$$

F1-score je harmonickým průměrem přesnosti a úplnosti, jedná se tedy o agregovanou metriku, kterou použijeme v případech, kdy chceme dosáhnout vyrovnaných výsledků přesnosti a úplnosti.

$$\text{F1-score} = 2\frac{\text{přesnost} * \text{úplnost}}{\text{přesnost} + \text{úplnost}} \tag{1.5}$$

Tyto metriky jsou vhodné pro interpretaci člověkem a pro vyhodnocení již natrénovaných modelů. Pro použití v rámci **tréninku** modelu jsou ovšem naprosto nevhodné. Trénink modelu je v jádru optimalizační úloha a jako taková potřebuje mít na vstupu spojitou diferencovatelnou funkci. Proto byl zaveden koncept **ztrátové funkce**.

1.3.1 Ztrátová funkce

Ztrátová funkce (někdy také nákladová funkce) je zpravidla značená notací $J(\theta)$, kde θ představuje **váhy** modelu. Tato funkce nabývá hodnot z $\mathbb{R}$ a používá se k výpočtu ztráty (chyby) modelu, což je číslo vyjadřující jak dobře konkrétní algoritmus modeluje vstupní datovou množinu.

> *„Nákladová funkce redukuje všechny dobré a špatné aspekty komplexního systému na jedno číslo, skalární hodnotu, která umožňuje hodnocení a porovnání kandidátských řešení.“* [3] (Reed et al., 1999)

Ztrátovou funkci si můžeme nadefinovat sami pro náš konkrétní problém, nebo můžeme vybírat z řady standardně používaných funkcí. Schopnost vybrat správnou ztrátovou funkci pro řešenou úlohu je kritická, jelikož přímo ovlivňuje proces učení a výsledný model.

Výběr ztrátové funkce může být nelehkým úkolem, poněvadž daná funkce musí nejen zachytit aspekty řešeného problému, ale měla by také brát v potaz zájmy zainteresovaných stran. V závislosti na vytyčených cílech daného projektu mohou dvě na první pohled totožné úlohy používat zcela odlišné ztrátové funkce (Reed et al., 1999).

Jednou ze základních ztrátových funkcí je střední čtvercová chyba MSE (mean squared error):

$$MSE = J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \tag{1.6}$$

Výsledkem MSE je průměrná chyba odhadu, tedy rozdílu mezi hodnotou predikovanou modelem $h_\theta(x^{(i)})$ a skutečnou hodnotou $y^{(i)}$, kde m je celkový počet případů. Střední čtvercová chyba se používá především v případě regresních úloh.

[3] *„The cost function reduces all the various good and bad aspects of a possibly complex system down to a single number, a scalar value, which allows candidate solutions to be ranked and compared.“*

U nebinárních klasifikačních úloh (počet tříd $k > 1$) se takřka vždy setkáme s křížovou entropií:

$$CrossEntropy(P, Q) = -\sum_{x} P(x) \log Q(x) \tag{1.7}$$

Křížová entropie vychází z teorie informace a standardně se používá k měření chyby mezi dvěma rozděleními pravděpodobnosti P a Q.

Při modelování klasifikačního problému, jehož cílem je mapování vstupních proměnných na výstupní třídy, můžeme problém uchopit jako predikci pravděpodobnosti, že daný příklad patří ke každé ze tříd k. V tréninkové datové množině je pravděpodobnost každé třídy buď 1, nebo 0, protože pro každý vstupní příklad známe **správnou** odpověď.

Při použití křížové entropie hledáme takové hodnoty modelových vah θ, které minimalizují chybu mezi modelem predikovaným rozdělením pravděpodobnosti Q a skutečným rozdělením pravděpodobnosti P.

Jak již bylo řečeno výše, proces učení je ve skutečnosti optimalizační úloha, respektive minimalizační problém $J(\theta) \rightarrow \mathbf{min}$. Tato minimalizace probíhá s využitím optimalizačních algoritmů, přičemž jedním z nich je i gradientní sestup, který budeme používat v této práci.

1.3.2 Gradientní sestup

Gradientní sestup je populární optimalizační algoritmus a zdaleka nejrozšířenější způsob optimalizace neuronových sítí. Gradientní sestup minimalizuje vstupní ztrátovou funkci $J(\theta)$ iterativní změnou jejího parametru θ.

Pro snadnější pochopení je gradientní sestup často popisován analogií, kdy se horolezec snaží najít co nejkratší cestu z hory do údolí. Před každým krokem se horolezec rozhlédne kolem sebe a vykročí směrem, kde je sráz nejstrmější. Tímto způsobem se dostane do údolí nejrychleji a pomocí nejmenšího počtu kroků.

Matematickým ekvivalentem strmosti srázu je gradient funkce (značený ∇), což je zobecněný tvar derivace pro funkci s více proměnnými. Gradient má formu vektoru parciálních derivací vstupní ztrátové funkce, přičemž směr tohoto vektoru odpovídá směru největšího růstu grafu funkce v daném bodě.

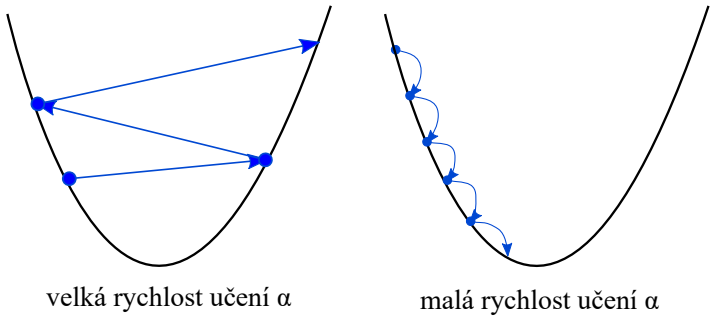
Minimalizace ztrátové funkce je dosaženo úpravou vah θ v opačném směru než je směr gradientu. Velikost jednoho kroku (tj. jedné úpravy vah) je pak dána produktem gradientu a parametru rychlosti učení α (learning rate). Rychlost učení je jedním z hyperparametrů modelu, což jsou nastavení, jejichž hodnoty musíme zvolit **před** samotným začátkem tréninku modelu (Ruder, 2016).

Jeden krok iterace gradientního algoritmu je dán rovnicí:

$$\theta = \theta - \alpha \nabla J(\theta) \tag{1.8}$$

Tento krok je opakován tak dlouho, dokud nedojde ke konvergenci algoritmu a dosažení lokálního minima ztrátové funkce. Pokud je ztrátová funkce konvexní, je každé lokální minimum zároveň i globální minimum (Anderson, 1995).

V závislosti na zvolené hodnotě parametru rychlosti učení α může dojít k několika situacím: Pokud je rychlost učení příliš velká, může algoritmus přeskočit lokální minimum funkce a dokonce může dojít i k divergenci a **zvýšení** chyby modelu (viz obrázek 1.3). Pokud je α příliš malá, bude gradientní sestup velmi pomalý. Za předpokladu, že je vstupní funkce nekonvexní, může zároveň dojít k situaci, kdy algoritmus konverguje na lokálním minimu, které by s větší hodnotou α přeskočil a mohl by teoreticky dosáhnout lepšího výsledku. Rychlost učení musí být vždy ověřena experimentálně a nelze ji předem určit (Goodfellow et al., 2016).



Obrázek 1.3: Chování gradientního sestupu v závislosti na rychlosti učení

Rozeznáváme tři základní typy gradientního sestupu:

- **Dávkový gradientní sestup (BGD - batch gradient descent)**: V případě BGD projde algoritmus v každém kroku **celou** tréninkovou množinu dat. Jedná se o pomalý a neefektivní přístup. V případech, kdy je tréninková množina rozsáhlá a nevejde se do paměti systému, není možné BGD použít.
- **Stochastický gradientní sestup (SGD - stochastic gradient descent)**: SGD používá v každé iteraci pouze **jeden** náhodně vybraný případ z tréninkové množiny a na jeho základě provede úpravu vah θ.
- **Mini-dávkový gradientní sestup (MBGD - mini-batch gradient descent)**: Mini-dávkový gradientní sestup spočívá v použití n případů v každé iteraci. Konkrétní počet je zvolen dle dané úlohy, modelu a výpočetního omezení systému.

2. Umělé neuronové sítě

Simon Haykin definuje umělou neuronovou síť jako výpočetní systém tvořený kombinací jednoduchých propojených procesních prvků, které dokáží získat znalosti z prostředí pomocí učícího procesu a tyto znalosti ukládají ve svých spojeních (Haykin, 1999).

Tyto procesní prvky se nazývají umělé neurony a jejich předobrazem jsou neurony tvořící lidský nervový systém. Lidský mozek slouží již velmi dlouho jako významná inspirace pro vědce všech oborů, počítačových vědců nevyjímaje. V roce 1943 vyvinuli neurolog Warren S. McCulloch a logik Walter Pitts první koncepční model umělé neuronové sítě. Ve svém článku *A logical calculus of the ideas imminent in nervous activity* popisují koncept neuronu, jedné buňky existující v síti buněk, která přijímá vstupy, následně tyto vstupy zpracovává a generuje výstup (McCulloch et al., 1943).

Cílem jejich práce a práce mnoha dalších vědců a výzkumníků ale nebylo přesně popsat, jak biologický mozek funguje. Umělá neuronová síť byla navržena jako výpočetní model k řešení určitých druhů problémů.

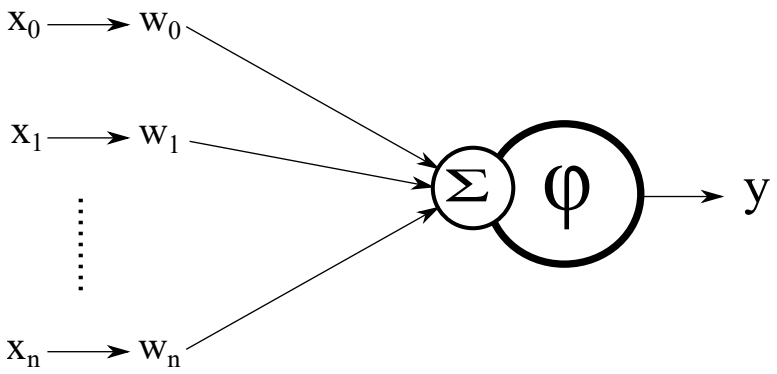
Klasicky byly výpočetní systémy využívány k řešení problémů, které jsou obtížné pro člověka, ale jednoduché pro stroj. Příkladem může být výpočet třetí odmocniny z čísla 42, což stroji zabere jednu milisekundu, ale průměrný člověk není schopen tento příklad vypočítat. Oproti tomu umělé neuronové sítě se využívají pro problémy, které jsou **lehké** pro člověka a obtížné pro stroj (rozpoznání obličeje, přečtení ručně psaného textu). Souhrnně tyto problémy označujeme jako rozpoznání vzorů (pattern recognition) a právě zde neuronové sítě excelují.

Jedním z klíčových prvků neuronové sítě je její schopnost učit se. Neuronová síť není jen komplexním systémem, ale komplexním **adaptivním** systémem, což znamená, že může měnit svou vnitřní strukturu na základě informací, které skrze ní proudí. Toho je dosaženo nastavením vah, které připadají spojům mezi dvěma neurony (Haykin, 1999).

2.1 Umělý neuron

Umělé neurony jsou základními stavebními prvky každé neuronové sítě. Skládají se z následujících částí:

- Vstupní signál ve formě vektoru příznaků $x = \{x_0, x_1, x_2, \ldots, x_n\}$
- Bias b: Bias si můžeme představit jako absolutní člen předpisu funkce, který umožňuje posouvat křivku funkce doprava, či doleva. Je často znázorňován jako samostatný prvek, ale v rámci této práce ho budeme považovat za vstup x_0 který má konstatní hodnotu $x_0 = 1$ a připadá mu váha θ_0
- Váhový vektor θ (někdy také značeno w)
- Aktivační (přenosová) funkce φ
- Výstupní signál $y = \varphi(\sum_{i=0}^{m} w_i x_i)$, vektorově $\varphi(\vec{x} \cdot \vec{w})$



Obrázek 2.1: Diagram umělého neuronu

Výstup umělého neuronu y odpovídá váženému součtu vah a vstupů, na který je aplikována aktivační funkce φ. Nazývá se aktivační funkcí, protože řídí prahovou hodnotu, při které je neuron aktivován, a sílu výstupního signálu. Mezi nejznámější aktivační funkce patří skoková funkce, funkce sigmoid, hyperbolický tangens a ReLU (rectified linear unit).

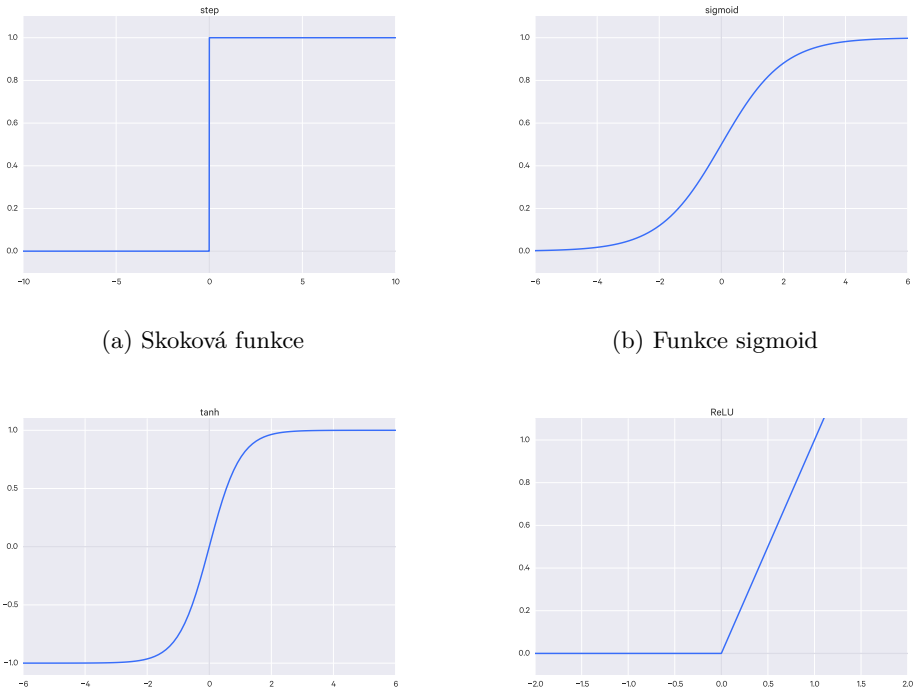
Skokové funkce jsou dnes již nepoužívané aktivační funkce, jejichž výstupem je buď 1, nebo 0 v závislosti na prahové hodnotě (viz obrázek 2.2 - a). Jelikož se jedná o nediferencovatelné funkce, nelze je pro hluboké neuronové sítě použít (Goodfellow et al., 2016).

Funkce sigmoid je logistická funkce s předpisem $\sigma(z) = \frac{1}{1+e^{-z}}$, která nabývá hodnot z intervalu $< 0{,}1 >$ (viz obrázek 2.2 - b). Po dlouhou dobu byla sigmoida výchozí aktivační funkcí v neuronových sítích. Použijeme ji především v případech, kdy chceme, aby výstupem neuronu byla *pravděpodobnost* určitého jevu. Příkladem může být binární klasifikace.

Zobecněnou logistickou funkcí, jejíž výstupem je pravděpodobnostní rozložení pro n jevů, je funkce **softmax**, která je vhodná pro nebinární klasifikaci.

Hyperbolický tangens (tanh) je podobný funkci sigmoid. Obor hodnot je ale narozdíl od sigmoidy $H(f) =< -1{,}1 >$ (viz obrázek 2.2 - c). Předpis této funkce je $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Modely používající hyperbolický tangens se lépe trénují a zpravidla mívají i lepší výkon než modely používající funkci sigmoid (Goodfellow et al., 2016).

ReLU, neboli rectified linear unit, je funkce s předpisem $R(z) = max(0, z)$ (viz obrázek obrázek 2.2 - d). V současné době se jedná o „zlatý standard“ a nejpoužívanější aktivační funkcí v oblasti hlubokého učení. Poprvé si získala pozornost odborné veřejnosti v letech 2009 a 2011, kdy bylo ukázáno, že použitím ReLU dochází k odstranění problému mizejícího gradientu, čímž došlo ke skokovému zlepšení výkonu hlubokých modelů (Nair et al., 2010).



(a) Skoková funkce

(b) Funkce sigmoid

(c) Hyperbolický tangens

(d) Funkce ReLU

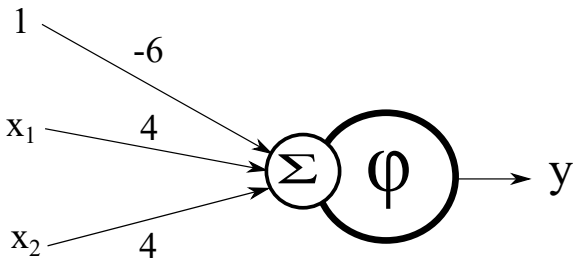
Obrázek 2.2: Vybrané aktivační funkce

2.2 Perceptron

Nejjednodušším typem umělé neuronové sítě je perceptron, který byl vytvořen Frankem Rosenblattem na konci 50. let 20. století (Rosenblatt, 1958). Perceptron je jednovrstevná neuronová síť tvořená jedním umělým neuronem. Využívá skokovou aktivační funkci a pro výstup y tedy platí

$$y = \begin{cases} 0, & \text{pokud } x \cdot w <= 0. \\ 1, & \text{pokud } x \cdot w > 0. \end{cases} \tag{2.1}$$

Jednoduchým příkladem použití perceptronu může být například implementace logické funkce AND (viz obrázek 2.3).



Obrázek 2.3: Implementace logické funkce AND pomocí perceptronu

V uvedeném příkladě platí, že $w = \{-6, 4, 4\}$ a po provedení výpočtu získáme pravdivostní tabulku, která ukazuje, že výstup perceptronu y a výstup standardní logické funkce AND je totožný (viz tabulka 2.1).

Tabulka 2.1: Pravdivostní tabulka funkce AND implementované pomocí perceptronu

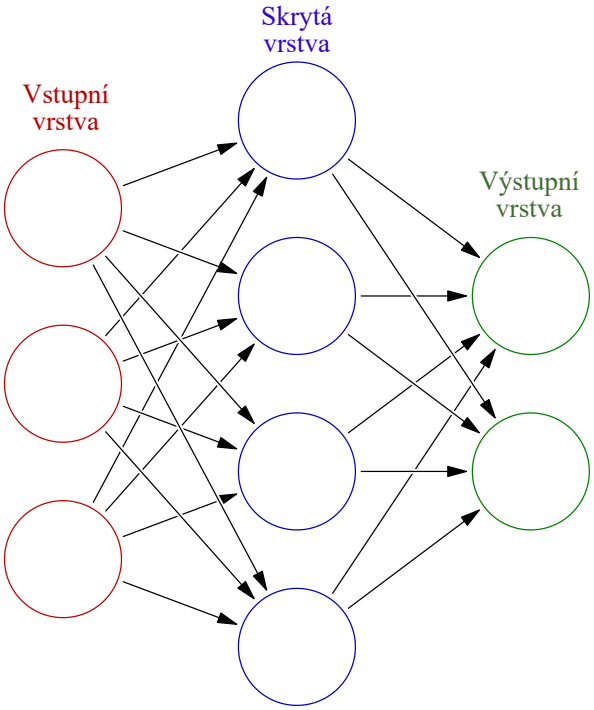
x_1, x_2	$\sum_{i=0}^{m} w_i x_i$	y	AND
0 0	$1 * -6 + 0 * 4 + 0 * 4 = -6$	0	0
0 1	$1 * -6 + 0 * 4 + 1 * 4 = -2$	0	0
1 0	$1 * -6 + 1 * 4 + 0 * 4 = -2$	0	0
1 1	$1 * -6 + 1 * 4 + 1 * 4 = 2$	1	1

Stejným způsobem lze pomocí perceptronu implementovat i další logické funkce a jednoduché algoritmy binární klasifikace. V praxi se ale téměř vždy setkáme s neuronovými sítěmi, které mají vícevrstvou architekturu a skládají se z velkého množství umělých neuronů.

2.3 Vícevrstvé neuronové sítě

Zřetězením více neuronů a jejich organizací do vrstev získáme vícevrstvou neuronovou síť. Jelikož výstup jednoho neuronu slouží jako vstup pro další neuron, můžeme si neuronovou síť

představit jako složenou funkci $y = f(g(x))$. Zpravidla se ale setkáme s přehlednější grafickou reprezentací ve formě orientovaného grafu, kde vrcholy představují jednotlivé neurony a hrany vyjadřují tok informací mezi nimi (viz obrázek 2.4). Každé hraně, která spojuje dva neurony, připadá váha Θ_{ij}, kde i je daná vrstva a j daný neuron.



Obrázek 2.4: Schéma jednoduché neuronové sítě

Neuronová síť se skládá ze tří základních typů vrstev: vstupní, skryté a výstupní.

Vstupní vrstva přijímá vstupy z datové množiny a posílá je do další vrstvy. V této vrstvě neprobíhají žádné výpočty. Někdy se také označuje jako viditelná vrstva, jelikož se jedná o odkrytou část sítě, která dokáže přijímat vstupy.

Skryté vrstvy jsou všechny vrstvy mezi vstupní a výstupní vrstvou. Jsou „skryté“, jelikož jejich hodnoty nejsou součástí vstupních dat, ale model se je sám učí. Síť, která má více než jednu skrytou vrstvu se nazývá hluboká neuronová síť (deep neural network).

Vzhledem ke zvýšení výpočetního výkonu a existenci efektivních numerických knihoven mohou být konstruovány i **velmi** hluboké neuronové sítě, což jsou sítě se stovkami skrytých vrstev. Teoretické i empirické důkazy indikují, že právě dostatečná hloubka sítě je jedním z důležitých faktorů určujících úspěch, či neúspěch modelu (R. K. Srivastava et al., 2015).

Výstupní vrstva je poslední vrstvou každé neuronové sítě a je zodpovědná za výstup hodnoty, nebo vektoru hodnot, které odpovídají požadovanému formátu pro řešený problém.

2.4 Učení neuronové sítě

Učení hluboké neuronové sítě probíhá ve třech krocích: Nejdříve jsou náhodně incializovány váhy Θ. Následně se iterativně opakuje dopředné šíření (forward propagation) a zpětné šíření (back propagation).

Dopředné šíření je pouze jiným označením pro získání predikce modelu. Pro vstup x postupně vypočteme hodnotu každého neuronu v sítí, dokud nezískáme závěrečný výstup y.

V rámci **zpětného šíření** je porovnán predikovaný výstup y s očekávaný výstupem (ground truth) a je vypočtena hodnota ztrátové funkce $J(\Theta)$. Ztráta je pak pomocí optimalizačního algoritmu zpětně šířená od poslední vrstvy k první vrstvě a jednotlivé váhy jsou upraveny v poměru ve kterém přispěly k výši celkové ztráty (Hecht-Nielsen, 1992).

Učení modelu končí v prvních pokusech zpravidla neúspěchem. Dvě hlavní příčiny špatného výkonu modelu jsou nedoučení (underfitting) a přeučení (overfitting).

Nedoučení je stav, kdy model není schopný dostatečně zachytit vnitřní strukturu vstupních dat a dosahuje špatných výsledků jak na tréninkových datech, tak i na validačních/testovacích datech. Tento problém je očividný na první pohled a je způsoben buď nevhodně zvolenou architekturou modelu, nastavením hyperparametrů, nebo nedostatečnou velikostí datové množiny.

Přeučení nastává, když se model naučí detail a šum vstupních dat jako nové koncepty, které se ve skutečnosti nevztahují na nová data a negativně ovlivňují schopnost generalizace modelu. Model tedy bude mít velmi dobré výsledky na tréninkové množině, ale špatné výsledky na nových datech (N. Srivastava et al., 2014).

Nelineární modely, které mají větší flexibilitu při učení cílové funkce, jsou náchylnější k problému přeučení. Jednou z metod, která řeší problém přeučení, je regularizace, která funguje na principu penalizace modelu v případech, kdy se učí příliš komplexní příznaky.

Z těchto důvodů je v rámci tréninku důležité použít i validační data, na kterých model není trénován a která slouží k indikaci případného přeučení. Jakmile je velký rozdíl mezi trénovací a validační ztrátou, existuje velká šance, že dochází právě k přeučení.

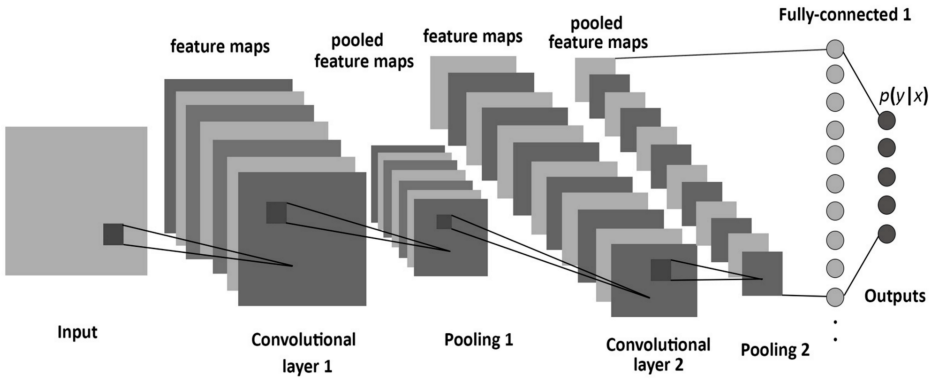
2.5 Konvoluční neuronové sítě

Konvoluční neuronové sítě (CNN) jsou specifickou kategorií umělých neuronových sítí využívanou především v oblasti počítačového vidění. Velká část aplikací simulujících nějakým způsobem lídsky smysl zraku je dnes postavena právě na technologii CNN. Zjednodušeně můžeme konvoluční neuronovou síť definovat jako umělou neuronovou síť, která využívá matematické operace konvoluce alespoň v jedné svojí vrstvě (Goodfellow et al., 2016).

Na rozdíl od klasických neuronových sítí mají vrstvy CNN až tři rozměry: výšku, šířku a hloubku. Jako vstup CNN totiž slouží bitmapové grafické soubory ve formě tří rozměrné tabulkové struktury nazývané tenzor. Tento tenzor obsahuje hodnoty jednotlivých pixelů vstupního obrázku z intervalu $< 0{,}255 >$. Třetím rozměrem je v tomto případě hloubka, která představuje počet barevných kanálů vstupního obrázku. U barevných obrázků se setkáme s kanály RGB (red, green a blue) a hloubka je tedy 3. Naopak černobílé obrázky mají kanál pouze jeden.

Konvoluční neuronové sítě využívající standardní architekturu (viz obrázek 2.5) jsou složeny ze dvou základních částí:

- **Extraktor příznaků**: Jedná se o souhrnné označení konvolučních a sdružovacích vrstev, které rozpoznávají příznaky ze vstupních dat.
- **Klasifikátor**: Plně propojené vrstvy na konci sítě se označují jako klasifikátor. Jejich výstupem je rozložení pravděpodobnosti pro cílové třídy.



Obrázek 2.5: Standardní architektura konvoluční neuronové sítě

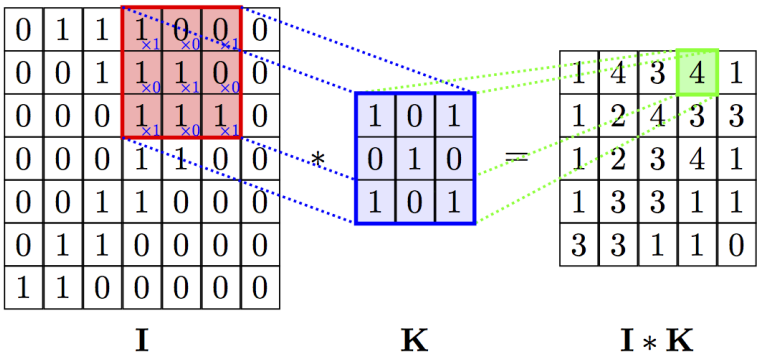
2.5.1 Konvoluce

Základním stavebním kamenem každé CNN je konvoluční vrstva, která využívá matematické operace konvoluce. Konvoluce transformuje dvě vstupní funkce na jednu výstupní funkci a je dána následující rovnicí:

$$(f * g)(x) = \sum_{j=1}^{m} g(j) f(x - j + m/2) \tag{2.2}$$

Na vstupu konvoluční vrstvy je obrázek, na který je aplikováno konvoluční jádro (filtr). Jádro má formu matice reálných čísel o rozměrech $m\,x\,m$ (obvykle $3\,x\,3$ až $6\,x\,6$). Jádro postupně aplikujeme na vstupní obrázek, tj. násobíme výřez vstupního obrázku (receptivní pole) o rozměrech $m\,x\,m$ konvolučním jádrem, čímž získáme hodnotu jedné buňky výstupní příznakové mapy. Následně posuneme jádro o jednu buňku a opakujeme, dokud neprojdeme celý obrázek (viz obrázek 2.6). Výsledkem je tzv. příznaková mapa, která dle hodnoty jádra detekuje různé příznaky vstupního obrázku jako jsou např. hrany, barvy, křivky atd. Proto se konvolučnímu jádru někdy říká také detektor příznaků.

Konvoluční jádra jsou zpravidla náhodně incializována a síť se je učí sama. Řetězením konvolučních vrstev za sebe se síť učí stále komplexnější příznaky až je ve výsledku schopná klasifikovat cílovou třídu (Krizhevsky et al., 2012).



Obrázek 2.6: Ukázka konvoluční operace

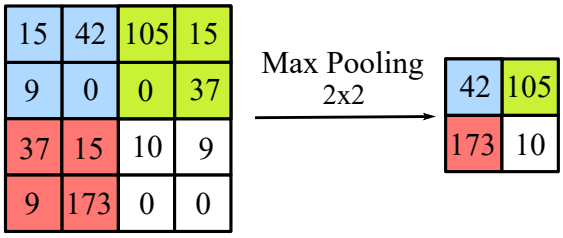
Kromě hlubokého učení se s konvolucemi setkáme ve všech grafických editorech, jelikož na této operaci jsou postaveny nástroje pro zpracování obrazu jako je zaostření, rozmazání obrazu a další. Například hojně používanému Gaussovskému rozostření (vychází z Gaussovy funkce) odpovídá při zvolené velikosti 3 x 3 jádro:

$$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

2.5.2 Sdružování

Sdružovací vrstva snižuje dimenzionalitu příznakové mapy získané v konvoluční vrstvě, ale zároveň z ní zachovává nejdůležitější informaci. Snížením dimenzionality dosáhneme i snížení celkového počtu parametrů a výpočetních operací v síti, čímž dochází k výkonové optimalizaci. Zároveň se model stává odolným vůči menším deformacím a zkreslením vstupního obrázku, jelikož malá změna vstupu nevede ke změně výstupu sdružování. Existuje několik základních typů sdružování: sdružování dle maxima, sdružování dle minima a sdružování dle průměru.

Sdružování používá filtr (nejčastěji o velikosti 2 x 2), který je aplikován na vstupní příznakovou mapu a to v závislosti na velikosti kroku (stride). Tím dojde k agregaci pixelů do jedné hodnoty a to buď na základě výběru maximální hodnoty (sdružování dle maxima), průměrné hodnoty (sdružování dle průměru), nebo minimální hodnoty (sdružování dle maxima). V případě zvolené velikosti filtru 2 x 2 dojde ke zmenšení dimenzionality na polovinu (viz obrázek 2.7).



Obrázek 2.7: Sdružování dle maxima s filtrem 2x2 a velikostí kroku 2

2.6 Přenos učení

Tvorba rozsáhlých neuronových sítí „od nuly“ (tj. s náhodně incializovanými váhami) je velice náročný proces, který může na výkonných paralelizovaných systémech trvat desítky hodin čistého času. Např. konvoluční neuronová síť Inception-v3 společnosti Google klasifikující tisíc tříd byla trénována na padesáti grafických kartách NVidia Kepler a to po dobu několika dnů (Szegedy et al., 2016).

Existuje metoda, která nám za určitých podmínek umožňuje rychle a efektivně vytvořit model dosahující state-of-the-art výsledků. Tato metoda se nazývá přenos učení (transfer learning) a využívá vlastnosti generalizace, kterou robustní neuronové sítě zpravidla disponují (Goodfellow et al., 2016).

Přenos učení znamená použití již existujícího modelu, který řeší úlohu A, jako výchozí bod pro tvorbu nového modelu řešícího úlohu B, přičemž úlohy A a B si musí být nějakým způsobem podobné (Yosinski et al., 2014). Použitím podkladového modelu dochází k přenosu znalostí, které jsou uloženy ve váhách daného modelu. Dle konkrétní úlohy můžeme buď využít všechny vrstvy a váhy podkladové sítě, nebo pouze některé části. Zpravidla budeme v rámci přenosu učení trénovat pouze vybrané vrstvy, čímž dochází k významným časovým úsporám.

Kromě úspory času řeší přenos učení i jedno z největších omezení, se kterým se v oblasti strojového učení setkáme, a to **nedostatek** kvalitních vstupních dat. Vzhledem k tomu, že využíváme již naučených znalostí podkladového modelu, potřebujeme k dosažení srovnatelných výsledků řádově menší datový korpus (Donahue et al., 2013).

Přenos učení je mocným nástrojem a mnozí ho považují za jeden z klíčových prvků, které povedou k masivnímu komerčnímu úspěchu a dalšímu rozšíření aplikací strojového učení. Jedním z příkladů možného využití této metody je učení pomocí simulací. Pro mnoho zajímavých aplikací strojového učení, které s okolím interagují pomocí hardware, je shromažďování dat a trénink modelu v reálném světě buď drahé, časově náročné, nebo prostě příliš nebezpečné. Proto je vhodné shromažďovat data jiným, méně riskantním způsobem.

Zde se nabízí právě simulace, která usnadňuje proces získávání dat a zároveň umožňuje rychlé trénování modelu, jelikož učení může být takřka neomezeně paralelizováno. Simulace je záro-

veň prerekvizitou pro rozsáhlé projekty strojového učení, které interagují s reálným světem, jako jsou např. autonomní vozidla.

Další oblastí, kde je učení pomocí simulací klíčové, je robotika: Trénování robota v reálném prostředí je pomalé a drahé. Učení ve virtuálním prostředí a následný přenos znalostí do reality nás zbavuje těchto problémů. Tato metoda již byla s úspěchem ověřena na experimentu s robotickou paží v práci *Sim-to-Real Robot Learning from Pixels with Progressive Nets* (Rusu et al., 2016).

3. Aplikační oblast

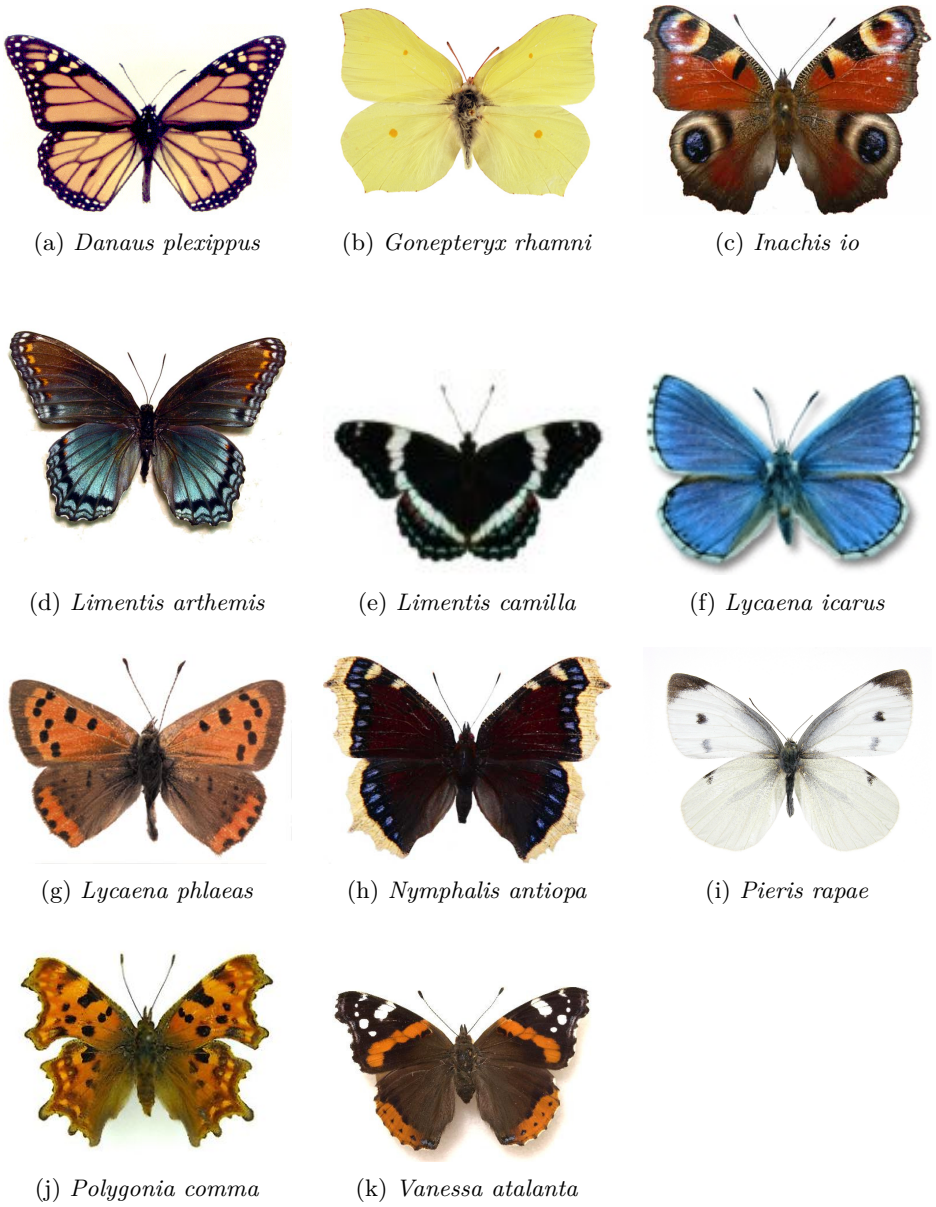
Jako aplikační oblast pro klasifikační úlohu byl zvolen živočišný řád motýlů (*Lepidoptera*). Při výběru bylo zapotřebí zohlednit několik významných faktorů:

- **Dostupnost dat:** Pro tvorbu jakéhokoliv modelu pomocí strojového učení je zapotřebí relativně velké množství vstupních dat. Pokud data nesbíráme sami, ale používáme veřejně dostupné zdroje, musí být také zajištěno splnění licenčních podmínek.
- **Řešitelnost:** Aplikační oblast musí být zvolena tak, aby úloha byla řešitelná s ohledem na výpočetní, časová a další omezení. Trénování jakékoliv sítě zpracovávající obraz je náročné na výpočetní kapacity systému, a proto musí být v případě klasifikační úlohy vybrán vhodný počet cílových tříd.
- **Variabilita:** Cílové třídy musí být dostatečně variabilní, jinak se snižuje pravděpodobnost, že model bude dosahovat dobrých výsledků. V rámci řádu *Lepidoptera* to znamená, že vybrané druhy musí mít významné morfologické odlišnosti. Pokud by od sebe dané druhy nedokázal po důkladném zkoumání rozeznat laik, dá se předpokládat, že je nerozezná ani konvoluční neuronová síť.

S ohledem na body výše bylo vybráno jedenáct druhů motýlů a to z čeledi babočkovitých (*Nymphalidae*), běláskovitých (*Pieridae*) a modráskovitých (*Lycaenidae*). Tato práce tedy pokrývá pouze malý zlomek z celkového počtu druhů, jelikož řád *Lepidoptera* jich celkově obsahuje okolo 180 000 (Capinera, 2008).

Jako zdroj vstupních dat posloužil veřejně dostupný datový korpus Leeds Butterfly Dataset (Wang et al., 2009), který byl rozšířen o data z databáze ImageNet (Deng et al., 2009). Použití dat z ImageNet bylo pro nekomerční účely schváleno Princetonskou a Standfordovou univerzitou.

Výsledný datový korpus se skládá z jedenácti tříd a ke každé z nich připadá 350 vstupních obrázků o různých velikostech ve formátu JPEG (ukázka jednotlivých tříd viz obrázek 3.1 na další straně).



(a) *Danaus plexippus* (b) *Gonepteryx rhamni* (c) *Inachis io*

(d) *Limentis arthemis* (e) *Limentis camilla* (f) *Lycaena icarus*

(g) *Lycaena phlaeas* (h) *Nymphalis antiopa* (i) *Pieris rapae*

(j) *Polygonia comma* (k) *Vanessa atalanta*

Obrázek 3.1: Ukázka jednotlivých tříd

4. Implementace

V této kapitole jsou popsaný použité technologie a postup tvorby modelu konvoluční neuronové sítě s pomocí metody přenosu učení. Následně je popsán proces optimalizace vytvořeného modelu a problémy, které bylo nutno překonat. Na závěr je model vyhodnocen.

V rámci této kapitoly jsou popsány vybrané části kódu, celý skript implementace lze pak nalézt v Příloze A.

4.1 Použité technologie

Strojové učení stojí na pevných matematických základech, které mohou leckoho odradit. Existuje velké množství knihoven, které poskytují větší či menší míru abstrakce od těchto základů a tím nejenom zpřístupňují strojové učení širšímu publiku, ale zároveň i značně urychlují vývoj a výzkum v oblastech strojového učení a umělé inteligence.

Mezi nejznámější knihovny patří TensorFlow, Theano, Caffe, Pytorch, SciKit a Keras. Tyto knihovny jsou spravovány různými institucemi od univerzit po korporace jako je Facebook a Google.

Pro účely této práce byly využity knihovny TensorFlow (Abadi et al., 2016) a Keras (Chollet et al., 2015), které jsou detailněji popsány níže. Programovacím jazykem této práce je Python a to v open-source distribuci Anaconda, která je zaměřená na strojové učení a data science.

Jako podkladový model byla zvolena síť MobileNet (Howard et al., 2017), která se vyznačuje architekturou orientovanou především na výkon. Jedná se o malý model (< 20 MB), který je zároveň velmi rychlý a poskytuje vynikající výkony na benchmarku ImageNet. Síť MobileNet je vhodná především v případech, kdy chceme model použít i na mobilních zařízeních, nebo nemáme k dispozici dostatečný výpočetní výkon pro trénink robustnějších modelů.

4.1.1 TensorFlow

TensorFlow je softwarová knihovna pro rychlé numerické výpočty vytvořená a spravovaná společností Google. Tato knihovna je zaměřená na problematiku strojového učení, především pak na neuronové sítě a hluboké učení. V roce 2015 bylo TensorFlow API zpřístupněno široké veřejnosti pod Apache 2.0 open-source licencí. Výchozím programovacím jazykem je Python, ale existuje i C++ API.

Narozdíl od ostatních numerických knihoven zaměřených na hluboké učení, jako je např. Theano, byla knihovna TensorFlow od počátku koncipována pro využití nejen ve výzkumu a vývoji, ale i v produkčních systémech. Aplikace vytvořené v TensorFlow mohou být s

minimálními změnami použity na široké škále heterogenních systémů, ať už se jedná o mobilní zařízení, tablety, nebo rozsáhlé distribuované systémy stovek a tisíců výpočetních zařízení.

Základem každého TensorFlow programu je výpočetní graf, kde hrany grafu představují tok dat a uzly grafu představují jednotlivé výpočetní operace, které jsou nad daty prováděny.

Data jsou reprezentována pomocí tenzorů, což je zobecněný model n-dimenzionální tabulkové struktury. Každý tenzor má datový typ, tvar (shape) a řád (rank), který určuje dimenzionalitu daného tenzoru.

4.1.2 Keras

Keras je minimalistická Python knihovna pro hluboké učení, která slouží jako vysokoúrovňové API pro TensorFlow a Theano. Keras byl vytvořen François Cholletem a jeho hlavním účelem je poskytnout nástroj pro rychlou implementaci a prototypování modelů využívajících hluboké učení (Chollet et al., 2015).

Základními principy Kerasu jsou modularita, minimalismus a rozšiřitelnost.

Proces tvorby nového modelu se dá shrnout do pěti základních kroků:

1. **Definice modelu:** Model je datová struktura Kerasu, která představuje sekvenci jednotlivých vrstev neuronové sítě. Základem je metoda **model.add()**, která na konec daného modelu přidá novou vrstvu.

2. **Konfigurace modelu:** Konfigurace modelu spočívá ve specifikaci ztrátové funkce, optimalizačního algoritmu a metriky, které bude učící algoritmus využívat. Použitá metoda je **model.compile()**.

3. **Trénink modelu:** Jakmile je model nakonfigurovaný a vstupní data jsou ve správném formátu, můžeme spustit proces učení pomocí metody **model.fit()**.

4. **Vyhodnocení modelu:** Po vytrénování můžeme model vyhodnotit s použitím metody **model.evaluate()**, která vrací hodnotu sledované metriky a ztrátové funkce.

5. **Použití modelu:** Pokud jsme s modelem spokojeni, můžeme ho využít pro získání predikcí na nových datech. Použitá metoda je **model.predict()**.

4.2 Příprava prostředí

Nejdříve je pomocí příkazového řádku Anaconda vytvořeno virtuální prostředí s názvem bc_petm09.

```
> conda create --name bc_petm09
> activate bc_petm09
```

Následně jsou nainstalovány balíčky Keras, TensorFlow pro GPU a grafické balíčky potřebné pro tvorbu grafů.

```
> conda install tensorflow-gpu
> conda install graphviz pydot matplotlib
> conda install keras
```

Keras nativně využívá TensorFlow jako výchozí knihovnu, proto není zapotřebí žádné další konfigurace.

Před samotnou implementací modelu je nutné připravit vstupní data. Datový korpus musí být rozdělen na trénovací množinu, validační množinu a testovací množinu. Trénovací část je použita k hlavnímu procesu učení, validační část slouží ke kontrole, zda nedochází k přetrénování a testovací množina jsou potom data, která model během učení nemá vůbec k dispozici a je na nich proveden až závěrečný test modelu.

Použijeme standardně doporučované rozdělení datového korpusu na jednotlivé části, tj. poměr trénink : test : validace = 70% : 15% : 15% (Bengio, 2012).

Data také musí být uspořádána v hierarchické adresářové struktuře tak, abychom měli tři základní adresáře: test, train a validation. V nich se nacházejí adresáře nazvané dle jednotlivých tříd s odpovídajícími vstupními obrázky.

4.3 Tvorba modelu

Každý Python skript začíná importem závislostí a není tomu jinak ani v tomto případě. Jedná se o nezbytný krok, který nám umožní využít cílové třídy a metody zvolených knihoven. Keras v sobě obsahuje nejznámější modely konvolučních neuronových sítí a po jejich importu je můžeme bez jakýchkoliv dalších kroků používat. Mezi tyto modely patří např. InceptionV3, ResNetV2, VGG19 a MobileNet.

Za zmínku stojí třída SGD, díky které můžeme v modelu použít stochastický gradientní sestup, a třídy Dense a Dropout, které umožňují vytvořit nové plně propojené a výpadkové vrstvy.

```
1  import keras
2  import numpy as np
3
4  from keras.applications.mobilenet import MobileNet, preprocess_input
5  from keras.layers import Dense, Dropout
6  from keras.optimizers import SGD
7  from keras.models import Model
8  from keras.callbacks import EarlyStopping
9  from keras.preprocessing.image import ImageDataGenerator
10 from plot_functions import plot_history
11 from sklearn.metrics import classification_report
```

Výpis kódu 1: Import závislostí

V rámci počáteční přípravy dále definujeme hyperparametry a další globální nastavení nutná k tréninku modelu. Tato nastavení budou zároveň sloužit i k optimalizaci modelu. Nejvýznamnější z nich jsou:

- **Rychlost učení (learning rate)**: Pravděpodobně nejdůležitější hyperparametr, jehož změna vyvolává největší změny ve výkonnosti modelu. Standardně je doporučováno zvolit hodnoty v rozmezí $< 10^{-6}; 1 >$ (Bengio, 2012). Jelikož používáme metodu přenosu učení a trénujeme pouze finální část sítě, je rozumným předpokladem, že rychlost učení by měla být nižší, a proto byla zvolena počáteční hodnota 0.0001.

- **Velikost dávek (batch size)**: Určuje, kolik vstupních příkladů bude zpracováno před tím, než dojde k úpravě vah. Vzhledem k tomu, že vyšší hodnota tohoto parametru by měla zrychlovat proces učení (Smith et al., 2017), byla zvolena výchozí hodnota 128.

- **Počet epoch (epochs number)**: Určuje, kolikrát učící algoritmus projde celou tréninkovou množinu dat než doje k ukončení procesu. Keras nabízí funkcionalitu **early stopping**, díky které dojde k zastavení tréninku na základě sledované metriky, proto je počet epoch nastaven na relativně vysokou hodnotu 100.

- **Trénované vrstvy (trainable layers)**: Základní premisou přenosu učení je fakt, že netrénujeme celou síť, ale pouze vybranou část. Neexistuje metoda, které by určila, kolik přesně vrstev musíme přetrénovat, abychom dosáhli optimálních výsledků. Ideální hodnota se bude lišit v závislosti na podkladovém modelu, na úloze i na struktuře datového korpusu, a proto je vždy nutné nastavení počtu trénovaných vrstev ověřit experimentálně. Byla zvolena počáteční hodnota 6.

Nyní je možné vytvořit novou instanci modelu MobileNet, což je v Kerasu extrémně jednoduché. Při prvním spuštění je model automaticky stažen z GitHub repozitáře a uložen na lokální disk.

```
base_model = MobileNet(
        weights = 'imagenet',
        include_top = True)
```

V kódu výše si lze všimnout, že v konstruktoru třídy MobileNet jsou použity dva důležité parametry: Parametr **weights** udává, že váhy sítě nebudou náhodně inicializovány, ale budou použity hodnoty z benchmarku ImageNet. Parametr **include_top** potom říká, že chceme celou síť včetně posledních klasifikačních vrstev.

Síť MobileNet byla učena na 1000 třídách a obsahuje celkem 88 vrstev. Abychom ji mohli požít pro zvolenou aplikační oblast, je nutné manuálně odstranit pět posledních vrstev této sítě, které mají nevhodné rozměry a vlastnosti, a nahradit je plně propojenou vrstvou s jedenácti výstupními neurony, které odpovídají jedenácti klasifikovaným druhům motýlů (viz výpis kódu 2).

```
base_tensor = base_model.layers[-6].output
output_tensor = Dense(11, activation='softmax')(base_tensor)

model = Model(
        inputs = base_model.input,
        outputs = output_tensor)
```

Výpis kódu 2: Úprava podkladového modelu MobileNet

Dalším krokem je tzv. „zmrazení“ vrstev, které nemáme v úmyslu trénovat. Počet těchto vrstev je dán parametrem TRAINABLE_LAYERS. Ve smyčce projdeme všechny dotčené vrstvy a jednoduchým příkazem je označíme jako vrstvy, které nemají být trénovány.

```
for layer in model.layers[:-TRAINABLE_LAYERS]:
    layer.trainable = False
```

Model je připraven ke konfiguraci a kompilaci pomocí metody model.compile(). Tato metoda používá parametry:

- **Optimizer:** Instance optimalizačního algoritmu, v našem případě se jedná o stochastický gradientní sestup (SGD).
- **Loss:** Ztrátová funkce, zde křížová entropie (categorical_crossentropy),

- **Metrics:** Seznam metrik, které chceme měřit, zde pouze správnost modelu (accuracy).

Nejdříve vytvoříme instanci gradientního sestupu, který má v konstruktoru hyperparametr rychlost učení, a potom zavoláme metodu model.compile().

```
gradient_descent = SGD(
        lr = LEARNING_RATE)

model.compile(
        optimizer = gradient_descent,
        loss = 'categorical_crossentropy',
        metrics = ['accuracy'])
```

Výpis kódu 3: Vytvoření instance SGD a kompilace modelu

4.4 Předzpracování vstupních dat

Abychom mohli použít obrázky motýlů jako vstup pro podkladovou síť MobileNet, musí mít formu tenzoru o rozměrech 224 x 224 x 3 (Howard et al., 2017).

Keras obsahuje třídu ImageDataGenerator, která dokáže vstupní obrazová data rozdělit na dávky a zároveň na ně aplikovat funkci předzpracování a různé grafické transformace, které mohou teoreticky zvýšit výkonnost modelu. Tuto třídu použijeme pro vytvoření generátoru img_generator, pomocí kterého následně vygenerujeme trénovací, testovací a validační dávky obrázků.

```
img_generator = ImageDataGenerator(
        preprocessing_function = preprocess_input)

train_data = img_generator.flow_from_directory(
        TRAIN_PATH,
        target_size = (224,224),
        batch_size = BATCH_SIZE,
        color_mode = 'rgb',
        class_mode = 'categorical',
        shuffle = True)
...
```

Výpis kódu 4: Předzpracování dat a vytvoření datových dávek

Nyní máme k dispozici trénovací, validační i testovací data ve správném formátu a model je připraven pro trénink.

4.5 Trénink a optimalizace modelu

Celý trénink probíhá zavoláním metody model.fit_generator. Tato metoda má parametry:

- **Generator:** ImageDataGenerator obsahující trénovací dávky.
- **Steps per epoch:** Počet kroků v jedné epoše, standardně se počítá jako $\frac{\text{počet případů}}{\text{velikost dávky}}$ zaokrouhleno nahoru.
- **Validation data:** ImageDataGenerator obsahující validační dávky.
- **Epochs:** Počet tréninkových epoch.
- **Callbacks:** Jedná se o speciální funkcionalitu, která po každé epoše umožňuje zavolat určitou funkci, např. uložení aktuálního modelu, nebo kontrolu, která může trénink předčasně ukončit. Zde je použit callback early_stop, který zastaví trénink, pokud se hodnota ztrátové funkce tři epochy po sobě nesníží.

Jelikož metoda model.fit_generator() vrací historické hodnoty metrik na konci jednotlivých tréninkových epoch, můžeme pomocí knihovny Matplotlib (Hunter, 2007) vytvořit graf průběhu tréninku (implementace viz Příloha B).

```
history = model.fit_generator(
        generator = train_data,
        steps_per_epoch = training_step_size,
        validation_data = validation_data,
        validation_steps = validation_step_size,
        epochs = EPOCHS_NUMBER,
        callbacks = [early_stop],
        verbose = 2)
```

Výpis kódu 5: Trénink modelu

Jakmile je trénink modelu dokončen, můžeme ho pomocí metody **evaluate_generator()** vyhodnotit. Tato metoda projde všechny příklady z **testovací** množiny, přiřadí jim výslednou třídu a následně spočítá celkovou správnost modelu a hodnotu ztrátové funkce. Zde je opět důležité zdůraznit, že model vyhodnocujeme na datech, která neměl v průběhu učení k dispozici, a tudíž by výsledek měl odpovídat reálnému chování na nových datech.

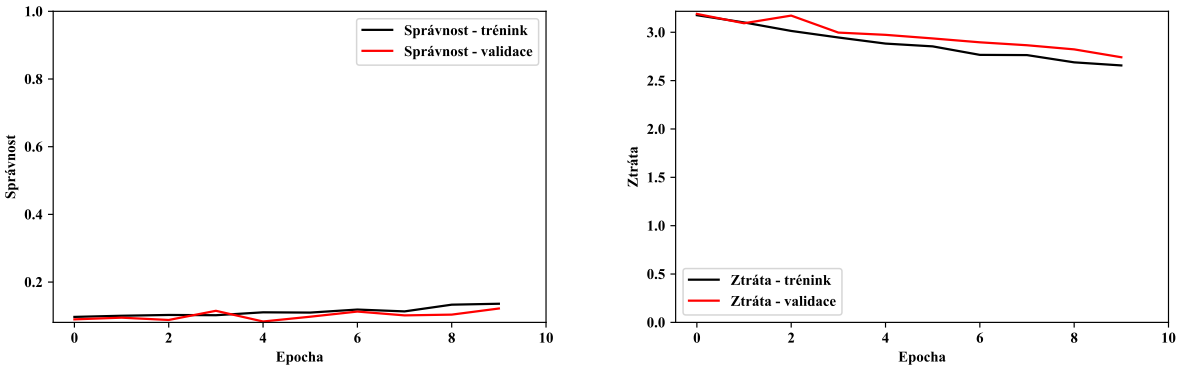
```
test_accuracy = model.evaluate_generator(
        generator = test_data,
        steps = validation_step_size,
        verbose = 1)

print(test_accuracy)
```

Výpis kódu 6: Vyhodnocení modelu

Výsledný model dosáhl po deseti epochách pouze 12% celkové správnosti (viz obrázek 4.1), což

při jedenácti výstupních kategoriích v podstatě odpovídá náhodnému výběru ($1/11 \approx 9{,}1\%$). Z výsledků se dá usuzovat, že počáteční hodnoty hyperparametrů nebyly vhodně zvoleny a je nutné vyzkoušet jiná nastavení.

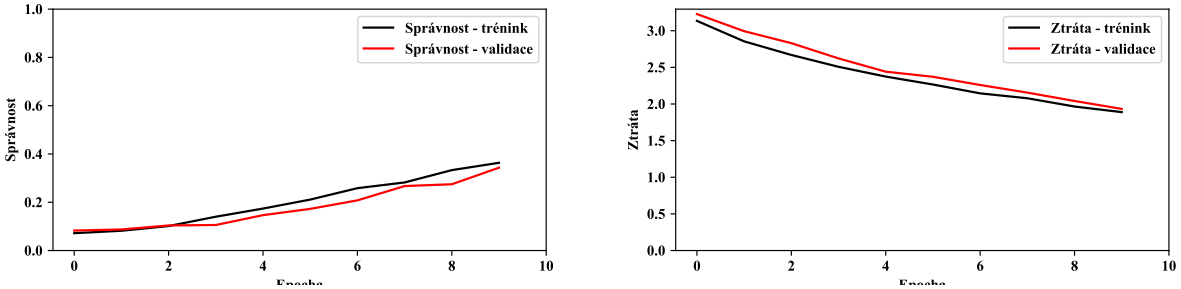


Obrázek 4.1: Experiment č. 1 - Správnost a Ztráta

Ukázalo se, že při použití stochastického gradientního sestupu může mít velikost dávek zásadní negativní vliv na kvalitu výsledného modelu (Keskar et al., 2016). To v kombinaci s nízkou hodnotou rychlosti učení pravděpodobně způsobilo faktickou nefunkčnost učícího algoritmu v experimentu č.1.

> *„Je-li rychlost učení příliš velká, gradientní sestup může neúmyslně zvýšit tréninkovou chybu místo jejího snížení. Je-li rychlost učení příliš malá, trénink je nejen pomalejší, ale může se stát, že se trvale zasekne s vysokou tréninkovou chybou.“* [1] (Goodfellow et al., 2016)

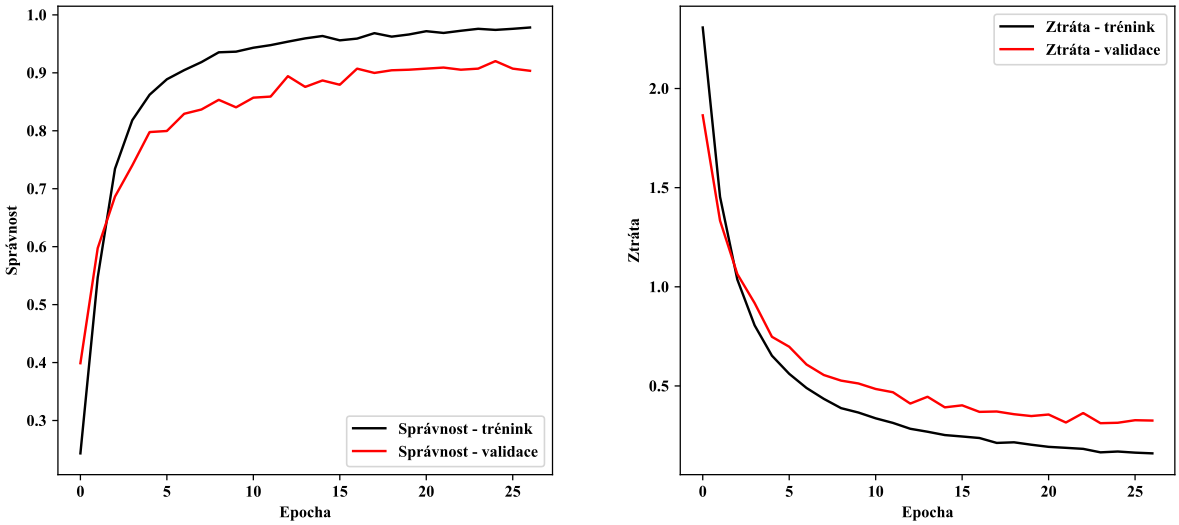
Experiment číslo 2 proběhl s hodnotou $BATCH\_SIZE = 32$ a jak je vidět z historických dat (viz obrázek 4.2), model dosáhl po deseti epochách správnosti 34,32%. Původní chyba tedy byla odstraněna. Z trendu grafu ale vyplývá, že aby došlo ke konvergenci ztrátové funkce, musel by algoritmus projít ještě několik desítek epoch, což není z časového hlediska přijatelné, jelikož jedna epocha může trvat až několik minut.



Obrázek 4.2: Experiment č. 2 - Správnost a Ztráta

[1] *„When the learning rate is too large, gradient descent can inadvertently increase rather than decrease the training error. When the learning rate is too small, training is not only slower, but may become permanently stuck with a high training error.“*

V experimentu číslo 3 použijeme nastavení $LEARNING\_RATE = 0.001$, tedy o řád vyšší, než původní hodnota. To způsobí větší změny vah a tím pádem by hypoteticky mělo dojít i k rychlejší konvergenci ztrátové funkce.



Obrázek 4.3: Experiment č.3 - Správnost a Ztráta

Z počátku skutečně došlo k rychlé konvergenci ztrátové funkce a model dosáhl po 27 epochách správnosti 90,35% (viz obrázek 4.3). V pozdějších epochách ale došlo k oscilaci měřených metrik, z čehož vyplývá, že hodnota rychlosti učení je v těchto případech moc vysoká a algoritmus nedokáže udělat dostatečně malé změny parametrů nutných k dosažení lokálního optima. K řešení tohoto problému můžeme využít parametry stochastického gradientního sestupu **decay** a **momentum**.

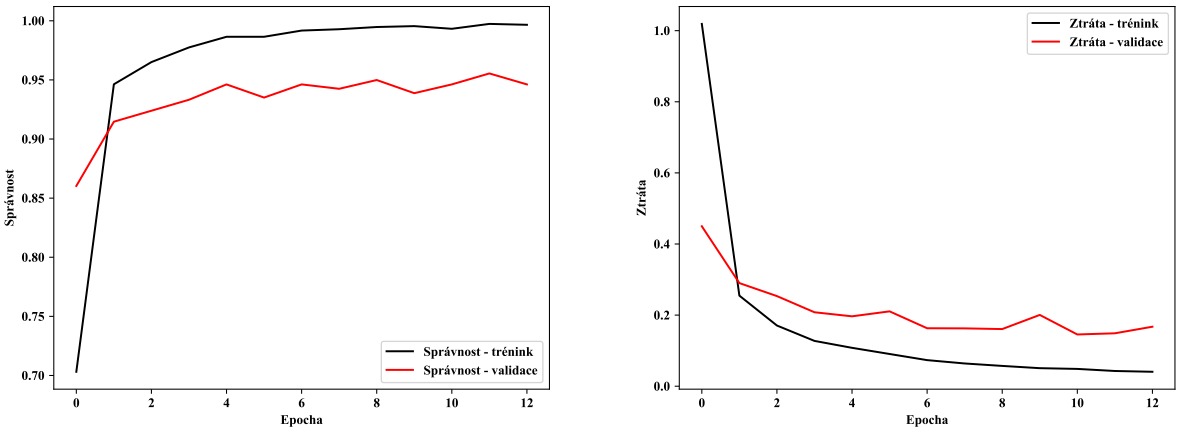
Decay snižuje hodnotu learning rate po každé zpracované dávce, čímž se snižuje riziko, že algoritmus nebude schopen dosáhnout lokálního optima. Algoritmus využívající dynamickou hodnotu rychlosti učení by měl zpravidla konvergovat mnohem rychleji než algoritmus se špatně zvolenou fixní hodnotou tohoto hyperparametru (Reed et al., 1999).

Parametr momentum, neboli setrvačnost, dokáže urychlit proces optimalizace modelu a zvyšuje pravděpodobnost, že budou nalezeny lepší hodnoty vah v menším počtu epoch.

Nově má tedy instance SGD následující formu:

```
gradient_descent = SGD(
        lr = LEARNING_RATE,
        decay = 0.0001,
        momentum = 0.9)
```

Takto upravený model dosahuje správnosti 94,62%, což je jen o 0,38% méně než je cíl této práce. Z grafu je zřejmý jeden problém: Křivka správnosti tréninku je výrazně výše než křivka správnosti validace, což může naznačovat problém přeučení (viz obrázek 4.4).



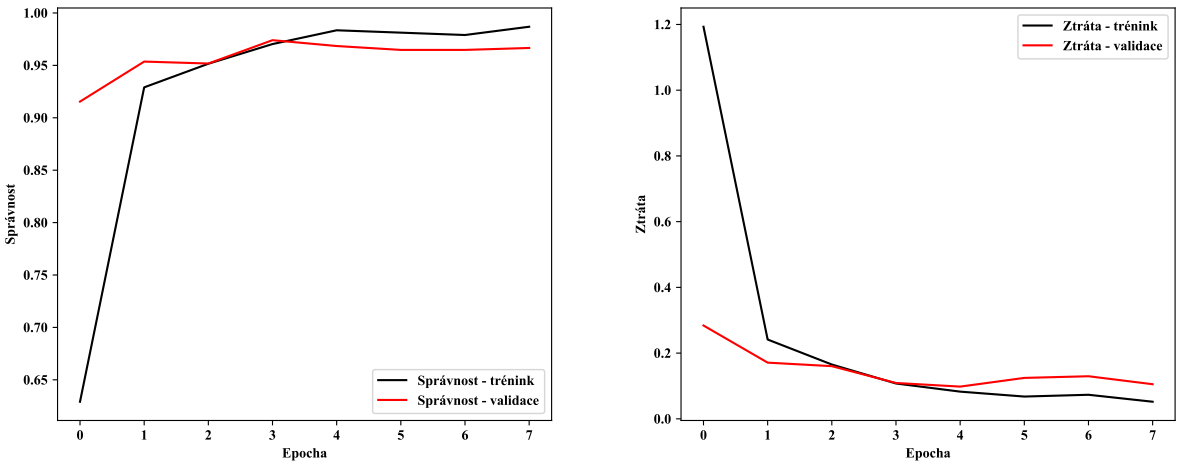
Obrázek 4.4: Experiment č. 4 - Správnost a Ztráta

Nejjednodušším řešením problému přeučení je v tomto případě vytvoření nové výpadkové (dropout) vrstvy. Výpadková vrstva v rámci tréninku deaktivuje náhodně vybrané neurony, čímž je zlepšena schopnost generalizace modelu a mělo by také dojít k minimalizaci přeučení (N. Srivastava et al., 2014).

Výpadkovou vrstvu přidáme před plně propojenou výstupní vrstvu:

```
1 base_tensor = base_model.layers[-6].output
2 output_tensor = Dropout(0.5)(base_tensor)
3 output_tensor = Dense(11, activation='softmax')(output_tensor)
```

Zároveň byl změněn i počet trénovaných vrstev z původních 6 na 24. To by se mělo odrazit na zlepšení trénovaného modelu.



Obrázek 4.5: Experiment č. 5 - Správnost a Ztráta

Tato verze modelu dosahuje celkové správnosti 97,77% na testovací datové množině a to již po osmi epochách. Došlo tedy nejen k významnému zlepšení modelu, ale i ke zvýšení

efektivity, kdy pro dosažení daného výsledku bylo zapotřebí o 40% méně epoch než v minulém experimentu.

Sblížení testovacích a validačních křivek také potvrdilo, že v předchozím případě se opravdu vyskytoval problém přeučení, který po zapojení výpadkové vrstvy zmizel.

Vzhledem k tomu, že bylo dosaženo cíle práce a další experimenty s nastavením hyperparametrů již nevedely ke zvýšení celkové správnosti, je možné model č.5 označit za finální.

Model je uložen ve formátu .h5 jako externí příloha této práce v informačním systému Vysoké školy ekonomické v Praze. Je možné ho stáhnout a dále s ním libovolně pracovat. Inicializace uloženého modelu probíhá pomocí příkazu:

```
keras.models.load_model('final_model.h5')
```

4.6 Vyhodnocení modelu

Finální verze modelu dosáhla celkové správnosti 97,77%, čímž došlo k naplnění vytyčeného cíle této práce.

Správnost není jediná zajímavá metrika v kontextu klasifikačních úloh. Model dosáhl zároveň i průměrně přesnosti 96,18%, průměrné úplnosti 95,90% a průměrného f1-score 95,90%. Model se tedy vyznačuje veskrze dobrým a vyrovnaným výkonem napříč všemi základními metrikami (podrobný výpis viz tabulka 4.1).

Tabulka 4.1: Přesnost a úplnost dle jednotlivých tříd

Index třídy	**Název třídy**	**Přesnost**	**Úplnost**	**F1-score**
0	Danaus plexippus	93%	98%	95%
1	Gonepteryx rhamni	95%	98%	96%
2	Inachis io	96%	98%	97%
3	Limentis arthemis astyanax	98%	98%	98%
4	Limentis camilla	98%	96%	97%
5	Lycaena icarus	98%	98%	98%
6	Lycaena phlaeas	94%	96%	95%
7	Nymphalis antiopa	90%	98%	94%
8	Pieris rapae	98%	94%	96%
9	Polygonia comma	100%	87%	93%
10	Vanessa atalanta	98%	94%	96%
x	Průměr	96,18%	95,90%	95,90%

Ačkoliv již máme hodnoty agregovaných metrik, zajímají nás i konkrétní příklady, kdy síť obrázek špatně klasifikovala. Z nich lze často získat velmi zajímavé informace o chování modelu.

První skupinou nesprávně klasifikovaných případů jsou chybně označené vstupní obrázky. Výchozí data byla totiž anotována lidmi, a proto se chybám takřka nedá vyhnout. Síť odhalila dva případy, kdy byl motýl již na vstupu zařazen do špatné třídy.

V dalším případě se na obrázku, který byl součástí třídy *Vanessa atalanta* (Babočka admirál), vyskytoval jak zástupce *Inachis io* (Babočka paví oko), tak i zástupce *Vanessa atalanta* (viz obrázek 4.6). Jelikož je náš model pouze jednoduchým klasifikátorem, nedokáže si s touto situací poradit a každý obrázek přiřadí pouze k jedné cílové třídě.

Obrázek 4.6: Špatně klasifikovaný případ č.1

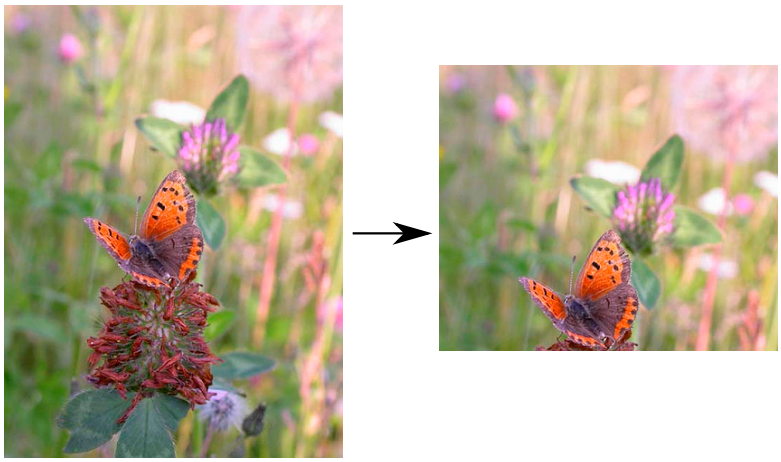
Pro zajímavost je uveden i výpis pravděpodobnostního rozložení pro výše uvedený příklad:

```
> inachisio 0.9930345
> vanessaatalanta 0.002110848
> polygoniacomma 0.0014958379
> gonepteryxrhamni 0.0014707124
> lycaenaphlaeas 0.00071836315
> ...
```

Vzhledem k výskytu dvou motýlů na vstupním obrázku bylo očekáváno, že pravděpodobnost bude rozložena mezi třídu *Inachis io* a *Vanessa atalanta*, proto je poměrně překvapivé, že třídě *Vanessa atalanta* připadá pouze $P = 0{,}211\%$ (v obrázku 4.6 je nahoře).

Po bližším zkoumání obrázku je u zástupce třídy *Vanessa atalanta* zřejmý zásadní problém: Křídla motýla v podstatě splývají s květinou na pozadí, čímž se stává pro síť „neviditelným“. Naopak u zástupce *Inachis io* je kontura křídla zřejmá, což v kombinaci s výrazným zabarvením ústí v pravděpodobnost $P = 99{,}3\%$ pro tuto třídu.

Dalším špatně klasifikovaným případem je fotografie zástupce *Lycaena phlaeas* (Ohniváček černokřídlý), který byl špatně označen jako *Polygonia comma* s $P = 73{,}8\%$. Motýl je na fotografii sice jasně viditelný, je zde ale přítomna také rostlina s velmi výraznou strukturou (viz obrázek 4.7), která je s největší pravděpodobností příčinou nesprávné klasifikace. Po oříznutí rostliny jíž model klasifikuje motýla korektně s pravděpodobností $P = 99{,}2\%$.

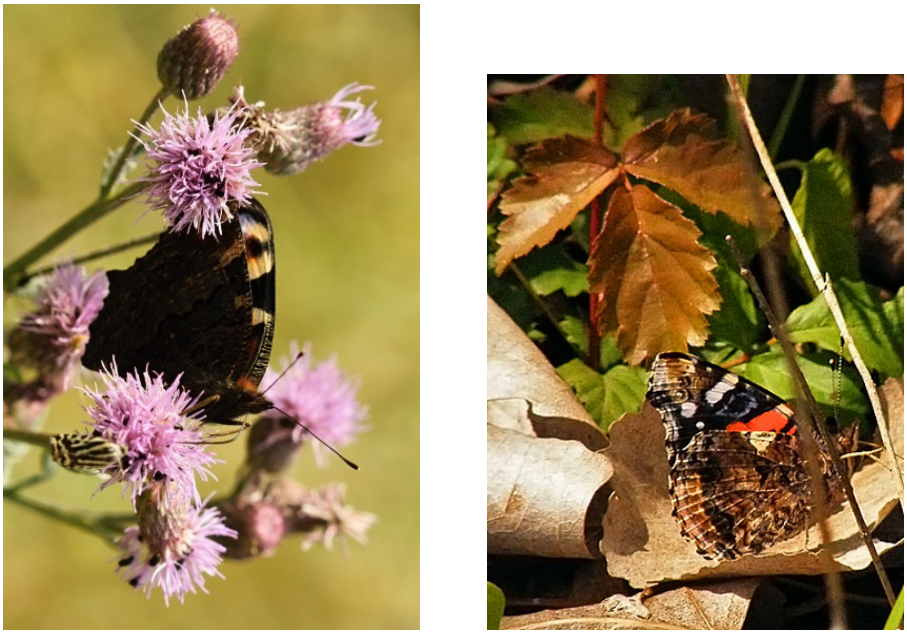


Obrázek 4.7: Špatně klasifikovaný případ č.2 před úpravou a po úpravě

Posledním zajímavým případem je špatná klasifikace zástupce *Danaus plexippus* (Monarcha stěhovavý). V tomto případě byl motýl na obrázku vzhůru nohama (viz obrázek 4.8). Po vertikálním převrácení obrázku ho již síť klasifikovala správně. Dá se tedy předpokládat, že kdyby na vstupní data byly aplikovány během tréninku grafické transformace, jako je náhodné vertikální a horizontální převrácení, mohla by se celková správnost modelu ještě zvýšit.

Obrázek 4.8: Špatně klasifikovaný případ č.3

V kontrastu s předchozími řádky uveďme i dva případy, kdy síť správně klasifikovala relativně obtížná vstupní data. Na obou obrázcích je vidět pouze malá část barevného vzoru křídel motýla a zároveň se na pozadí vyskytují listy a květiny. Přesto byly tyto případy klasifikovány správně s pravděpodobností $P = 72{,}1\%$ (obrázek 4.9 - a) a $P = 97{,}5\%$ (obrázek 4.9 - b).



(a) *Inachis io* (b) *Vanessa atalanta*

Obrázek 4.9: Ukázka správně klasifikovaných případů

Závěr

Strojové učení a umělá inteligence se s největší pravděpodobností stanou jedním z hlavních katalyzátorů vývoje moderní společnosti a to nejenom technologického, ale i politického a sociálního.

Aplikace strojového učení mají potenciál bezprecedentně zvýšit celkový komfort lidstva. Tesla, Uber a další společnosti jsou v dnešní době schopné vytvořit semi-autonomní automobily a k plně autonomnímu řízení se blížíme každým dnem. Každý meč má ale dvě ostří a již nyní jsme svědky zneužití technologií umělé inteligence politickým režimem v Číně a nelze s jistotou tvrdit, že se podobného vývoje časem nedočkáme i v Evropě, Americe a zbytku světa.

Cílem této práce bylo vytvořit model konvoluční neuronové sítě metodou přenosu učení, který bude schopen klasifikovat jedenáct vybraných druhů motýlů s celkovou správností $\geq 95\%$. Tento cíl byl splněn, jelikož výsledný model dosáhl celkové správnosti 97,77% na testovací datové množině.

Cesta k tomuto výsledku ovšem nebyla zcela přímá, poněvadž tvorba modelu neuronové sítě je vždy záležitostí experimentální a neexistují žádné exaktní postupy, které by garantovaly dosažení optimálních výsledků. Existují pouze praxí ověřená doporučení a praktiky. Pro splnění vytyčeného cíle bylo nutné překonat několik problémů, zejména pak neúčinnost a neefektivitu učícího algoritmu pro zvolená nastavení hyperparametrů. Tyto problémy se povedlo vyřešit po konzultaci s odbornou literaturou. V rámci několika experimentů se také vyskytl problém přeučení, který byl odstraněn pomocí techniky regularizace.

Závěrečné vyhodnocení modelu odhalilo několik nedostatků: Vzhledem k tomu, že na vstupní data nebyly použity náhodné grafické transformace, má výsledný model problém správně klasifikovat případy, kdy je motýl na obrázku vertikálně převrácen. Z použité architektury vyplývá, že model není schopen klasifikovat více motýlů na jednom vstupním obrázku.

Jelikož byl model vytvořen na základě konvoluční neuronové sítě MobileNet, jež je optimalizovaná pro použití na mobilních zařízeních, dovedu si představit rozvinutí této práce v podobě mobilní aplikace. Pokud by byl zároveň vhodně rozšířen vstupní datový korpus, mohla by být nově vytvořená aplikace vhodným doplňkem pro výstavy motýlů, které koná kupříkladu Botanická zahrada Praha, ale určitě i další instituce po celém světě.

Další možný rozvoj vidím v použití složitější modelové architektury, díky které by síť nejdříve zjistila, zda se na vstupním obrázku nachází jeden, či více motýlů a ty by následně samostatně klasifikovala. Předpokládám, že toto by jednak zvýšilo výkonnost modelu, a zároveň by se zvýšila i jeho praktická využitelnost. Takový model by byl schopen správně klasifikovat i takové obrázky, které obsahují více motýlů, což stávající model nedokáže.

Seznam použité literatury

ABADI, Martin et al., 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR*. Roč. abs/1603.04467. Dostupné z arXiv: 1603.04467.

ANDERSON, James, 1995. *An introduction to neural networks*. Cambridge, Mass: MIT Press. ISBN 9780262011440.

BENGIO, Yoshua, 2012. Practical recommendations for gradient-based training of deep architectures. *CoRR*. Dostupné z arXiv: http://arxiv.org/abs/1206.5533v2 [cs.LG].

BURKOV, Andriy, 2019. *The hundred-page machine learning book*. Quebec, Canada: Andriy Burkov. ISBN 978-1999579500.

CAPINERA, John, 2008. *Encyclopedia of entomology*. Dordrecht London: Springer. ISBN 9781402062421.

DENG, J.; DONG, W.; SOCHER, R.; LI, L.-J.; LI, K.; FEI-FEI, L., 2009. ImageNet: A Large-Scale Hierarchical Image Database. In: *ImageNet: A Large-Scale Hierarchical Image Database. CVPR09*.

DONAHUE, Jeff; JIA, Yangqing; VINYALS, Oriol; HOFFMAN, Judy; ZHANG, Ning; TZENG, Eric; DARRELL, Trevor, 2013. DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition. *CoRR*. Roč. abs/1310.1531. Dostupné z arXiv: 1310.1531.

GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron, 2016. *Deep Learning*. MIT Press. http://www.deeplearningbook.org.

HAYKIN, Simon, 1999. *Neural Networks : A Comprehensive Foundation*. Upper Saddle River, N.J: Prentice Hall. ISBN 0132733501.

HECHT-NIELSEN, Robert, 1992. Theory of the Backpropagation Neural Network. In: *Theory of the Backpropagation Neural Network. Neural networks for perception*. Elsevier, s. 65–93.

HOWARD, Andrew G.; ZHU, Menglong; CHEN, Bo; KALENICHENKO, Dmitry; WANG, Weijun; WEYAND, Tobias; ANDREETTO, Marco; ADAM, Hartwig, 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. *CoRR*. Dostupné z arXiv: 1704.04861.

HUNTER, J. D., 2007. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*. Roč. 9, č. 3, s. 90–95. Dostupné z DOI: 10.1109/MCSE.2007.55.

CHOLLET, Francois et al., 2015. *Keras* [https://github.com/fchollet/keras]. GitHub.

KESKAR, Nitish Shirish; MUDIGERE, Dheevatsa; NOCEDAL, Jorge; SMELYANSKIY, Mikhail; TANG, Ping Tak Peter, 2016. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. *CoRR*. Dostupné z arXiv: 1609.04836.

KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E., 2012. ImageNet Classification with Deep Convolutional Neural Networks. In: *ImageNet Classification with Deep Convolutional Neural Networks. Advances in neural information processing systems*, s. 1097–1105.

MANYIKA, James; LUND, Susan; CHUI, Michael; BUGHIN, Jacques; WOETZEL, Jonathan; BATRA, Parul; KO, Ryan; SANGHVI, Saurabh, 2017. *Jobs lost, jobs gained: workforce transitions in a time of automation*. [San Francisco]: McKinsey Global Institute. Dostupné také z: `https://www.mckinsey.com/global-themes/future-of-organizations-and-work/what-the-future-of-work-will-mean-for-jobs-skills-and-wages`.

MARSLAND, Stephen, 2009. *Machine learning : an algorithmic perspective*. Boca Raton: CRC Press. ISBN 1420067184.

MCCULLOCH, Warren S.; PITTS, Walter, 1943. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*. Roč. 5, č. 4, s. 115–133.

MITCHELL, Tom, 1997. *Machine Learning*. New York: McGraw-Hill. ISBN 978-0-07-042807-2.

NAIR, Vinod; HINTON, Geoffrey E., 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In: *Rectified Linear Units Improve Restricted Boltzmann Machines. Proceedings of the 27th international conference on machine learning (ICML-10)*, s. 807–814.

NG, Andrew Y.; COATES, Adam; DIEL, Mark; GANAPATHI, Varun; SCHULTE, Jamie; TSE, Ben; BERGER, Eric; LIANG, Eric, 2006. Autonomous Inverted Helicopter Flight via Reinforcement Learning. In: ANG, Marcelo H.; KHATIB, Oussama (ed.). *Experimental Robotics IX*. Berlin, Heidelberg: Springer Berlin Heidelberg, s. 363–372. ISBN 978-3-540-33014-1.

REED, Russell; MARKSII, Robert J., 1999. *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*. A Bradford Book. ISBN 0262527014.

ROSENBLATT, F., 1958. The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain. *Psychological Review*. Roč. 65, č. 6, s. 386–408. Dostupné z DOI: `10.1037/h0042519`.

RUDER, Sebastian, 2016. An overview of gradient descent optimization algorithms. *CoRR*. Roč. abs/1609.04747. Dostupné z arXiv: `1609.04747`.

RUSU, Andrei A.; VECERIK, Matej; ROTHÖRL, Thomas; HEESS, Nicolas; PASCANU, Razvan; HADSELL, Raia, 2016. Sim-to-Real Robot Learning from Pixels with Progressive Nets. *CoRR*. Dostupné z arXiv: `1610.04286`.

SAMUEL, A. L., 1959. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*. Roč. 3, č. 3, s. 210–229. ISSN 0018-8646. Dostupné z DOI: `10.1147/rd.33.0210`.

SMITH, Samuel L.; KINDERMANS, Pieter-Jan; YING, Chris; LE, Quoc V., 2017. Don't Decay the Learning Rate, Increase the Batch Size. *CoRR*. Dostupné z arXiv: `http://arxiv.org/abs/1711.00489v2 [cs.LG]`.

SRIVASTAVA, Nitish; HINTON, Geoffrey; KRIZHEVSKY, Alex; SUTSKEVER, Ilya; SALAKHUTDINOV, Ruslan, 2014. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *The Journal of Machine Learning Research*. Roč. 15, č. 1, s. 1929–1958.

SRIVASTAVA, Rupesh K.; GREFF, Klaus; SCHMIDHUBER, Jurgen, 2015. Training Very Deep Networks. In: CORTES, C.; LAWRENCE, N. D.; LEE, D. D.; SUGIYAMA, M.; GARNETT, R. (ed.). *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc. Dostupné také z: `http://papers.nips.cc/paper/5850-training-very-deep-networks.pdf`.

SZEGEDY, Christian; VANHOUCKE, Vincent; IOFFE, Sergey; SHLENS, Jon; WOJNA, Zbigniew, 2016. Rethinking the Inception Architecture for Computer Vision. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. ISBN 9781467388511. Dostupné z DOI: `10.1109/cvpr.2016.308`.

WANG, Josiah; MARKERT, Katja; EVERINGHAM, Mark, 2009. Learning Models for Object Recognition from Natural Language Descriptions. In: *Learning Models for Object Recognition from Natural Language Descriptions. Proceedings of the British Machine Vision Conference*.

YOSINSKI, Jason; CLUNE, Jeff; BENGIO, Yoshua; LIPSON, Hod, 2014. How transferable are features in deep neural networks? *CoRR*. Roč. abs/1411.1792. Dostupné z arXiv: `1411.1792`.

Přílohy

Příloha A: Hlavní skript implementace

```
# -*- coding: utf-8 -*-
"""
@author: Martin Petráň
"""

import keras
import numpy as np

from keras.applications.mobilenet import MobileNet, preprocess_input
from keras.layers import Dense, Dropout
from keras.optimizers import SGD
from keras.models import Model
from keras.callbacks import EarlyStopping
from keras.preprocessing.image import ImageDataGenerator
from plot_functions import plot_history
from sklearn.metrics import classification_report

# GlOBÁLNÍ NASTAVENÍ

TRAIN_PATH = 'dataset_FINAL/train'
TEST_PATH = 'dataset_FINAL/test'
VALIDATION_PATH = 'dataset_FINAL/validation'

BATCH_SIZE = 32
LEARNING_RATE = 0.001
EPOCHS_NUMBER = 100
TRAINABLE_LAYERS = 24

# Rozdělení dat na dávky

img_generator = ImageDataGenerator(
        preprocessing_function = preprocess_input)

train_data = img_generator.flow_from_directory(
        TRAIN_PATH,
        target_size = (224,224),
        batch_size = BATCH_SIZE,
        color_mode = 'rgb',
        class_mode = 'categorical',
        shuffle = True)

```

```
validation_data = img_generator.flow_from_directory(
        VALIDATION_PATH,
        target_size = (224,224),
        batch_size = BATCH_SIZE,
        color_mode = 'rgb',
        class_mode = 'categorical',
        shuffle = True)

test_data = img_generator.flow_from_directory(
        TEST_PATH,
        target_size = (224,224),
        batch_size = 1,
        color_mode = 'rgb',
        class_mode = 'categorical',
        shuffle = False)

training_step_size = train_data.samples // BATCH_SIZE
validation_step_size = validation_data.samples // BATCH_SIZE

# Nová instance MobileNet
base_model = MobileNet(
        weights = 'imagenet',
        include_top = True)

# Definice nového modelu
base_tensor = base_model.layers[-6].output
output_tensor = Dropout(0.5)(base_tensor)
output_tensor = Dense(11, activation='softmax')(output_tensor)

model = Model(
        inputs = base_model.input,
        outputs = output_tensor)

# Zmrazení vrstev
for layer in model.layers[:-TRAINABLE_LAYERS]:
    layer.trainable = False

# Incializace SGD
gradient_descent = SGD(
        lr = LEARNING_RATE,
        decay = 0.0001,
        momentum = 0.9)

# Konfigurace modelu
model.compile(
        optimizer = gradient_descent,
        loss = 'categorical_crossentropy',
        metrics = ['accuracy'])

```

```
early_stop = EarlyStopping(
        monitor = 'val_loss',
        min_delta = 0,
        patience = 3,
        verbose = 0,
        mode = 'auto')

# Trénink modelu
history = model.fit_generator(
        generator = train_data,
        steps_per_epoch = training_step_size,
        validation_data = validation_data,
        validation_steps = validation_step_size,
        epochs = EPOCHS_NUMBER,
        callbacks = [early_stop],
        verbose = 2)

plot_history(history)

# Uložení / načtení modelu
model.save('final_model.h5')
#model = keras.models.load_model('final_model.h5')

# Závěrečná evaluace na testovacích datech
test_accuracy = model.evaluate_generator(
        generator = test_data,
        steps = 583,
        verbose = 1)

print(test_accuracy)

# Získáni predikcí a vytisknutí reportu
fnames = test_data.filenames
ground_truth = test_data.classes
label2index = test_data.class_indices
class_labels = list(test_data.class_indices.keys())

predictions = model.predict_generator(
        test_data,
        steps = 583,
        verbose = 1)

y_pred = np.argmax(predictions, axis=1)

report = classification_report(ground_truth, y_pred, target_names=class_labels)
print(report)
```

Příloha B: Tvorba grafu

```python
# -*- coding: utf-8 -*-
"""
@author: Martin Petráň
"""
import matplotlib.pyplot as plt

plt.rcParams["font.family"] = "Times New Roman"

def plot_history(history):
    acc = history.history['acc']
    val_acc = history.history['val_acc']
    loss = history.history['loss']
    val_loss = history.history['val_loss']

    epochs = range(len(acc))

    plt.subplot(1, 2, 1)
    plt.plot(epochs, acc, color = 'black', label = 'Správnost - trénink')
    plt.plot(epochs, val_acc, color = 'red', label = 'Správnost - validace')

    plt.ylabel('Správnost')
    plt.xlabel('Epocha')
    plt.legend()

    plt.subplot(1, 2, 2)
    plt.plot(epochs, loss, color = 'black', label = 'Ztráta - trénink')
    plt.plot(epochs, val_loss, color = 'red', label = 'Ztráta - validace')

    plt.ylabel('Ztráta')
    plt.xlabel('Epocha')
    plt.legend()

    plt.tight_layout()
    plt.show()
    plt.savefig('plot.pdf')
```