VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

WEBOVÉ ROZHRANÍ PRO TRÉNOVÁNÍ NEURONOVÝCH SÍTÍ

A WEB INTERFACE FOR TRAINING NEURAL NETWORKS

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — Bc. Jaromír Szymutko

AUTHOR

VEDOUCÍ PRÁCE — Ing. Pavel Sikora

SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Telekomunikační a informační technika**

Ústav telekomunikací

Student: Bc. Jaromír Szymutko ***ID:*** 211273

Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Webové rozhraní pro trénování neuronových sítí

POKYNY PRO VYPRACOVÁNÍ:

Student nastuduje a teoreticky popíše vývojové prostředky k vytváření webových aplikací a ke trénování neuronových sítí za účelem zpracování digitálních obrazů. Pomocí vybraných vývojových prostředků vytvoří web, který uživateli umožní vzdálené načtení databáze obrazů s anotacemi, návrh a natrénování modelu neuronové sítě a uložení výsledků, včetně natrénovaného modelu. Web bude umožňovat také zpracování libovolného obrazu natrénovaným modelem neuronové sítě.

DOPORUČENÁ LITERATURA:

Podle pokynů vedoucího práce.

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 19.5.2023

Vedoucí práce: Ing. Pavel Sikora

prof. Ing. Jiří Mišurec, CSc.
předseda rady studijního programu

ABSTRAKT

Tato práce se zabývá vývojem webové aplikace pro trénování neuronových sítí. V úvodní části práce je přiblížena problematika související s umělou inteligencí a strojovým učením. Dále jsou v práci popsány hlavní nástroje použité při vývoji aplikace, mezi něž patří především technologie Blazor a Keras.NET. Práce následně popisuje postup vývoje webové aplikace s databází, která umožňuje nahrání datasetu, tvorbu modelu neuronové sítě, natrénování modelu a provedení predikce modelem na uživatelem zvolených obrázcích. Následně je představen výsledný vzhled a fungování aplikace.

KLÍČOVÁ SLOVA

ASP.NET, Blazor, C#, EF Core, Keras, Keras.NET, neuronové sítě, strojové učení, umělá inteligence, webová aplikace, .NET

ABSTRACT

The thesis deals with the development of a web application for neural networks training. The introduction of the thesis outlines issues related to artificial intelligence and machine learning. Then the main tools used in the development of the application are described, including the Blazor and Keras.NET. The thesis describes the development process of a web application with a database that allows uploading datasets, creating a neural network model, training the model, and performing predictions on user-selected images. Finally, the resulting appearance and functionality of the application are presented.

KEYWORDS

artficial intelligence, ASP.NET, Blazor, C#, EF Core, Keras, Keras.NET, machine learning, neural networks, web application, .NET

SZYMUTKO, Jaromír. *Webové rozhraní pro trénování neuronových sítí*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2022, 72 s. Diplomová práce. Vedoucí práce: Ing. Pavel Sikora

Prohlášení autora o původnosti díla

Jméno a příjmení autora:	Bc. Jaromír Szymutko
VUT ID autora:	211273
Typ práce:	Diplomová práce
Akademický rok:	2022/23
Téma závěrečné práce:	Webové rozhraní pro trénování neuronových sítí

Prohlašuji, že svou závěrečnou práci jsem vypracoval samostatně pod vedením vedoucí/ho závěrečné práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené závěrečné práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podepisuje pouze v tištěné verzi.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing. Pavlu Sikorovi za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.
Dále bych také rád poděkoval své rodině a přítelkyni za stálou podporu během celého studia.

Obsah

Seznam obrázků

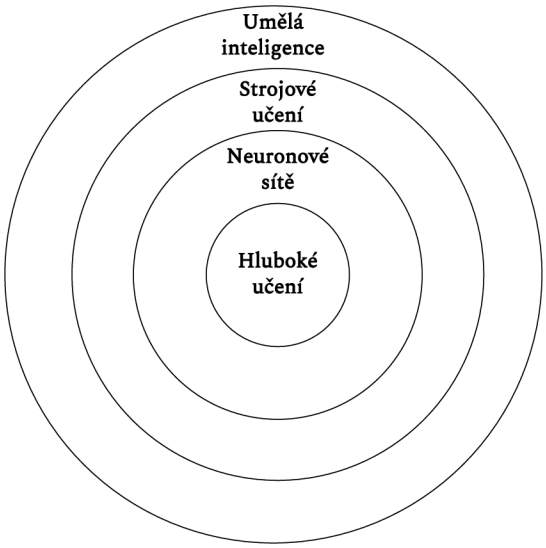
Seznam výpisů

Úvod

Umělá inteligence, strojové učení, neuronové sítě, hluboké učení a mnohé další pojmy, které s touto oblastí souvisí, jsou v posledních letech stále více diskutovány a používány. Všechny tyto technologie jsou poměrně mladé a dynamicky se rozvíjející. Jsou nasazovány v opravdu širokém spektru oborů a stále se objevují nové oblasti, ve kterých je možné jejich nasazení. Aby bylo možné umělou inteligenci nasadit v co nejširším spektru lidských činností, je potřeba co nejvíce zjednodušit případnou implementaci této technologie. Při aplikování umělé inteligence na určitou oblast je totiž potřeba rozumět především dané oblasti a správně interpretovat data, která umělá inteligence zpracovává, a následně také správně interpretovat data, která předloží daná implementace umělé inteligence na svém výstupu. Je tudíž nasnadě vytvořit rozhraní, přes které bude možné používat nástroje umělé inteligence a strojového učení, co nejjednodušší, aby jej mohl využívat odborník v dané oblasti a nepotřeboval k tomu hluboké znalosti v oblasti umělé inteligence či programování. O malé zlepšení v oblasti této problematiky se pokouší i tato práce, jejímž cílem je vytvořit webové rozhraní pro trénování neuronových sítí. Tato aplikace bude mít za úkol co nejvíce zjednodušit práci s hlubokými neuronovými sítěmi, v tomto případě s využitím knihovny Keras, pomocí responzivního a uživatelsky přívětivého webového rozhraní, které ale zároveň bude svůj základ stavět na robustní serverové části, která umožní bezproblémové fungování výpočetně náročného trénování neuronových sítí.

1 Úvod do umělé inteligence a strojového učení

Již v úvodu bylo použito několik pojmů, jako jsou umělá inteligence, strojové učení, neuronové sítě a hluboké učení, které spolu souvisejí a jsou často zaměňovány v běžném hovoru či populárním neodborném článku. Na úvod této práce je tedy vhodné si základní pojmy z této oblasti definovat, rozlišit a popsat vztahy, které mezi nimi jsou. Nejjednodušší je si vztahy mezi těmito pojmy představit podle obrázku 1.1. Z něj vyplývá, že hluboké učení je podmnožinou neuronových sítí, neuronové sítě jsou podmnožinou strojového učení a strojové učení je podmnožinou umělé inteligence [1]. Tyto pojmy budou dále podrobněji vysvětleny v následujících kapitolách.



Obr. 1.1: Nákres vztahů mezi základními pojmy.

1.1 Umělá inteligence

Z definice inteligence vyplývá, že se jedná o schopnost získávat znalosti a následně tyto znalosti aplikovat při řešení problémů. Inteligence poskytuje lidem schopnost profitovat z minulých událostí tím, že na základě zkušeností z těchto událostí jsou schopni řešit problémy v budoucnu. [2]

Se vznikem umělé inteligence se lidstvu podařilo vytvořit mocný nástroj, který mu pomáhá v rozhodování se při řešení nejrůznějších problémů. Samotný termín umělá inteligence (Artificial Intelligence) začal razit v roce 1956 americký počítačový

vědec John McCarthy na konferenci na Darthmouth College. Proto je také považován za otce umělé inteligence. Vědci od té dobry přišli s mnohými definicemi umělé inteligence jako například [2]:

- „Umělá inteligence zkoumá možnosti, jak přimět počítače dělat to, v čem jsou momentálně lidé lepší.“
- „Umělá inteligence je součást počítačových věd, která se zabývá návrhem inteligentních počítačových systémů, tzn. systémů, které vykazují charakteristiky, které jsou spojovány s lidskou inteligencí.“

Mezi hlavní cíle umělé inteligence patří:

- Vytvořit systémy, které budou vykazovat inteligentní chování a schopnosti učit se, vysvětlit a demonstrovat nabyté znalosti.
- Implementovat lidskou inteligenci ve strojích. Stroje budou schopny rozumět, myslet, učit se a chovat se jako lidé.

1.1.1 Výhody umělé inteligence

Výhody umělé inteligence jsou poměrně jasné z její samotné podstaty, protože stroje mají na rozdíl od lidí některé výhodné vlastnosti. Mezi hlavní výhody patří kupříkladu [2]:

- Stroje nepotřebují tak časté přestávky a odpočinek, díky tomu jsou schopny vykonat více práce za daný časový úsek.
- Pomocí strojů můžeme provádět výpočty, které jsou příliš složité a časově náročné pro člověka.
- Stroje, na rozdíl od lidí, nepodléhají stresovým situacím.

1.1.2 Příklady nasazení umělé inteligence

Jak již v úvodu této práce zaznělo, umělá inteligence se používá ve velmi široké škále oborů a odvětví lidské činnosti. Mezi oblasti, ve kterých se umělá inteligence nasazuje, patří například [2]:

- Expertní systémy – Expertní systémy jsou počítačové programy, do kterých je nahrána expertíza a tento program je poté schopen na základě této expertízy pomoci uživateli s rozhodováním o problému v oblasti, kterou pokrývá daná expertíza [3].
- Počítačové vidění – Cílem v této oblasti je, aby počítač byl schopen chápat a interpretovat obrazový vstup. Mezi klasické úlohy počítačového vidění patří klasifikace obrazu, při které počítač určuje, co se na daném obrazovém vstupu nachází, dále také detekce objektů, kdy počítač na základě klasifikace určuje, který objekt se v jaké oblasti obrázku nachází, či trasování pohybu objektu na videu.

- Rozpoznávání řeči – Umělá inteligence je schopná zpracovat zvukovou stopu s lidskou řečí a správně určit význam konkrétního sdělení.
- Medicína – V medicíně obecně je potřeba zpracovávat velká množství dat, která pochází z nejrůznějších měření a vyšetření, jako jsou například počítačová tomografie (CT), magnetická rezonance (MRI) a mnoho dalších. Pro toto zpracování je výhodné nasadit umělou inteligenci, která byla navržena pro zpracování a vyhodnocení velkého objemu dat. Navíc umělá inteligence může najít v datech vzorce, které člověk není schopen detekovat, a může tak vnést nový pohled na problematiku.
- Hraní her – Na řešení her se testoval a stále testuje vývoj umělé inteligence v oblasti prohledávání stavového prostoru. Hry v podstatě představují zjednodušené řešení situací v reálném životě, proto je vhodné je použít při pokusech s implementací umělé inteligence.

1.2 Strojové učení

Strojové učení (anglicky Machine learning) je jednou z podoblastí umělé inteligence. Strojové učení umožňuje umělé inteligenci provádět následující úkony [4]:

- Přizpůsobit se novým podmínkám, které vývojář při implementaci umělé inteligence nepředpokládal.
- Detekovat vzorce v datech.
- Vytvářet nové vzorce na základě detekovaných vzorců.
- Rozhodovat se na základě těchto vzorců chování.

Strojové učení je založeno na algoritmech, které analyzují velké kolekce dat (datasety) a na základě analýzy dokážou vytvořit odhad pro další data. K plnohodnotné umělé inteligenci, která by byla schopna myslet a řešit problémy, mají tyto algoritmy však daleko. Strojové učení však umožňuje lidem pracovat efektivněji. Pomáhá jim při rozhodování tím, že zpracuje analýzu velkého množství dat, která by pro člověka byla velmi náročná a časově neefektivní. Nedokáže ale vyvodit důsledky z této analýzy, to musí opět udělat lidé a na základě analýzy zvážit možné následky jednotlivých rozhodnutí a některé z nich přijmout [4]. Strojové učení je úzce spojeno s výpočetní statistikou, která se také zaměřuje na vytváření predikcí pomocí počítačů. [1]

Samotný proces strojového učení se dělí na 3 hlavní části:

- Předzpracování dat.
- Trénování modelu.
- Evaluace modelu.

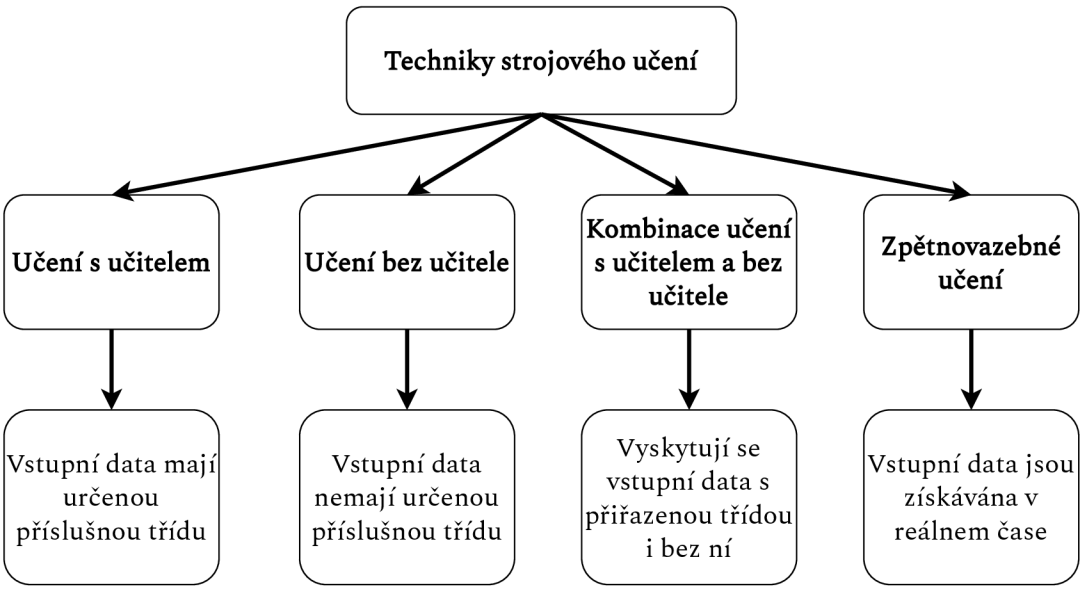
Předzpracování dat slouží k úpravě surových dat do správné podoby. Surová data totiž bývají špatně strukturovaná, nekompletní, nekonzistentní a duplicitní. V pro-

cesu předzpracování dat se provádí operace jako čištění, extrakce, transformace nebo slučování dat. Po dokončení předzpracování jsou data v podobě, ve které je možné je použít jako vstup strojového učení.

Ve fázi trénování modelu dochází k samotnému učení. Jsou vybrány algoritmy, které budou použity, dále jsou poupraveny parametry modelu tak, aby bylo docíleno kýžených výsledků při použití předzpracovaných dat.

Evaluace modelu je poslední z fází strojového učení a posuzuje se při ní kvalita natrénování modelu. Ve fázi evaluace se provádí testování natrénovaného modelu pomocí testovacích dat. Pomocí zvolených evaluačních metrik se vyhodnocuje, jak dobře je model schopen zpracovat data, ke kterým nikdy neměl přístup, na základě jeho natrénování na datech, která mu byla poskytnuta při trénování. V závislosti na výsledcích evaluace se poté přistupuje k případným úpravám parametrů modelu nebo výběru jiných algoritmů tak, aby byla výkonnost modelu, pokud možno, zvýšena. [5]

Rozdělení algoritmů strojového učení je i s poznámkou o povaze vstupních dat znázorněno na obrázku 1.2.



Obr. 1.2: Rozdělení strojového učení podle způsobu učení [6].

1.2.1 Učení s učitelem

Algoritmy strojového učení, které řadíme do skupiny učení s učitelem (anglicky Supervised learning), potřebují ke svému správnému natrénování na vstupu data, která mají přiřazený i správný výstup, tzv. třídu (anglicky label). Jednoduchým příkladem

může být algoritmus, který na obrázku rozpoznává, zda se jedná o kočku, či psa. Pro své natrénování tento algoritmus potřebuje jako vstup obrázky, na kterých budou kočky a psi, označené třídou „kočka“, či„pes“. Na těchto označených trénovacích datech se algoritmus natrénuje a měl by být poté schopen rozpoznat kočku či psa na obrázcích, které do této chvíle neměl k dispozici, a správně jim sám přiřadit danou třídu [4]. Nevýhodou tohoto přístupu je, že při přípravě datasetu, ze kterého se algoritmus má učit, je potřeba ručně přiřadit třídu každému souboru. Vzhledem k tomu, že dataset musí být pokud možno co nejobsáhlejší, aby poskytl modelu dostatek vzorků pro co nejlepší natrénování, je tato práce opravdu časově náročná a od tohoto se poté odvíjí také cena daného datasetu. Algoritmy učení s učitelem se osvědčují především v případech, kdy je cílem získat předpověď do budoucna analýzou historických dat. [1]

1.2.2 Učení bez učitele

Při učení bez učitele (anglicky Unsupervised learning) se algoritmus učí na datech, která nemají přiřazenou třídu. Neexistují tedy správně přiřazené páry vstupů a výstupů, které by už někdo předem vytvořil. Algoritmus tak musí sám najít určitý vzorec ve vstupních datech a zařadit je do správné třídy. [1]

1.2.3 Kombinace učení s učitelem a bez učitele

Anglicky se tento přístup nazývá Semi-supervised learning. Tyto algoritmy využívají jak data, která mají přiřazenou třídu, tak neoznačená data, která třídu přiřazenou nemají. Důvod je čistě praktický. Data, která nemají přiřazenou třídu, jsou výrazně levnější. [1]

1.2.4 Zpětnovazebné učení

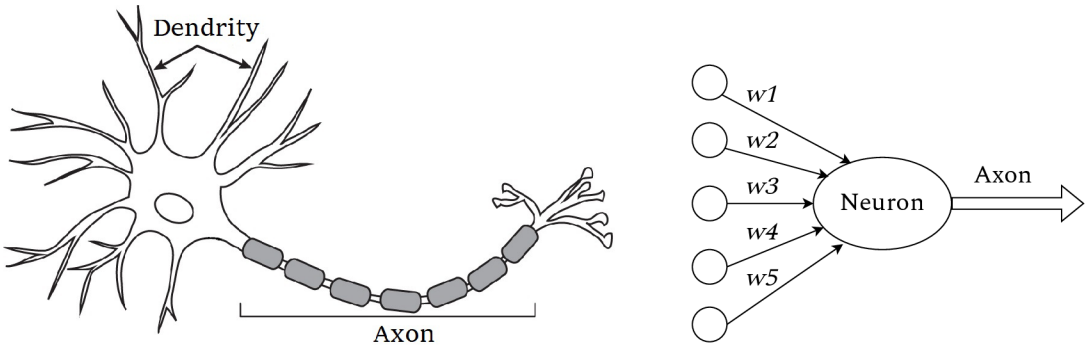
Zpětnovazebné učení je nejčastěji nasazováno v oblasti robotiky a hraní her. Při zpětnovazebném učení algoritmus metodou pokus-omyl zjišťuje, které akce mu přináší největší odměnu. Toto učení má 3 hlavní entity: agent, prostředí a akce. Během tohoto učení agent, který se nachází v jemu neznámém prostředí, provádí akce, které mu přináší určité množství odměn. Cílem agenta je, aby zvolil a provedl akce s maximálním ziskem odměn. [5]

1.3 Umělé neuronové sítě

Umělé neuronové sítě, dále také jen neuronové sítě, jsou populární technologií strojového učení, která simuluje mechanismus učení známý z živých organismů. Základní

součástí lidského nervového systému je buňka zvaná neuron. Neurony jsou mezi sebou vzájemně spojeny axony a dendrity. Tato spojení se nazývají synapse. Synapse se v reakci na vnější podněty často mění a přizpůsobují se těmto podnětům. Tyto změny reprezentují učení v živých organismech.

Umělý neuron je základní součástí umělé neuronové sítě a představuje základní výpočetní jednotku. Jednotlivé umělé neurony jsou vzájemně spojeny pomocí vah, které reprezentují sílu spojení. Každý vstup neuronu má přiřazenou váhu, která ovlivňuje výpočet daného neuronu. Srovnání umělého neuronu, kde jsou jednotlivé váhy označeny písmeny w s příslušným číslem, a fyziologického neuronu je na obrázku 1.3.



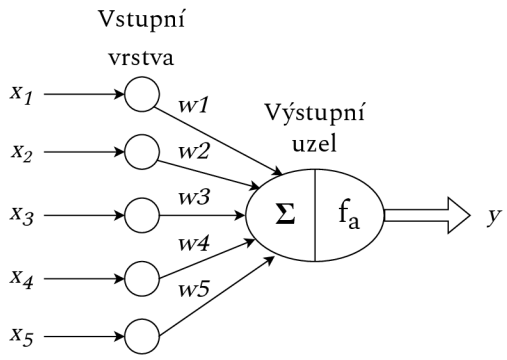
(a) Fyziologický neuron. Převzato z [7], upraveno. (b) Umělý neuron [8].

Obr. 1.3: Fyziologický a umělý neuron.

Umělá neuronová síť provádí výpočet výstupní funkce tak, že předává vypočítané hodnoty ze vstupního neuronu výstupnímu neuronu, přičemž používá váhy jako dočasné parametry. Samotné učení spočívá v přizpůsobování vah mezi neurony. Stejně jako jsou pro učení v živých organismech potřeba vnější stimuly, kterým se přizpůsobí síla synaptického spojení mezi neurony, potřebuje i umělá neuronová síť vnější stimuly. Takovéto stimuly poskytují trénovací data, která mají přiřazené třídy. Trénovací data, u kterých víme, jaký má být výstup pro jednotlivé vstupy, poskytují zpětnou vazbu o správnosti přiřazených vah podle toho, jak dobře dokázala neuronová síť vyřešit daný problém. Cílem je nastavit jednotlivé váhy v umělé neuronové síti tak, aby dokázala co nejpřesněji vypočítat zadanou funkci s minimální chybovostí. Umělé neuronové sítě lze rozdělit podle architektury na jednovrstvé a vícevrstvé. V jednovrstvé umělé neuronové síti je sada vstupů přemapována na výstupy pomocí variací různých lineárních funkcí. Ve vícevrstvých umělých neuronových sítích jsou kromě vstupní a výstupní vrstvy umělé neurony uspořádány do tzv. skrytých vrstev, které se nachází mezi vstupní a výstupní vrstvou. Umělá neuronová síť s touto architekturou se také nazývá hluboká neuronová síť a učení takovéto sítě je označováno jako hluboké učení (anglicky Deep learning). [1]

1.3.1 Perceptron

Perceptron, jehož schéma je na obrázku 1.4, je nejjednodušší neuronová síť, která obsahuje pouze vstupní a výstupní vrstvu. Je tedy výhodné si na perceptronu popsat základní princip fungování neuronových sítí.



Obr. 1.4: Perceptron [8].

Předpokládejme, že každá epocha učení má tvar $(\overline{X}, y)$, kde $\overline{X} = [x_1, ...x_d]$ je množina vstupů, které obsahují příznaky, a $y \in \{-1, +1\}$ je výstupem. Správné výstupy pro jednotlivé vstupy poskytuje trénovací množina dat. Příkladem může být neuronová síť, která provádí detekci objektu v obrazu. V takové aplikaci by v trénovacím datasetu byla množina $\overline{X}$ složena z obrazových dat, ke kterým by byl přiřazen výstup $y \in \{-1, +1\}$, který udává, zda se jedná o detekovaný objekt $(y = +1)$, nebo nikoliv $(y = -1)$. Vstupní vrstva obsahuje d uzlů, které přenášejí příznaky z množiny $\overline{X}$ po hranách s přiřazenými váhami $\overline{W} = [w_1, ...w_d]$ do výstupního uzlu. Vstupní vrstva sama o sobě neprovádí žádné výpočty. Výstupní uzel provádí výpočet lineární funkce $\overline{W}.\overline{X} = \sum_{i=1}^{d} w_i x_i$. Rovnice pro výpočet predikce poté vypadá následovně:

$$\widehat{y} = f_a\left\{\overline{W}.\overline{X}\right\} = f_a\left\{\sum_{i=1}^{d} w_i x_i\right\}, \tag{1.1}$$

kde f_a je vybraná aktivační funkce. V případě výše popsané aplikace se jedná o skokovou funkci, která transformuje vypočtenou hodnotu patřící do oboru reálných čísel na hodnoty +1 nebo -1, které byly zvoleny jako klasifikátory příslušných tříd.

Samotný algoritmus trénování funguje tak, že do sítě jsou na vstup přivedena trénovací data $\overline{X}$ a je sledován výstup y. Následně dochází k úpravám vah na základě chyby, kterou neuronová síť při učení dělá. Výpočet chyby se provádí pomocí chybové funkce (anglicky Loss function). Obecně se chyba vypočítává jako:

$$E(\overline{X}) = (y - \widehat{y}), \tag{1.2}$$

kde y je správný výstup a $\widehat{y}$ je predikce vypočtená algoritmem. Následně dochází úpravám vah na základě chyby podle rovnice 1.3:

$$\overline{W} \Longleftarrow \overline{W} + \alpha(y - \widehat{y})\overline{X}, \tag{1.3}$$

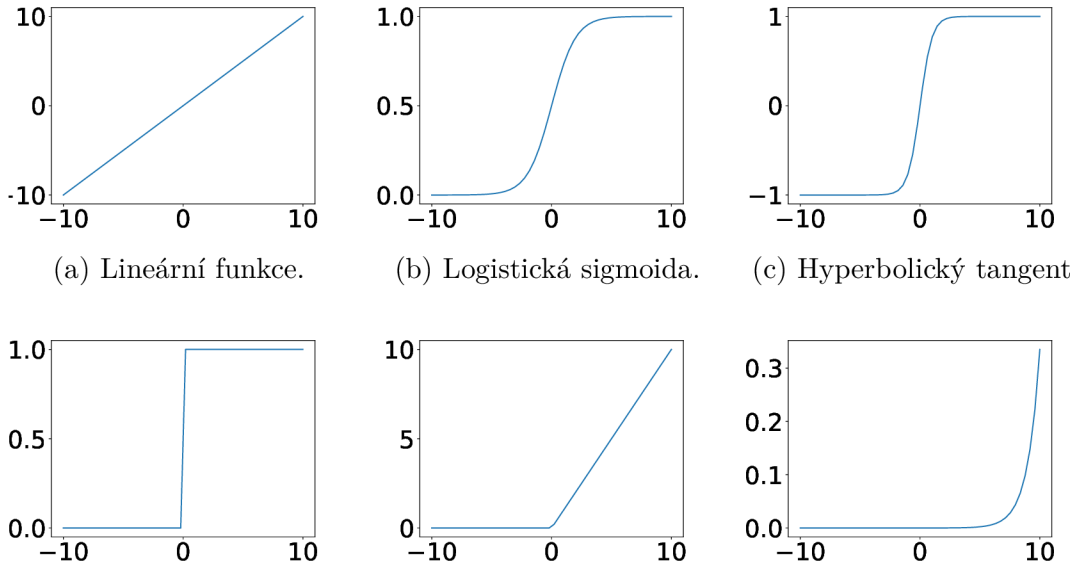
kde parametr α ovlivňuje rychlost učení neuronové sítě. Po dosazení z rovnice 1.2 je možné rovnici 1.3 zapsat jako:

$$\overline{W} \Longleftarrow \overline{W} + \alpha E(\overline{X})\overline{X}. \tag{1.4}$$

Algoritmus perceptronu opakovaně v cyklech prochází vzorky trénovacích dat a mění hodnoty vah, než v ideálním případě dojde ke konvergenci chybové funkce do globálního minima. Takovémuto cyklu se říká také epocha. V praxi může chybová funkce konvergovat i do lokálních minim, u perceptronu například pokud nejsou vstupní data lineárně separovatelná, a učení pak neprobíhá správně. Z rovnice 1.4 je patrné, že pokud je chyba nenulová, tedy predikce neuronové sítě byla chybná, dojde ke změně vah. [8]

1.3.2 Aktivační funkce

Nejčastěji používané aktivační funkce jsou ilustrovány na obrázku 1.5. Výběr aktivační funkce je klíčovou součástí návrhu neuronové sítě.



(a) Lineární funkce. (b) Logistická sigmoida. (c) Hyperbolický tangent.

(d) Skoková funkce. (e) ReLU. (f) Softmax.

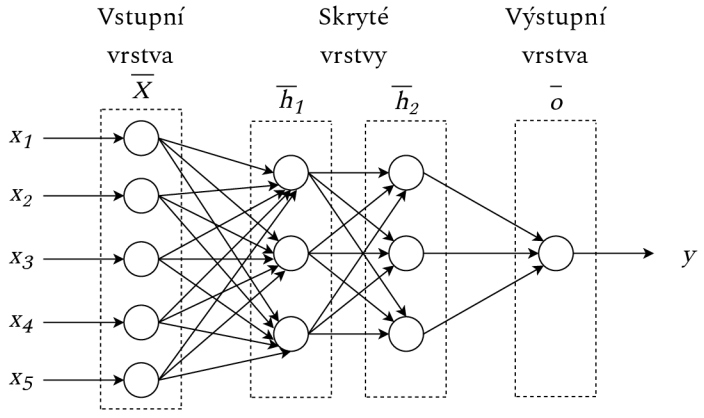
Obr. 1.5: Příklady aktivačních funkcí [8].

V případě perceptronu popsaného výše byla jako aktivační funkce zvolena skoková funkce, protože používá binární klasifikaci. V jiných aplikacích jsou ovšem potřeba jiné klasifikace, a proto je potřeba použít jiné aktivační funkce. Nelineární aktivační funkce nalézají využití především u složitějších vícevrstvých neuronových sítí. [8]

1.4 Hluboké učení

Jako hluboké učení označujeme učení umělých neuronových sítí, které obsahují více než jednu skrytou vrstvu. Tyto sítě se také označují jako hluboké neuronové sítě. Hluboké neuronové sítě využívají strukturu mnoha skrytých vrstev pro extrakci a transformaci příznaků z dat. Každá z vrstev přijímá data z předchozí vrstvy, provádí výpočty a vypočtené hodnoty předává následující vrstvě. Učící algoritmus rozděluje příznaky do několika úrovní, kdy jsou příznaky vyšší úrovně odvozovány z příznaků nižší úrovně a vytváří se tak hierarchická struktura příznaků. Hluboké učení využívá různých způsobů reprezentace dat. Obrázek může být například reprezentován jako vektor hodnot, které reprezentují vlastnosti v jednotlivých pixelech, nebo jako množina hran, které se na obrázku vyskytují, případně jako oblasti s různými tvary a mnoho dalších. Tyto různé reprezentace jsou výhodné v různých aplikacích a mohou poskytovat různé příznaky pro hlubokou neuronovou síť. [1]

Základním typem hlubokých neuronových sítí jsou dopředné sítě. Jednotlivé vrstvy těchto sítí si předávají hodnoty směrem od vstupní vrstvy k výstupní vrstvě. Základní architektura dopředných neuronových sítí předpokládá, že všechny neurony v jedné vrstvě jsou připojeny ke všem neuronům v následující vrstvě. Nákres dopředné neuronové sítě se dvěma skrytými vrstvami je na obrázku 1.6.



Obr. 1.6: Neuronová síť se dvěma skrytými vrstvami [8].

1.4.1 Princip fungování hlubokých neuronových sítí

Obecný princip fungování hlubokých neuronových sítí je postaven na základě jednoduchých neuronových sítí, které byly vysvětleny v kapitole 1.3.1. Váhy spojení mezi vstupní vrstvou a první skrytou vrstvou jsou uloženy v matici W_1, která má rozměry $d \times p_1$, kde d je počet neuronů ve vstupní vrstvě a p_1 je počet neuronů v první skryté vrstvě. Obecně váhy mezi r-tou skrytou vrstvou a vrstvou $r+1$ jsou v matici rozměrů $p_r \times p_{r+1}$, která bude označena jako W_r. Vektor vstupních hodnot $\overline{X}$ bude zpracován následujícím způsobem:

$$\overline{h}_1 = f_a(W_1^T \overline{X}), \tag{1.5}$$

kde $\overline{h_1}$ je výstup vstupní vrstvy a zároveň vstup první skryté vrstvy a f_a je vybraná aktivační funkce. Mezi skrytými vrstvami má výpočet výstupu následující podobu:

$$\overline{h}_{p+1} = f_a(W_{p+1}^T \overline{h}_p), \tag{1.6}$$

kde $\overline{h}_{p+1}$ je matice hodnot vypočtená v následující vrstvě, f_a je zvolená aktivační funkce, W_{p+1}^T je transponovaná matice vah, která spojuje předchozí vrstvu p s vrstvou $p+1$, a $\overline{h}_p$ je matice hodnot vypočtených v předchozí vrstvě. Výpočet matice hodnot ve výstupní vrstvě je proveden jako:

$$\overline{o} = f_a(W_{k+1}^T \overline{h}_k), \tag{1.7}$$

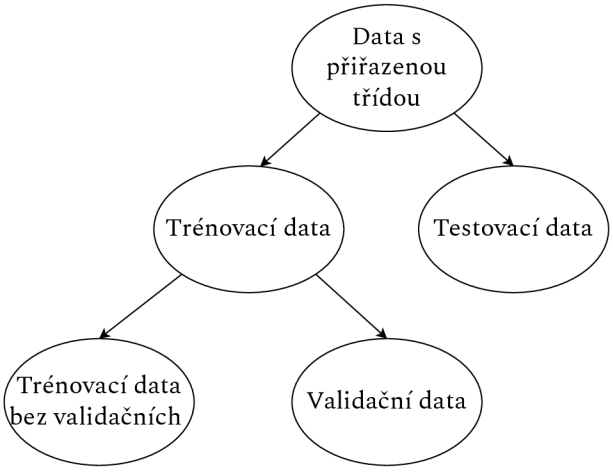
kde $\overline{o}$ je matice hodnot vypočtených ve výstupní vrstvě, f_a je zvolená aktivační funkce, W_{k+1}^T je matice vah, které spojují předchozí skrytou vrstvu a výstupní vrstvu a $\overline{h}_k$ je matice hodnot vypočtených v poslední skryté vrstvě. [8]

1.4.2 Problematika generalizace modelu

Neuronová síť jako celek bývá také označována jako model. Aby byl model schopen správně predikovat výstupy, je potřeba jej nejprve natrénovat, trénování validovat a následně model otestovat. Trénování modelu probíhá přiváděním trénovacích dat na vstup modelu. Tato trénovací data nelze použít pro další kroky - validaci a testování. Cílem je totiž co nejvíce model generalizovat. Dobře generalizovaný model je schopen správně reagovat na neznámá data, s nimiž dosud nepřišel do kontaktu. Pokud by pro trénování a testování byla použita ta stejná data, model by se naučil perfektně predikovat na tomto datasetu, na jiném datasetu by však jeho výsledky byly jen velmi málo přesné. Tomuto jevu se říká přeučení modelu. Rozdělení dat potřebných pro správné fungování neuronových sítí je na obrázku 1.7. Tato data se dělí na:

- Trénovací data – Na těchto datech se model učí, tzn. upravuje váhy v jednotlivých vrstvách.

- Validační data – Tato část dat se používá pro první ověření správnosti natrénování modelu. Na základě tohoto ověření je možné přistoupit k úpravě parametrů modelu. Validační dataset většinou vzniká vymezením určité části trénovacích dat.
- Testovací data – Tento dataset je použit pro testování natrénovaného modelu. Pro vyhodnocení kvality natrénovaní modelu se používá celá řada metrik. Mezi hlavní patří například senzitivita, přesnost, specificita a mnoho dalších.



Obr. 1.7: Rozdělení dat [8].

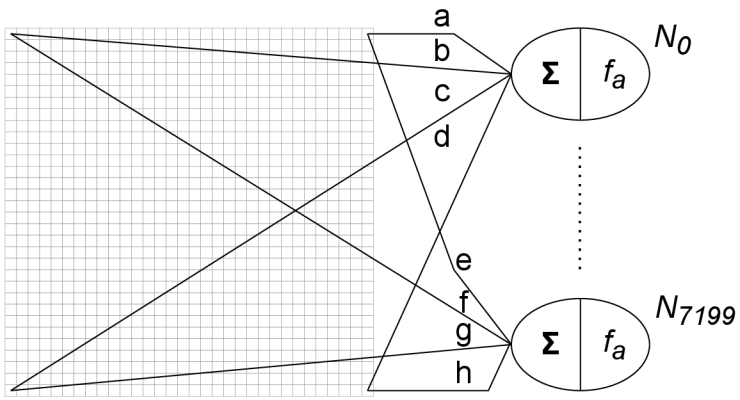
Pravidla, která se týkají rozdělení dat pro různé úkony v procesu trénování, validace a testování modelu, jsou poměrně striktní. Pokud by totiž nebyla dodržována, nefungovalo by učení neuronové sítě správně. Doporučené rozdělení datasetu na trénovací, validační a testovací data je v poměru 2:1:1 [8].

1.5 Konvoluční neuronové sítě

V kapitole 1.4 byla představena základní architektura hluboké neuronové sítě s plně propojenými vrstvami. Takováto architektura ovšem není pro zpracování obrazů úplně ideální. Neuronová síť s plně propojenými vrstvami, která by měla schopnost se natrénovat na velký počet tříd obrazů, by musela obsahovat velké množství neuronů a její běh by byl výpočetně velmi náročný. Z tohoto důvodu se u konvolučních neuronových sítí (anglicky Convolutional neural network) přistupuje k redukci množství vstupních dat pomocí matematické operace konvoluce. U obrazových dat lze předpokládat značnou míru podobnosti u sousedících pixelů, je proto možné aproximovat hodnotu z určité oblasti pixelů a tím zmenšit množství dat, která do neuronové sítě vstupují, přičemž ztráty informací jsou minimální. [9, 10]

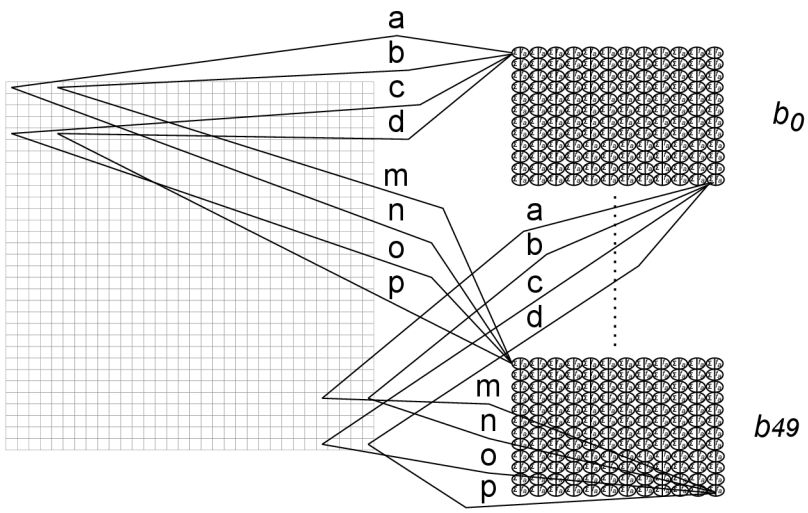
1.5.1 Odvození konvoluční vrstvy z plně propojené vrstvy

Odvození vzniku konvoluční neuronové sítě z plně propojené neuronové sítě je vhodné vysvětlit na příkladu. Mějme obraz reprezentovaný pouze v odstínech šedi o rozměrech 32 × 32 pixelů a plně propojenou skrytou vrstvu neuronové sítě, která obsahuje 7200 neuronů. Obraz je převeden na vektor o délce 1024 a každý neuron je propojen s každým pixelem. Tímto způsobem je vytvořeno 1024 × 7200 = 7 372 800 spojení. Tato situace je schématicky naznačena na obrázku 1.8. Z praktických důvodů zde nejsou naznačena všechna spojení a neurony. Mřížka o rozměrech 32 × 32 symbolizuje zpracovávaný obraz, N_0 a N_{7199} symbolizují první a poslední neuron ve skryté plně propojené vrstvě, písmena u jednotlivých spojení symbolizují váhy, kterými jsou neurony spojeny s příslušnými pixely.



Obr. 1.8: Zpracování obrazu plně propojenou vrstvou [10].

Pro vytvoření konvoluční vrstvy z plně propojené vrstvy je nejprve potřeba neurony rozdělit do bloků. Tímto způsobem je vytvořeno 50 bloků, které obsahují 12 × 12 neuronů. Počet spojení prozatím zůstává nezměněn. Jak již bylo zmíněno v úvodu této kapitoly, v obrazových datech spolu hodnoty sousedních pixelů korelují. Z tohoto důvodu je možné pixely sdružit do skupin a extrahovat příznaky z těchto skupin a ne z každého pixelu zvlášť. V tomto případě byly pixely sdruženy do skupin o rozměrech 5 × 5 pixelů. Tímto se výrazně redukuje počet spojení na: (5 × 5) × 50 × 12 × 12 = 180 000. Pro další redukci spojení můžeme předpokládat, že všechny neurony v jednom bloku sdílí váhy. Touto redukcí je dosaženo 5 × 5 × 50 = 1250 spojení mezi obrazem a skrytou vrstvou. Oproti plně propojené vrstvě je tedy možné zredukovat počet spojení o 99,98 %. Tato technika redukující počet spojení se nazývá sdílení vah (anglicky Weight sharing). Graficky je tento přístup naznačen na obrázku 1.9, kde mřížka 32 × 32 symbolizuje zpracovávaný obraz, b_0 a b_{49} symbolizují první a poslední blok neuronů o rozměru 12 × 12 a jednotlivá písmena symbolizují váhy, kterými jsou spojeny bloky neuronů s bloky pixelů.



Obr. 1.9: Zpracování obrazu konvoluční vrstvou [10].

Výstup jednotlivých neuronů je poté shodný jako výstup konvoluce s filtrem o velikosti 5 × 5 a impulzní charakteristikou odpovídající vahám neuronu. Výstupem konvoluční vrstvy je množina obrazů, zvaných příznaková mapa, jejichž počet je shodný s počtem filtrů. Na tento výstup jsou poté aplikovány aktivační funkce zvoleného průběhu, typicky se jedná o aktivační funkci ReLu. V případě, že je zpracováván barevný obraz reprezentovaný RGB modelem a ne pouze obraz černobílý, jsou konvoluční filtry stále dvourozměrné, ale jsou aplikovány na každý barevný kanál zvlášť. [10]

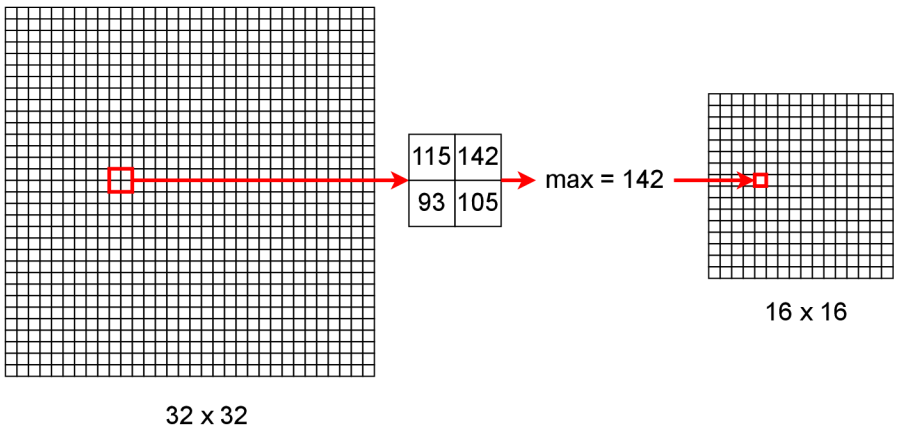
1.5.2 Pooling vrstva

Další vrstvou, která se typicky nachází v konvolučních neuronových sítích je pooling vrstva, česky také nazývaná sdružovací vrstva. Jejím úkolem je podvzorkovat příznakové mapy, což v tomto případě znamená zmenšit jejich rozměr. Pokud by k podvzorkování nedošlo, další práce s takovýmito příznakovými mapami by byla výpočetně velmi náročná, proto je na každou z příznakových map aplikována pooling vrstva s určitými rozměry filtru a krokem (anglicky Stride), které definují, kolikrát bude zmenšena příznaková mapa. Rozlišujeme 3 základní pooling vrstvy:

- Max pooling – Z vybrané oblasti bude použita maximální hodnota.
- Min pooling – Z vybrané oblasti bude použita minimální hodnota.
- Average pooling – Z vybrané oblasti bude použita průměrná hodnota.

Fungování Max pooling vrstvy je naznačeno na obrázku 1.10. Filtr zde má rozměry 2 × 2 a krok má hodnotu 2. Z původního obrazu o rozměru 32 × 32 je vždy vybrána oblast o rozměrech 2 × 2 pixely a z tohoto vzorku je vybrána maximální hodnota

pixelu, která je následně přiřazena pixelu v nově vznikajícím obrazu o rozměrech 16×16. [10]

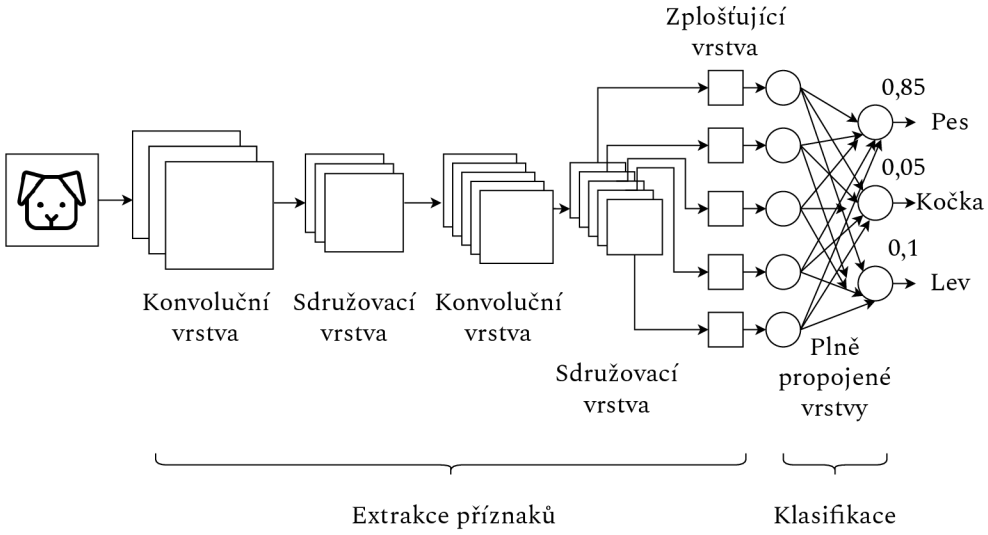


Obr. 1.10: Max pooling vrstva [10].

1.5.3 Typická architektura konvoluční neuronové sítě

Návrh konvoluční neuronové sítě, která bude vykazovat dobré výsledky a bude zároveň co nejméně výpočetně náročná, může být opravdu výzvou. V praxi neexistuje nějaké univerzální pravidlo, které by definovalo návrh takovéto neuronové sítě [10]. Je proto nutné experimentovat a podle výsledků, které síť vykazuje při trénování a případné validaci, upravovat architekturu a parametry jednotlivých vrstev. Jsou zde však základní pravidla, která budou popsána na obrázku 1.11.

Do konvoluční neuronové sítě vstupuje obraz, který prochází konvoluční vrstvou, přičemž po konvoluční vrstvě typicky ihned následuje pooling vrstva, ve které je obraz podvzorkován. Takovýchto iterací se typicky provádí několik. Výstup pooling vrstvy je však multidimenzionální. Aby tento výstup byly schopny zpracovat plně propojené vrstvy, je nutné tento výstup převést na jednodimenzionální, což provádí vrstva flatten, česky nazývaná také zplošťující vrstva. Tímto končí proces extrakce příznaků z obrazu a výstup flatten vrstvy je předán typicky několika plně propojeným vrstvám. Tyto vrstvy provádí klasifikaci a výpočet pravděpodobnosti, s jakou je na zpracovaném obraze daná třída. Výstupem neuronové sítě je množina čísel, které reprezentují pravděpodobnosti s jakými síť daný obrázek zařadila do některé z možných tříd. V modelovém případě na obrázku 1.11 tedy konvoluční neuronová síť predikovala, že na obrazu, který do sítě vstupuje, je s pravděpodobností 85% pes, s pravděpodobností 5% kočka, nebo s pravděpodobností 10% lev. [10]



Obr. 1.11: Architektura konvoluční neuronové sítě [11].

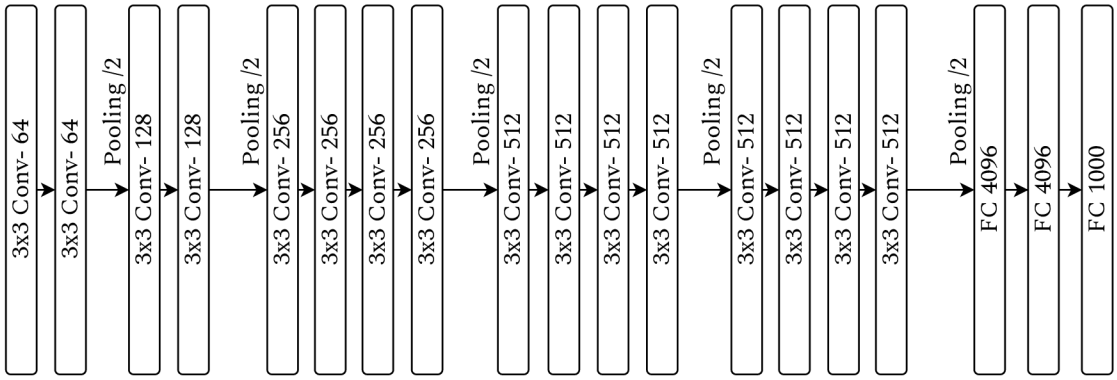
1.5.4 Předtrénované modely

V praxi se poměrně často využívají již vytvořené a předtrénované modely. Toto výrazně ulehčuje práci, kterou by bylo potřeba vynaložit na vytvoření neuronové sítě a její počáteční natrénování. Předtrénované modely lze totiž upravit a také znovu natrénovat na jiném datasetu pro konkrétní řešený problém. Tato technika se nazývá Transfer learning. Modely, jenž budou popsány v této kapitole, byly natrénovány na datasetu s názvem ImageNet. ImageNet obsahuje více než 21 tisíc tříd, do kterých lze daný obrázek zařadit, a celkově více než 14 milionů obrázků [12]. Na tomto datasetu se také hodnotí kvalita natrénovaného modelu. K tomu jsou použity metriky přesnosti Top-1 a Top-5. Top-1 přesnost určuje v kolika procentech případů model správně klasifikoval objekt na obrázku, přičemž správná třída byla na prvním místě. Top-5 přesnost určuje v kolika procentech případů model správně klasifikoval objekt na obrázku, přičemž správná třída byla mezi pěti prvními třídami s nejvyšší pravděpodobností [13].

VGG

Konvoluční neuronová síť VGG dostala název podle výzkumné skupiny Visual Geometry Group z Oxfordské univerzity. Vznikla v roce 2014 a i přes její poměrně jednoduchou architekturu dosahuje velmi dobrých výsledků. Existuje ve více verzích, přičemž mezi hlavní patří VGG16 a VGG19. Vstupem sítě VGG je obraz o velikosti 224×224. Ten následně prochází konvolučními vrstvami, které mají fixní velikost filtru 3×3 a velikost kroku 1. Konvoluční vrstvy jsou u obou modelů VGG roz-

děleny do 5 bloků, přičemž výstup každého z bloků projde operací Max pooling. Výstup z posledního bloku konvolučních vrstev následně prochází třemi plně propojenými vrstvami, z nichž dvě obsahují 4096 neuronů a poslední vrstva obsahuje 1000 neuronů. Architektura neuronové sítě VGG19 je na obrázku 1.12. [14]

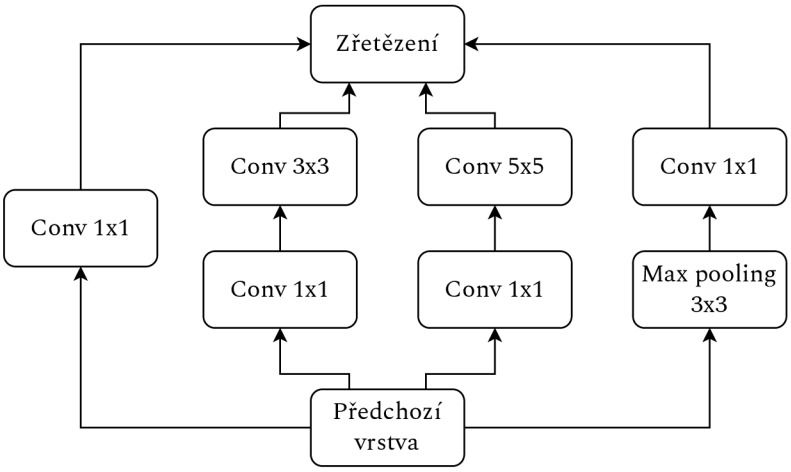


Obr. 1.12: Architektura neuronové sítě VGG19 [14].

Inception

Jedním z problémů, který bylo nutné vyřešit při klasifikaci obrazů, je rozdílná velikost objektů na obrazu. Na některých snímcích může klasifikovaný objekt zabírat celou plochu, na jiných může být jen v malé oblasti. Tento problém řeší architektura Inception V1 tím, že neprovádí pouze jednu konvoluční operaci, ale rovnou tři s různými rozměry filtru. Zároveň je prováděna také operace Max pooling a výstupy těchto bloků jsou následně zřetězeny. Toto řešení, které se anglicky označuje jako „Naive version“, ale poměrně zvyšuje výpočetní náročnost, protože několik konvolučních filtrů zvyšuje dimenzionalitu výstupu. Z tohoto důvodu vznikla upravená verze tohoto přístupu, kde je před každý z konvolučních filtrů přidán ještě jeden konvoluční filtr o rozměrech 1×1, který redukuje dimenzionalitu výstupu a tím snižuje výpočetní náročnost. Tento přístup je patrný na obrázku 1.13. [15]

Inception V2 přinesla další zlepšení přesnosti a redukci výpočetní náročnosti. Experimenty bylo zjištěno, že neuronová síť funguje lépe, pokud nejsou původní rozměry obrazu měněny příliš drasticky. Přílišné změny totiž mohou způsobovat ztrátu informace. Tvůrci sítě Inception V2 tento problém vyřešili pomocí faktorizace jednotlivých konvolučních bloků. Model tímto způsobem lépe udržuje rovnováhu mezi šířkou a hloubkou. Konvoluční blok s filtrem o rozměrech 5×5 lze například rozložit na dva bloky s menšími filtry o rozměrech 3×3, přičemž výsledek filtrace zůstane stejný a výpočet je 2,78× rychlejší.



Obr. 1.13: Modul sítě Inception s redukcí dimenzionality [15].

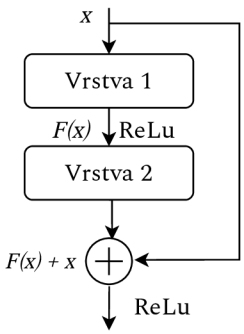
Dále lze také faktorizovat konvoluční bloky o rozměrech n × n na kombinace asymetrických konvolučních bloků o rozměrech 1 × n a n × 1. Tímto přístupem je ušetřeno až 33% prostředků. [15]

Inception V3 je dalším vylepšením Inception V2. Mezi klíčové změny oproti předchozí verzi patří [16]:

- Vylepšení loss funkce.
- Faktorizace 7 × 7 konvolučních bloků.
- Implementace batch normalization v auxiliárních klasifikátorech.

ResNet

Při vývoji hlubokých konvolučních sítí bylo zjištěno, že dále zvětšovat hloubku, tzn. přidávat vrstvy, těchto sítí nestačí a při určitém množství začne chybovost při trénování opět růst. Tomuto jevu se říká „Degradation problem“. Řešení tohoto problému přináší sítě typu ResNet, které používají tzv. hluboké residuální učení. Toto učení na rozdíl od klasického učení používá při průchodu jednotlivými vrstvami tzv. zkratky, kdy mohou být přeskočeny jednotlivé vrstvy nebo celé bloky vrstev. Toto je patrné na obrázku 1.14a. Experimenty bylo zjištěno, že hluboké residulální sítě jsou odolné vůči degradaci a zároveň nepřináší zvýšení výpočetní náročnosti při narůstajícím počtu vrstev v neuronové síti. To je patrné na obrázku 1.14b, který obsahuje hodnoty Top-1 chybovosti. Pro dosažení ještě lepších výsledků se v praxi používají i modely, které kombinují ResNet a Inception architekturu, například InceptionResNet V2. [17]



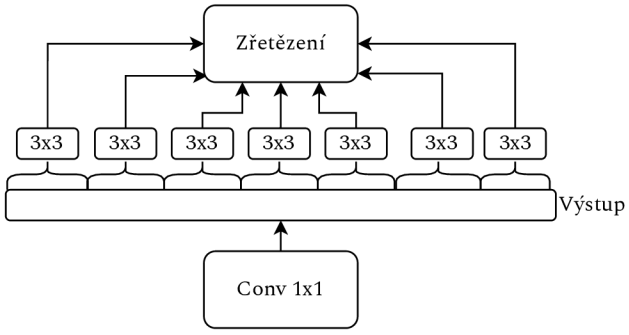
(a) Princip fungování sítě ResNet [17].

	běžná CNN	ResNet
18 vrstev	27,94	27,88
34 vrstev	28,54	25,03

(b) Srovnání Top-1 chybovosti běžné konvoluční sítě s ResNet sítí [17].

Xception

Název Xception vznikl spojením slov „Extreme Inception“. Z tohoto je patrné, že model Xception vychází z modelu Inception V3, obsahuje ale jisté modifikace. Vstup je přiveden do konvolučního bloku s filtrem o rozměrech 1×1 a výstup tohoto bloku je poté rozdělen a jsou s ním provedeny konvoluce 3×3, podle obrázku 1.15. Dalším rozdílem je, že na rozdíl od Inception V3 nenásleduje po první konvoluci nelinearita v podobě aktivační funkce ReLu. Do modelu Xception bylo také implementováno residuální učení, protože experimenty bylo zjištěno, že model vykazuje lepší výsledky, pokud jsou do něj přidány zkratky. Spojením vlastností svých předchůdců a jejich lehkými úpravami byl model Xception schopen na datasetu ImageNet překonat jak VGG, tak Inception V3 i ResNet modely. [18]



Obr. 1.15: Modul sítě Xception [18].

Srovnání předtrénovaných modelů

Jak bylo popsáno v úvodu této kapitoly, modely jsou mezi sebou porovnávány na základě Top-1 a Top-5 přesnosti. V tabulce 1.1 jsou pro představu uvedeny hodnoty

těchto přesností pro některé vybrané předtrénované modely dostupné v nástroji Keras.NET.

Model	**Top-1 přesnost**	**Top-5 přesnost**
InceptionResNetV2	80,3%	95,3%
Xception	79%	94,5%
ResNet152V2	78%	94,2%
InceptionV3	77,9%	93,7%
ResNet50	76%	92,1%
VGG16	71,3%	90,1%
VGG19	71,3%	90%

Tab. 1.1: Srovnání vybraných předtrénovaných modelů [19].

2 Použité technologie

Základem pro vytvoření kvalitní aplikace je volba správných nástrojů pro její vytvoření. Vybrané stěžejní technologie použité v této aplikaci budou v krátkosti uvedeny v následujících podkapitolách.

2.1 C#

C# je moderní multiplatformní programovací jazyk s otevřeným kódem patřící do platformy .NET. Jedná se o objektově orientovaný programovací jazyk, který je mezi pěti nejpoužívanějšími programovacími jazyky na platformě GitHub. Základ své syntaxe zdědil ze svých předchůdců, jazyků C a C++, takže se znalostí těchto jazyků není příliš náročné přejít na novější C#. Zároveň je C# použit v opravdu širokém spektru nasazení, které zahrnuje desktop aplikace, webové aplikace, cloudová řešení, IoT aplikace či umělou inteligenci. [20]

2.2 ASP.NET

ASP.NET je další součástí platformy .NET. Jedná se o framework, který slouží k vývoji webových aplikací. Obsahuje nástroje pro zpracování webových dotazů, tvorbu statických či dynamických webových stránek, knihovny pro obvyklé architektury webových aplikací, jako je například MVC (Model View Controller), či autentizační mechanismy pro přihlašování, databáze a další. [21]

Jednou z klíčových technik, kterou platforma .NET, potažmo ASP.NET nabízí, je asynchronní programování. Pomocí této techniky je možné provádět simultánní operace, aniž by se vzájemně blokovaly. Při porovnání synchronního a asynchronního kódu nelze konstatovat, že by asynchronní kód vykazoval větší rychlost než synchronní kód při konkrétních jednoduchých operacích, jako je například získání dat z databáze. Asynchronní kód ale přináší zlepšení tzv. nepřímé výkonnosti, protože server je schopen zpracovat více požadavků zároveň.

Při použití synchronního přístupu je maximální počet požadavků, které je schopen server zpracovat v jeden okamžik, omezen na počet dostupných vláken. Počet dostupných vláken se odvíjí od použitého procesoru serveru. V případě, že jsou všechna vlákna vyčerpána, protože každé vlákno momentálně obsluhuje jiný požadavek, nastává stav, kterému se anglicky říká „Thread starvation“. Při tomto stavu musí být nejprve dokončen některý z požadavků, následně je uvolněno vlákno, které tento požadavek zpracovávalo, a až poté je možné obsloužit další požadavek. Důsledkem tohoto je zhoršení doby odezvy aplikace, případně chybové odpovědi serveru informující o nedostupnosti služby.

Asynchronní programování představuje řešení těchto problémů. Stejně jako u synchronního přístupu je potřeba pro zpracování požadavku vyčlenit vlákno z dostupných vláken. Na rozdíl od předchozího přístupu je zde ale vlákno uvolněno i během zpracování požadavku, pokud není zrovna aktivně využíváno. Příkladem může být situace, kdy server přijímá požadavky a vrací data z databáze. Databázi trvá určitou dobu, než vrátí požadovaná data a během této doby je vlákno uvolněno a přidáno mezi dostupná vlákna, než databáze dokončí potřebné operace. Během této doby, kdy aplikace čeká na data z databáze, je tak možné obsloužit další požadavky, které na server přišly.

Pro implementaci asynchronního přístupu pomocí platformy .NET jsou použity 3 základní klíčová slova:

- `async`,
- `await`,
- `Task`.

Klíčové slovo `async` je použito při deklaraci metod a označuje asynchronní metodu. Samotné použití tohoto klíčového slova ale nezajišťuje asynchronní průběh. V těle metody je potřeba použít také klíčové slovo `await`. Toto klíčové slovo umístěné před volání metody zajišťuje, že v případě, že bude potřeba čekat na návratovou hodnotu této metody, bude vlákno obsluhující tuto metodu uvolněno a bude moci obsloužit další požadavek. Asynchronní metoda může mít pouze 3 návratové typy: `Task`, `Task<T>` nebo `void`. Návratový typ `Task` je použit, pokud není potřeba vracet žádná data, jedná se tedy prakticky o ekvivalent návratového typu `void` u asynchronních metod. Návratový typ `Task<T>` vrací libovolný datový typ, který je dosazen za generický parametr `T`. Návratový typ `void` je použit výjimečně, například při zpracování uživatelských interakcí s grafickým rozhraním, kde je tento návratový typ vyžadován. [22]

Typické použití asynchronního kódu je patrné na výpisech 4.4 a 4.5 v implementační části této práce.

2.3 Blazor

Blazor je framework určený pro vytváření dynamického uživatelského rozhraní v jazyce C#. Eliminuje nutnost použití programovacího jazyka JavaScript a umožňuje implementovat jak serverovou část aplikace (Back end), tak uživatelské rozhraní (Front end) v jazyce C#. Uživatelské rozhraní je vygenerováno jako HTML a CSS soubory a díky tomu má podporu v široké škále internetových prohlížečů na různých platformách. Blazor aplikaci lze vyvíjet podle dvou základních architektur: Blazor Web Assembly a Blazor Server.

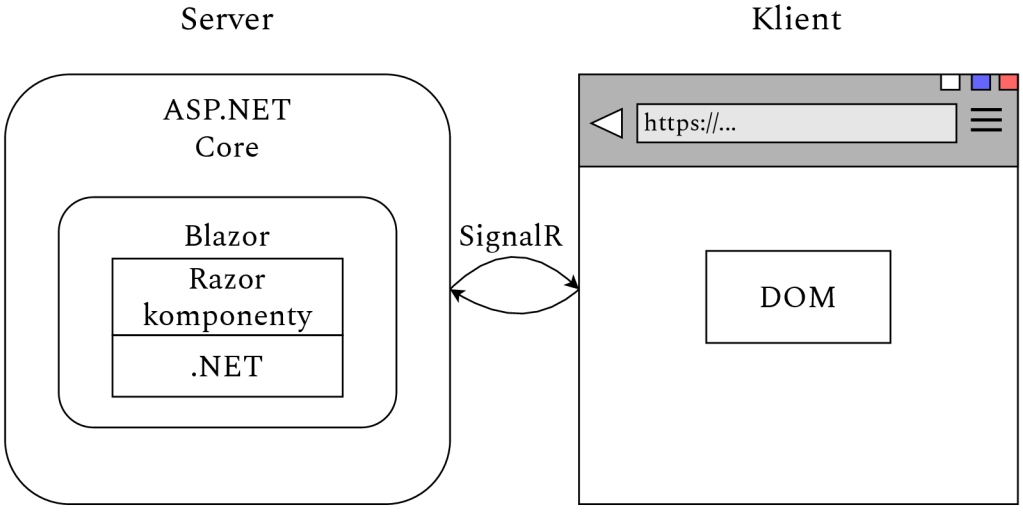
Blazor Web Assembly aplikace běží na straně klienta přímo v prohlížeči na runtime platformě .NET. Takováto Blazor aplikace je celá stažena do prohlížeče a až poté spuštěna. Výhody tohoto přístupu jsou rychlost, fungování aplikace i v offline módu, eliminace potřeby serveru. Nevýhodou tohoto přístupu je pomalejší načítání aplikace kvůli nutnosti stažení všech souborů a také chybějící podpora této technologie u starších prohlížečů. [23]

Blazor Server aplikace běží na serveru v rámci ASP.NET Core aplikace. Aktualizace uživatelského rozhraní, uživatelské interakce a volání JavaScriptu jsou zpracovávána pomocí SignalR spojení. Díky technologii SignalR je možná komunikace mezi klientem a serverem v reálném čase. S každým klientem připojeným k serveru je vytvořeno spojení, které se anglicky nazývá „Circuit“. Tato spojení jsou schopná zvládat dočasné výpadky spojení a v případě potřeby jsou znovu navázána. Spojení je automaticky ukončeno, pokud uživatel zavře okno prohlížeče s aplikací. Výhody tohoto přístupu jsou:

- Rychlejší prvotní načtení aplikace.
- Možnosti aplikace nejsou omezeny na možnosti prohlížeče.
- Podpora starších verzí prohlížečů.
- Možnost debuggování.

Nevýhodou naopak může být vyšší latence nebo nefunkčnost aplikace v režimu offline [23].

Pro tuto aplikaci byla zvolena architektura Blazor Server především z důvodu širší podpory v internetových prohlížečích. Tato architektura je znázorněna na obrázku 2.1.



Obr. 2.1: Blazor Server architektura [23].

Blazor aplikace jsou založeny na použití tzv. komponent. Jako komponenta se označují jednotlivé prvky uživatelského rozhraní jako jsou například stránky, formuláře nebo dialogy. Tyto komponenty jsou definovány jako C# třídy, které mohou flexibilně generovat uživatelské rozhraní a obsluhovat uživatelská volání. Jejich velkou výhodou je možnost mnohonásobného využití na několika místech v aplikaci. Komponenty používají syntaxi Razor, která kombinuje značkovací jazyk HTML a jazyk C#. [23]

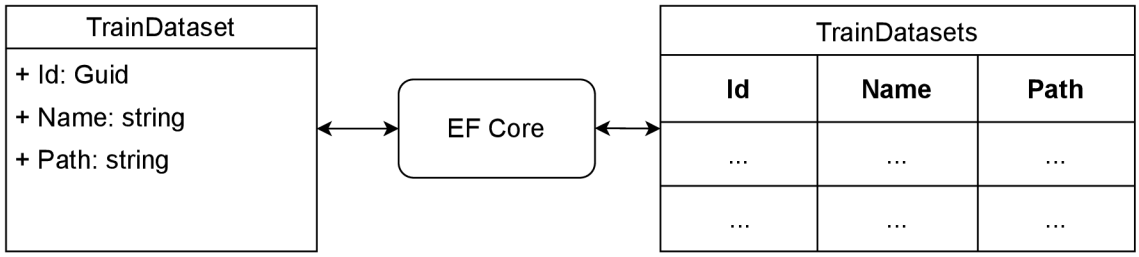
2.4 Radzen

Radzen je projekt s otevřeným zdrojovým kódem, který vytváří nadstavbu nad Blazor komponentami. V současné době Radzen nabízí více než 70 různých komponent. Tyto komponenty jsou implementovány v jazyce C# plně v souladu s Blazor technologií. Nevyužívají tak existující JavaScript knihovny nebo frameworky. Radzen podporuje obě architektury Blazor aplikací - jak Blazor Web Assembly, tak Blazor Server. Radzen samozřejmě není jediným projektem, který vytváří nadstavbu nad Blazor komponenty, v tomto projektu byl však zvolen pro jeho dobrou komunitní podporu, jednoduché začlenění do projektu a širokou škálu komponent, které nabízí. [24]

2.5 EF Core

Entity Framework Core je multiplatformní verze populárního nástroje EF (Entity Framework). Jedná se o nástroj, který slouží k ORM (Objektově Relační Mapování). Objektově relační mapování je technika, která umožňuje pracovat v aplikaci s jednotlivými záznamy v databázových tabulkách jako s objekty. Není tedy nutné ručně psát SQL dotazy a je možné pouze provádět operace nad objekty. EF Core podporuje řadu poskytovatelů databází. Mezi hlavní patří například SQLite, MariaDB a PostgreSQL [25]. Pro toto konkrétní řešení byl zvolen databázový engine SQLite z důvodu jeho vhodnosti pro prototypový vývoj aplikací. Na obrázku 2.2 je znázorněno fungování nástroje EF Core.

Na levé straně je znázorněna třída `TrainDataset`, která má atributy `Id` (datový typ Guid), `Name` a `Path` (datový typ string). EF Core provádí mapování mezi databázovou tabulkou `TrainDatasets`, jejíž sloupce odpovídají jednotlivým atributům třídy `TrainDataset` a objekty, které jsou vytvořeny jako nové instance třídy `TrainDataset`.



Obr. 2.2: Fungování EF Core.

2.6 Keras - Keras.NET

Keras je vysokoúrovňové API pro hluboké učení implementované v programovacím jazyce Python. Jeho základem je platforma TensorFlow. Keras byl vyvinut s důrazem na jednoduchost a z ní pramenící rychlost adaptace uživatele na tento nástroj. Nástroj Keras je [26]:

- Jednoduchý – Keras se snaží omezit kognitivní zátěž na uživatele, aby se mohl soustředit na jádro problému.
- Flexibilní – Lze jej používat jak na řešení jednoduchých problémů a počáteční experimentování, tak na pokročilé a náročnější problémy.
- Výkonný – Keras nalézá své upotřebení ve společnostech jako NASA či YouTube, je tedy schopen zvládat průmyslová nasazení ve velkých firmách.

Keras.NET je knihovnou, která umožňuje využití funkcionalit nástroje Keras pomocí jazyka C#. Umožňuje tedy jednoduše experimentovat se strojovým učením bez nutnosti použití programovacího jazyka Python. Stejně jako Keras umožňuje provádět výpočty jak na CPU, tak na GPU. [27]

3 Návrh implementace

Nezbytnými součástmi vývoje aplikace jsou rozbor problematiky a návrh řešení, které přichází na řadu jako první. Je potřeba navrhnout architekturu aplikace a promyslet logickou návaznost jednotlivých částí, aby bylo dosaženo co nejlepšího výsledku při implementaci.

3.1 Nahrání datasetu

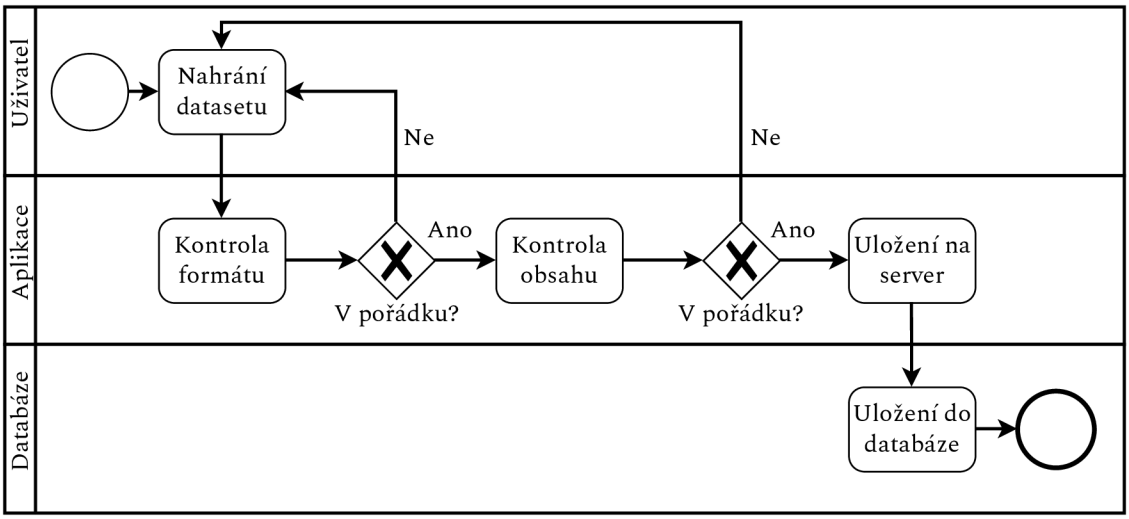
Diagram, který ilustruje jednotlivé kroky při nahrání datasetu na server, je na obrázku 3.1. Tento typ diagramu se nazývá BPMN (Bussiness Process Model and Notation) diagram. Používá se především pro modelování podnikových procesů, je ale vhodný i pro ilustraci fungování softwarových řešení [28].

V diagramu vystupují 3 entity: Uživatel, Aplikace a Databáze. Aby uživatel mohl natrénovat model, jsou potřeba trénovací data. Trénovací data budou nahrána jako archiv formátu ZIP, který bude mít následující vnitřní strukturu:

```
Dataset.zip
├── Třída1
│   ├── Obrázek1-1.jpg
│   └── Obrázek1-2.jpg
├── Třída2
│   ├── Obrázek2-1.jpg
│   └── Obrázek2-2.jpg
├── ...
├── ...
└── TřídaN
    ├── ObrázekN-1.jpg
    └── ObrázekN-2.jpg
```

V souboru `Dataset.zip` tedy budou složky, jejichž název bude reprezentovat třídu, do které obrázky v této složce náleží. Pokud je v aplikaci potřeba nechat uživatele nahrát na server soubory, jedná se o potenciální bezpečnostní riziko a je potřeba kontrolovat, co se uživatel snaží nahrát na server. Jak z hlediska funkcionality, tak z hlediska bezpečnosti je tedy potřeba kontrola nahrávaných souborů.

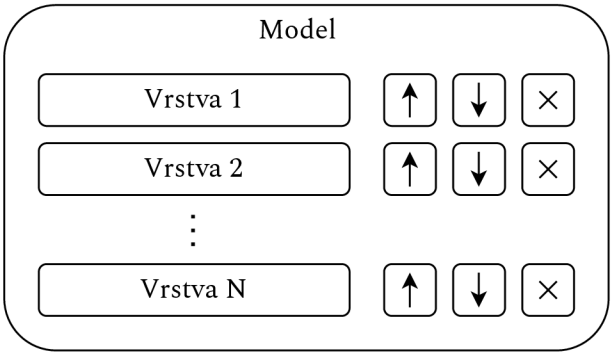
Aplikace bude pro nahrání přijímat pouze 1 soubor, u kterého bude kontrolováno, zda se jedná o archiv formátu ZIP. Soubory jiného formátu nebudou nahrány a uživatel bude upozorněn, že tento soubor není podporován. Pokud proběhne první kontrola v pořádku, bude kontrolován obsah archivu ZIP. Aplikace bude kontrolovat, zda archiv obsahuje pouze obrázky ve formátu JPEG nebo ve formátu PNG. Ostatní formáty nebudou podporovány. V případě splnění těchto podmínek bude dataset uložen na uložiště serveru a do databáze bude uložen záznam o tomto datasetu.



Obr. 3.1: Diagram nahrání datasetu.

3.2 Tvorba modelu

Model neuronové sítě se skládá z jednotlivých vrstev. Z uživatelského hlediska tak bude přívětivé, pokud uživatel bude moci skládat jednotlivé vrstvy na sebe jako na obrázku 3.2. Uživateli tak bude nabídnuto vybrat kýženou vrstvu k přidání, specifikování jejích parametrů a samotné přidání. Po přidání bude uživateli umožněno měnit pořadí vrstev, upravovat parametry vrstev či odebrat vrstvy z modelu. Poté, co bude uživatel spokojen se svým modelem, bude moci vytvořený model uložit. I poté se ale může stát, že v budoucnu bude uživatel svůj model chtít upravit, z tohoto důvodu je nutné implementovat i možnost modifikace uloženého modelu.

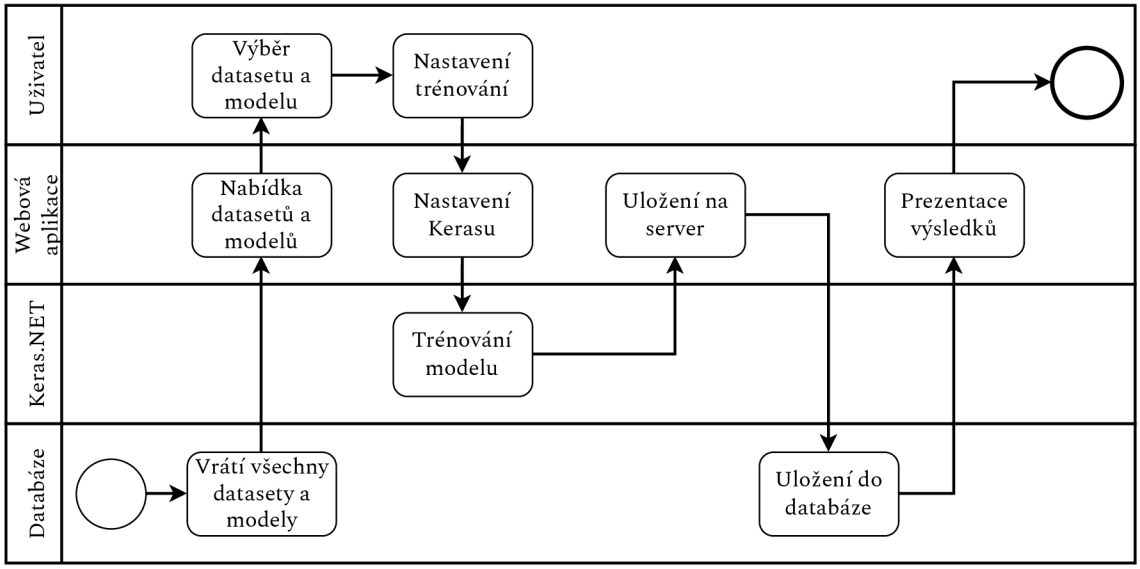


Obr. 3.2: Návrh tvorby modelu.

3.3 Trénování modelu

Diagram reprezentující trénování modelu je na obrázku 3.3. Oproti předchozímu diagramu jsou zde dvě nové entity: Webová aplikace a Keras.NET. Tyto dvě entity byly v předchozím diagramu sloučeny do entity Aplikace, protože nebylo nutné sledovat zvlášť jejich akce. Zde je však vhodné sledovat akce obou těchto entit, a proto byly rozděleny.

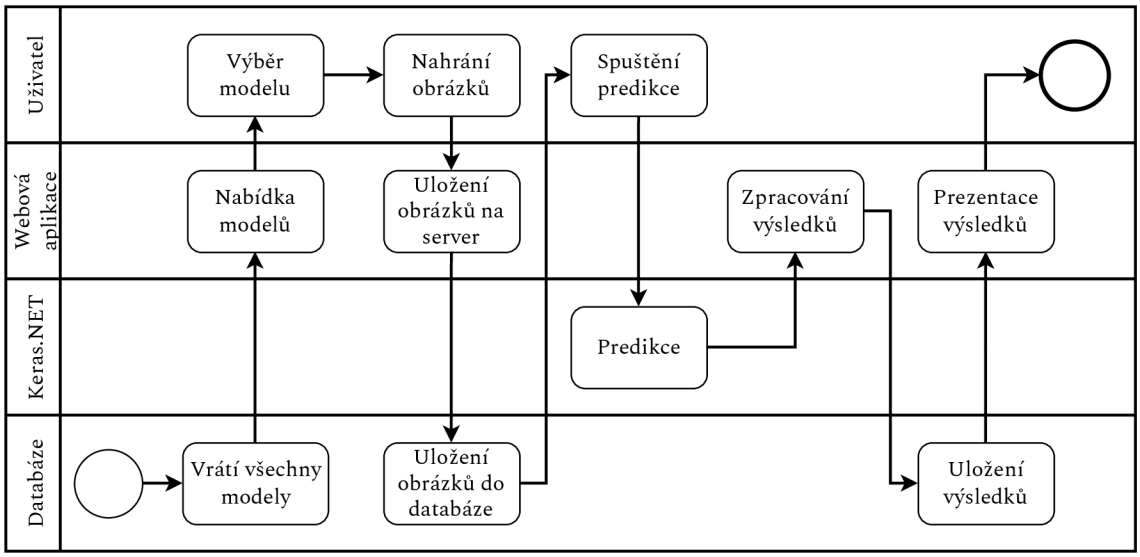
Nejprve musí uživatel vybrat dataset, který bude sloužit jako trénovací, a model, na kterém bude provedeno trénování. Z databáze bude zjištěno, jaké datasety jsou na serveru uloženy a uživateli budou nabídnuty k výběru. Totéž proběhne pro modely uložené v databázi. Uživatel následně provede nastavení trénování. Nastavit bude moci celou řadu parametrů, od nastavení předzpracování dat, přes parametry trénování modelu, až po nastavení evaluace dat. Tato nastavení budou zpracována webovou aplikací a přemapována na potřebný formát, který vyžadují vstupy nástroje Keras.NET. Poté, co bude dokončeno trénování modelu, bude model uložen jak na server, tak do databáze, spolu s některými metrikami daného modelu, jako je například přesnost (anglicky accuracy).



Obr. 3.3: Diagram trénování modelu.

3.4 Predikce modelu

Aby uživatel mohl prakticky otestovat správné natrénování modelu, bude moci vyzkoušet klasifikaci libovolného obrázku zvoleným modelem. Navržený průběh predikce je na diagramu 3.4.



Obr. 3.4: Klasifikace zvoleného obrázku.

Uživatel nejprve vybere model pro klasifikaci. Poté nahraje jeden nebo více obrázků, na kterých bude model testovat. Tyto obrázky budou uloženy na server a záznam o nich do databáze, v níž budou přiřazeny ke zvolenému modelu. Po úspěšném nahrání uživatel spustí predikci. Nástroj Keras.NET nejprve snímky transformuje do potřebné podoby a následně provede samotnou predikci. Výstupem této predikce je soubor tříd, ke kterým může daný obrázek náležet, s příslušnou pravděpodobností. Tyto výsledky aplikace zpracuje, uloží je k příslušným obrázkům do databáze a uživateli zobrazí výsledky predikce.

4 Implementace

V předchozí kapitole byl představen základní návrh jednotlivých funkcionalit vyvíjené aplikace. V této kapitole budou přiblíženy praktické postupy při samotné implementaci těchto funkcionalit.

4.1 Perzistence dat

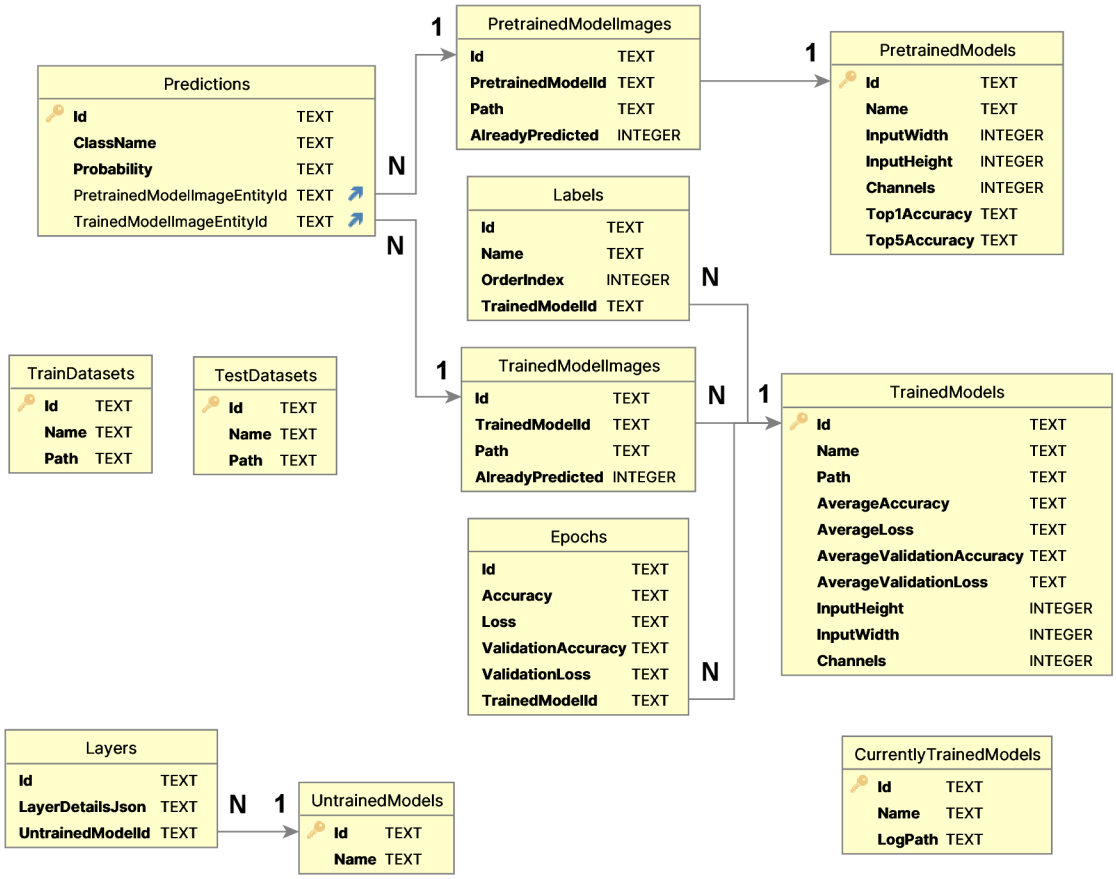
Jednou z klíčových funkcionalit této aplikace je ukládání natrénovaných modelů a příslušných průběhů trénování. Ke správné funkčnosti aplikace je ale potřeba ukládat dat mnohem více. K tomuto účelu slouží databáze. Díky použití nástroje EF Core, jenž byl zmíněn v kapitole 2.5, je možné pružně měnit strukturu databáze podle potřeb, které postupně vznikají během vývoje aplikace. Ještě před samotnou implementací kódu bylo potřeba nainstalovat potřebné balíčky pro SQLite a EF Core ze správce balíčku NuGet, který je určen pro platformu .NET. Struktura databáze pro tuto aplikaci je na obrázku 4.1. Všechny tabulky obsahují primární klíč, unikátní identifikátor `Id`, který je použit při referencích mezi tabulkami.

Tabulka `TrainDatasets` obsahuje informace o trénovacích datasetech, které jsou uložené na serveru. Konkrétně obsahuje název datasetu, který zvolil uživatel, a umístění na serveru. V tabulce `TestDatasets` se nachází obdobné údaje, ale pro testovací datasety.

V tabulce `TrainedModels` se nachází informace o natrénovaných modelech, které byly uloženy. Obsahuje informace o názvu modelu, který definuje uživatel, umístění na uložišti serveru, průměrné přesnosti a průměrnou hodnotu chybové funkce jak při trénování, tak při validaci a informace o rozměrech vstupních dat modelu. Pomocí referencí mají jednotlivé záznamy v tabulce `TrainedModels` přiřazeny záznamy z tabulek `Epochs`, `TrainedModelImages` a `Labels`.

Tabulka `Epochs` obsahuje informace o výsledcích jednotlivých trénovacích epoch. Ukládá hodnoty přesnosti a hodnoty chybové funkce na trénovacích a validačních datech.

V tabulce `TrainedModelImages` jsou uloženy informace o obrázcích, které uživatel nahrál pro predikci daného modelu. Obsahuje cestu k danému obrázku (`Path`), proměnnou typu boolean, která definuje, zda již byla pro daný obrázek provedena predikce (`AlreadyPredicted`) a navigační proměnnou `TrainedModelId`, pomocí které je obrázek přiřazen k modelu.



Obr. 4.1: Entitně relační diagram databáze.

Pro uložení informací o třídách, do kterých byly zařazeny obrázky z datasetu, na kterém probíhalo učení daného modelu slouží tabulka `Labels`. Ta obsahuje názvy tříd a jejich pořadí.

Informace o pořadí je nutné udržovat, protože při ukládání do databáze nemusí být pořadí dodrženo a výstupem predikce modelu je pouze pole, ve kterém jsou pravděpodobnosti, které indikují příslušnost obrázku k jednotlivým třídám. Aby tedy bylo možné správně interpretovat predikci modelu, je nutné mít přehled o pořadí jednotlivých tříd.

Samotné predikce pro jednotlivé obrázky jsou uloženy v tabulce `Predictions`. Tato tabulka obsahuje název třídy a pravděpodobnost, s jakou obrázek patří do dané třídy. Dále obsahuje také navigační vlastnosti (anglicky Navigation property), které vytváří reference s modely.

Druhou tabulkou, obsahující informace o modelech, je tabulka `PretrainedModels`. V této tabulce jsou informace o modelech, které jsou již předtrénované a dostupné v nástroji Keras.NET, z nichž některé byly zmíněny v kapitole 1.5.4. Uloženy jsou

zde názvy modelů, rozměry vstupní vrstvy a parametry Top-1 přesnost a Top-5 přesnost.

K předtrénovaným modelům jsou přiřazeny obrázky, nad kterými byla provedena predikce daného modelu. Údaje o nich udržuje tabulka `PretrainedModelImages`. Tato tabulka je téměř shodná s tabulkou `TrainedModelImages`, rozdíl je pouze v navigační vlastnosti, který zde odkazuje na již předtrénovaný model z tabulky `PretrainedModels`. Tyto obrázky mají obdobně jako obrázky přiřazené k uživatelským modelům z tabulky `TrainedModelImages` přiřazeny predikce z tabulky `Predictions`.

`UntrainedModels` obsahuje údaje o modelech, které byly pouze vytvořeny, ale ještě s nimi nebylo provedeno žádné trénování. K těmto modelům jsou přiřazeny vrstvy, které modely obsahují, z tabulky `Layers`.

`CurrentlyTrainedModels` je poslední tabulkou s informacemi o modelech. V této tabulce jsou informace o modelech, na nichž je právě prováděno trénování, přičemž nejdůležitějším údajem je umístění souboru se záznamy o trénování.

V databázi figurují především 1:N, resp. N:1 vztahy mezi tabulkami. To znamená, že 1 záznam z první tabulky má přiřazeno N záznamů z druhé tabulky, resp. N záznamů z druhé tabulky má přiřazeno 1 záznam z tabulky první. V praxi to znamená, že například 1 nenatrénovaný model z tabulky `UntrainedModels` má přiřazeno N vrstev z tabulky `Layers`, resp. N vrstev má přiřazeno 1 nenatrénovaný model.

Při řešení byl zvolen přístup anglicky označovaný jako „Code first“, kdy jsou nejprve pomocí kódu definovány jednotlivé třídy, jež reprezentují tabulky v databázi, a vztahy mezi nimi. Toto je patrné na výpisu 4.1.

Výpis 4.1: Vytvoření relace 1:N, resp. N:1.

```
1  public class UntrainedModelEntity : EntityBase
2  {
3      public string Name { get; set; }
4      public virtual List<LayerEntity> Layers { get; set; }
5  }
6
7  public class LayerEntity : EntityBase
8  {
9      public string LayerDetailsJson { get; set; }
10     public virtual UntrainedModelEntity UntrainedModel { get; set; }
11 }
```

Zde jsou vytvořeny třídy `UntrainedModelEntity` a `LayerEntity`, reprezentující záznamy ze stejnojmenných tabulek, které dědí ze třídy `EntityBase`. Vztah 1:N mezi

tabulkami, na které jsou tyto třídy mapovány, je vytvořen pomocí přidání kolekce typu `List<LayerEntity>` jako virtuální vlastnosti na jedné straně a přidáním virtuální vlastnosti typu `UntrainedModelEntity` na straně druhé.

Prvním krokem samotné implementace tedy bylo vytvořit třídy, které budou sloužit jako modely pro objektově relační mapování. Nejprve byla vytvořena abstraktní třída `EntityBase`. Tato třída má pouze vlastnost `Id`, která je inicializován při vytvoření instance této třídy. Tato vlastnost slouží jako primární klíč pro databázové tabulky. Pro takovéto vlastnosti obsahuje platforma .NET třídu Guid, která generuje 128 bitů dlouhý unikátní identifikátor. Z rodičovské třídy `EntityBase` dědí všechny třídy, které slouží jako modely tabulek k objektově relačnímu mapování. Podle návrhu databáze byly vytvořeny třídy stejné jako jsou uvedené na obrázku 4.1.

Následně byl vytvořen databázový kontext (anglicky DatabaseContext), který slouží jako vrstva mezi databází a aplikací. Obsah databázových tabulek je zde reprezentován jako kolekce typu `DBSet`. Toto je naznačeno na úryvku 4.2, kde jsou v třídě `DatabaseContext` vlastnosti `TrainDatasets` a `TestDatasets`, které reprezentují obsah stejnojmenných tabulek . Zbývající tabulky zde pro přehlednost nejsou uvedeny. Kromě jednotlivých kolekcí je v metodě `OnConfiguring` nastaven databázový engine a umístění připojené databáze. Informace o názvu databáze a jejím umístění jsou převzaty ze souboru s nastaveními projektu `appsettings.json`.

Výpis 4.2: Databázový kontext

```
public class DatabaseContext : DbContext
{
    public DbSet<TrainDatasetEntity> TrainDatasets { get; set; }
    public DbSet<TestDatasetEntity> TestDatasets { get; set; }
    ...
     protected override void OnConfiguring(...)
    {
        ...
    }
}
```

Pro práci s daty byly dále vytvořeny tzv. repozitáře. Repozitáře fungují jako jakýsi prostředník mezi databázovým kontextem a ostatními vrstvami aplikace. Tento přístup skýtá poměrně mnoho výhod:

- Redukce počtu databázových dotazů.
- Databázové dotazy jsou volány pouze z repozitářů.
- Zjednodušené možnosti testování.

Repozitáře definují CRUD (Create, Read, Update, Delete) operace pro každou tabulku. CRUD jsou metody, které umožňují vytvoření nového záznamu v tabulce, přečtení záznamu z tabulky, upravení záznamu v tabulce a smazání záznamu z tabulky. Jako základ pro implementaci repozitářů bylo použito rozhraní (anglicky Interface) `IRepository`. V tomto generickém rozhraní, které jako generický parametr akceptuje jednu z tříd, které dědí z třídy `EntityBase`, jsou pouze deklarovány jednotlivé CRUD metody, což je ilustrováno ve výpisu 4.3. Toto rozhraní dále implementuje třída `RepositoryBase`, ve které jsou jednotlivé metody z rozhraní implementovány. Aby bylo možné co nejuniverzálnější použití této třídy, akceptuje tato třída dva generické parametry, jeden pro databázový kontext a jeden pro typ databázové entity. První generický parametr je použit, aby bylo možné jednoduše měnit databázový kontext a typicky pro účely testování nahradit hlavní databázi databází testovací. Druhý generický parametr udává, se kterou databázovou entitou bude pracovat.

Výpis 4.3: Rozhraní IRepository

```
1 public interface IRepository<T> where T : EntityBase
2 {
3     public Task Add(T entity);
4     public Task<List<T>> GetAll();
5     public Task<T?> GetById(Guid id);
6     public Task<T> Update(T entity);
7     public Task<T> Delete(Guid id);
8 }
```

Následně je možné pro potřebné entity vytvořit jejich dedikované repozitáře vytvořením tříd, které dědí ze třídy `RepositoryBase`, zadáním generických parametrů a zavoláním konstruktoru rodičovské třídy.

Posledním krokem byla migrace databáze, při které je vytvořena databáze s odpovídajícími tabulkami. Migrace je funkcionalita nástroje EF Core, která podle vytvořených entitních tříd, jejich vztahů a nastavení databázového kontextu vytvoří databázi.

4.2 Nahrání datasetu

Další ze základních funkcionalit této aplikace je nahrání datasetu uživatelem. Aby bylo možné nahrát dataset, je potřeba implementovat kontrolér, který bude přijímat nahrávaná data. Kontrolér zodpovídá za zpracování webových HTTP (HyperText Transfer Protocol) požadavků. HTTP je protokol aplikační vrstvy, který funguje na principu dotaz - odpověď. Kontrolér, který obsluhuje uzly (anglicky Endpoint) pro nahrání datasetů, naslouchá na URL adresách `/upload/train` a `/upload/test`.

Pokud dojde k HTTP požadavku na jednu z daných adres pomocí HTTP metody POST, kontrolér přijme z těla tohoto požadavku archiv, v němž je nahrávaný dataset. Následně kontrolér volá metody ze servisní vrstvy pro kontrolu tohoto souboru a uložení daného souboru. Zjednodušená implementace endpointu je naznačena ve výpisu 4.4.

Výpis 4.4: Endpoint pro nahrání testovacího datasetu.

```
[HttpPost("/upload/train/{name}")]
public async Task<IActionResult> UploadTrainDataset(IFormFile file)
    {
        try
        {
            await _uploadDatasetService.UploadDataset(file, name);
        }
        catch (Exception e)
        {
            return BadRequest(e.Message);
        }
        return Ok();
    }
```

Endpoint je implementován jako metoda `UploadTrainDataset`, která naslouchá na URL `upload/train/name`, kde `name` je proměnná s informací o názvu zadaném uživatelem pro daný dataset. Tento endpoint akceptuje HTTP metody typu POST. Archiv s datasetem je přenášen v objektu typu `IFormFile`, jenž je vstupem metody `UploadTrainDataset`. Následně je ze servisní třídy `UploadDatasetService` volána metoda pro uložení datasetu. Pokud v servisní třídě vše proběhne v pořádku, endpoint vrací HTTP odpověď typu 200 OK, která symbolizuje, že vše proběhlo v pořádku. Pokud dojde při ukládání k chybě a v servisní nebo repozitářové vrstvě je vyvolána výjimka, je tato výjimka ošetřena v bloku `catch` a endpoint vrátí chybovou HTTP odpověď 400 Bad Request nesoucí informace o výjimce.

V servisní vrstvě je nejprve archiv zkontrolován a uložen do uložiště serveru. Následně je archiv rozbalen do zvláštní složky s náhodným jménem, aby nedocházelo ke kolizím mezi datasety, pro které uživatel zvolí stejný název, a následně je záznam o tomto datasetu uložen do příslušné tabulky do databáze.

S takto připraveným kontrolérem a servisní vrstvou je možné implementovat uživatelské rozhraní, pomocí něhož bude uživatel umožněno vybrat soubor pro nahrání. Nejprve byla vytvořena stránka (Blazor page) `TrainingDataset.razor`, jejíž zjednodušená struktura je ve výpisu 4.5.

Na tomto příkladu lze vidět základní architekturu Blazor stránky. Na prvním

řádku je po klíčovém slově `@page` specifikována URL, na které je stránka dostupná. Klíčové slovo `@inject` vloží závislost pomocí techniky „Dependency injection“. Následuje část s Razor syntaxí, která obsahuje HTML značky a značky vytvořených komponent. Takto vytvořená Blazor komponenta je `UploadDatasetComponent` se vstupním parametrem adresy pro nahrání. Následuje cyklus `foreach`, který iteruje v kolekci `TrainDatasets` a pro každý vygeneruje komponentu `DatasetCard` sloužící pro vizualizaci jednotlivých datasetů. Po klíčovém slově `@code` následuje část ve které je možné volat klasický C# kód. Zde je nejprve deklarována kolekce `TrainingDatasets` a následně je tato kolekce naplněna daty z databáze v metodě `OnInitilized`, která je volána vždy, když je daná stránka inicializována, tzn. načtena v prohlížeči.

Výpis 4.5: Stránka pro nahrání testovacího datasetu.

```
@page "/Dataset/Train"
@inject TrainDatasetRepository TrainDatasetRepository
<h3>Training Datasets</h3>
<UploadDatasetComponent
                UploadUrl="@_uploadUrl"></UploadDatasetComponent>
    @foreach (var dataset in TrainDatasets)
    {
        <DatasetCard DatasetKind="train" Dataset="@dataset">
        </DatasetCard>
    }
@code
{
    private IEnumerable<TrainDatasetEntity> TrainingDatasets{get;set;}

    protected override async Task OnInitializedAsync()
    {
        TrainingDatasets = await TrainingDatasetRepository.GetAll();
    }
}
```

Po úspěšném nahrání datasetu jsou pod komponentou pro nahrání zobrazeny všechny dostupné datasety, které se na serveru nachází. Kliknutím na příslušný dataset je možné si prohlédnout jeho obsah. Po kliknutí jsou tedy zobrazeny jednotlivé složky, reprezentující třídy obrázků, které dataset obsahuje. Pro ilustraci těchto datasetů byla vytvořena komponenta `DatasetCard`, která vizuálně reprezentuje jednotlivé datasety pomocí jejich názvu a obrázku vybraného z datasetu.

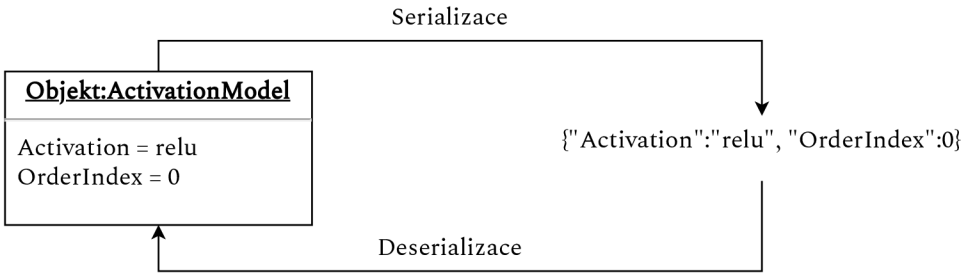
4.3 Vytvoření modelu

Před samotnou implementací tvorby modelu podle požadavků uživatele bylo potřeba nejprve vytvořit třídy pro jednotlivé vrstvy modelu podle dokumentace nástroje Keras.NET [27]. Tímto způsobem bylo vytvořeno více než 60 tříd se stejnými vlastnostmi jako mají jejich předlohy. Všechny tyto třídy dědí z rodičovské třídy `LayerBase`, která obsahuje název třídy, její pořadí a abstraktní metodu `MapToKerasLayer`. Tato metoda je implementována v každé třídě, která z této rodičovské třídy dědí a vrací příslušnou třídu z knihovny Keras.NET. Použít přímo třídy dostupné v knihovně Keras.NET nebylo možné z důvodu jejich pro toto řešení nevhodné implementace, kdy jednotlivé parametry tříd jsou ukládány do slovníku jako páry názvu a hodnoty daného parametru. Takovéto třídy by bylo problematické uložit do databáze a následně provést úpravy takto uložených vrstev modelu.

Dále bylo nutné vytvořit univerzální formulář, který bude generován podle potřebné třídy. Tento účel plní formulářová komponenta `LayerForm`. Tato komponenta přijímá jako vstupní parametr výše zmíněnou rodičovskou třídu reprezentující vrstvu modelu `LayerBase`. Jako parametr je tak možné vložit jakoukoliv třídu, který reprezentuje vrstvu modelu, předem však není možné určit, která konkrétní třída to bude. K tomuto účelu byla použita technika, která se nazývá reflexe (anglicky Reflection). Reflexe je součástí platformy .NET, která poskytuje nástroje pro práci s objekty za běhu programu. Díky této technologii je tak možné získat jednotlivé parametry objektu třídy, která vstupuje jako parametr formulářové komponenty, a dynamicky tak vytvořit formulář podle těchto parametrů.

Poté, co uživatel vytvoří kýžený model skládáním jednotlivých vrstev, je takovýto model potřeba uložit do databáze. Zde opět nastává stejný problém jako u vytváření modelu, kdy nebylo možné vytvořit jeden univerzální formulář, ale bylo potřeba generovat formulář dynamicky, protože jednotlivé třídy reprezentující vrstvy modelu jsou velmi různorodé. Tato různorodost poměrně limituje možnosti, jak vytvořit databázovou tabulku, do které by bylo možné uložit všechny existující třídy reprezentující vrstvy modelu. Před uložením tak bylo přistoupeno k serializaci do formátu JSON, která vstupní objekt převede na řetězec znaků, jenž je následně uložen do databáze. Serializace je naznačena na obrázku 4.2.

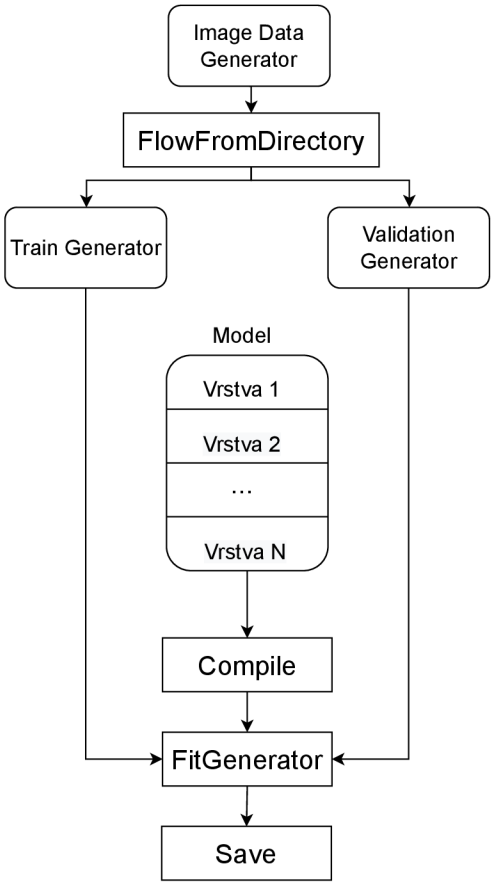
Aby bylo možné takto uložený model upravovat, je potřeba uložený řetězec znaků deserializovat z formátu JSON a vytvořit za běhu aplikace instanci třídy z uloženého řetězce. Zde byla opět využita reflexe, kdy je podle údajů uložených v databázi vytvořena nová instance odpovídající třídy a následně jsou tyto třídy reprezentující vrstvy seřazeny do správného pořadí. S takto seřazenými třídami je možné dynamicky vytvořit odpovídající formuláře, pomocí kterých je možné vrstvy modelu upravit.



Obr. 4.2: Serializace a deserializace objektu.

4.4 Trénování modelu

Trénování modelu bylo v této práci implementováno pomocí nástroje Keras, respektive Keras.NET, které byly zmíněny v kapitole 2.6. Před samotným trénováním uživatel nejprve zadá požadované parametry, poté spustí trénování a na závěr jsou mu prezentovány výsledky natrénovaného modelu. Graficky je průběh trénování modelu ilustrován na obrázku 4.3. Hlavní předností nástroje Keras je jednoduchost jeho použití. Aby uživatel mohl natrénovat model, stačí nastavit parametry několika objektů a metod. Prvním objektem je `ImageDataGenerator`, který generuje dávky tenzorů obrazových dat. Tento objekt má celou řadu parametrů, které je možné nastavit podle potřeb uživatele. Nad tímto objektem je následně zavolána metoda `FlowFromDirectory`, která umožňuje nahrání dat ze specifikované složky a nastavení některých dalších parametrů předzpracování dat jako je například změna rozměrů obrázků. Pomocí této metody jsou vytvořeny dva generátory dat. Trénovací generátor a validační generátor. Tyto generátory jsou vytvořeny z celého trénovacího datasetu v poměru, který je určen uživatelem. V dalším kroku je potřeba použít samotný model. Model uživatel vybere z nabídky uložených modelů, nebo si vytvoří nový. Po vytvoření modelu je potřeba jej zkompilovat. K tomu slouží metoda `Compile`. Kompilaci modelu uživatel opět nastaví pomocí požadovaných parametrů. Předposledním krokem je samotné učení modelu. Učení je provedeno pomocí metody `FitGenerator`, ve které jsou nastaveny trénovací data, validační data a mnoho dalších hodnot. Pro všechna zmíněná nastavení byly vytvořeny modely, podle kterých jsou vytvořeny jednotlivé formuláře. Spolu s trénováním je spuštěno také zaznamenávání průběhu trénování ve formátu, který požaduje aplikace TensorBoard. TensorBoard je součástí platformy TensorFlow a slouží ke sledování průběhu učení v reálném čase na různých vizualizacích. Po natrénování je model uložen pod vybraným jménem pomocí metody `Save`. Při ukládání jsou do dedikované tabulky uloženy také třídy obrázků, na kterých bylo provedeno trénování, a k danému záznamu je přiřazen odpovídající model.



Obr. 4.3: Diagram implementace hlubokého učení.

4.5 Predikce modelu

Aby uživatel mohl svůj natrénovaný model použít k predikcím, bylo nejprve nutné implementovat nahrávání obrázků. Pro zpracování nahrávaných obrázků byl vytvořen další kontrolér `ImageController`. Tento kontrolér přijímá HTTP akceptuje typu POST a umožňuje přijmout množinu obrázků, na kterých má být provedena predikce. Stejně jako u kontroléru, jenž zpracovává nahrávané datasety, i tento kontrolér má k dispozici servisní třídu, v tomto případě `UploadImagesService`. Tato třída implementuje veškerou funkcionalitu nutnou pro kontrolu nahrávaného obsahu a následné uložení jak do uložiště, tak do databáze. Při ukládání jsou obrázky přiřazeny k danému již natrénovanému modelu.

Před samotnou predikcí je potřeba obrázky převést do potřebné podoby, se kterou dokáže nástroj Keras.NET zpracovat. Pro tyto účely má Keras.NET třídu `ImageUtil`, která obsahuje potřebné metody pro načtení a převedení obrázku do podoby pole. Dále je načten požadovaný model, který má predikci provést, a nakonec je s každým obrázkem provedena predikce. Výstupem predikce je pole, které má

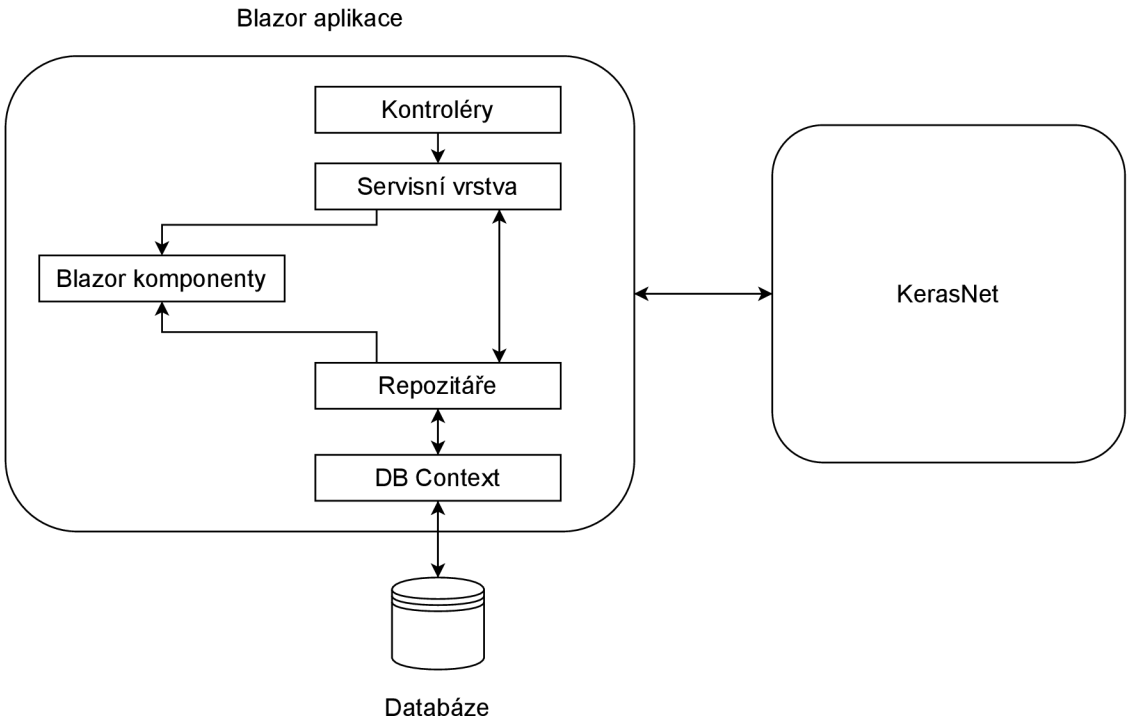
shodnou velikost, jako je počet možných tříd, do kterých obrázky náleží. Následně je podle uložených tříd pro daný model přiřazen slovní popis k nejpravděpodobnější predikci. Tato predikce je uložena do databáze s odkazem na příslušný obrázek k němuž náleží.

Kromě modelů, které byly vytvořeny uživatelem, jsou k dispozici také předtrénované modely dostupné v knihovně Keras.NET. Tabulka `PretrainedModels` obsahuje informace o 12 modelech již natrénovaných na datasetu ImageNet. K těmto modelům je možné stejně jako k uživatelským modelům nahrát libovolný počet obrázků k predikci. Po nezbytné transformaci obrázků je provedena samotná predikce. Výsledkem predikce je obdobně jako u uživatelských modelů pole pravděpodobností. U předtrénovaných modelů je ale výhodné, že všechny byly natrénovány na stejném datasetu, proto mají všechny stejné třídy, do kterých mohou obrázek zařadit. K dekódování predikcí tak lze použít již připravenou metodu z knihovny Keras.NET. Dvě nejpravděpodobnější predikce jsou uloženy do databáze a přiřazeny k příslušnému obrázku.

4.6 Architektura aplikace

Celková architektura aplikace je znázorněna na obrázku 4.4. Řešení se skládá ze dvou projektů (hlavní Blazor aplikace a knihovny KerasNet) a databáze. Blazor aplikace zajišťuje pomocí databázového kontextu komunikaci s databází, do níž jsou ukládána data potřebná pro běh aplikace. Nad databázovým kontextem byla vytvořena repozitářová nadstavba. Za pomocí těchto repozitářů jsou z databáze získávána data použitá v Blazor komponentách, které vytváří uživatelské rozhraní. Repozitáře však obsahují pouze základní CRUD aplikace, kvůli optimalizaci SQL dotazů do databáze. Pokud je tedy nutné provádět s databázovými daty složitější operace, jsou tato data nejprve načtena do servisní vrstvy, v níž jsou potřebné operace provedeny. Servisní vrstvu využívají také kontroléry zodpovědné za přijímání HTTP požadavků, v této konkrétní aplikaci za příjem nahrávaných souborů. Další operace s těmito soubory, jako jsou kontrola obsahu nebo například uložení, provádí opět některá z tříd servisní vrstvy.

Pro veškerou práci s knihovnou Keras.NET byl vytvořen druhý projekt KerasNet. Tento projekt obsahuje jak všechny modelové třídy použité pro mapování jednotlivých vrstev a nastavení, tak metody, které spouští samotné učení modelu nebo jeho predikce. K tomuto rozdělení bylo přistoupeno z důvodu přehlednosti, v průběhu vývoje aplikace se však toto řešení ukázalo jako nezbytné. Knihovna Keras.NET totiž na pozadí používá knihovnu Numpy.Bare, jejíž použití je problematické při vícevláknovém programování. Jak již bylo zmíněno v kapitole 2.2, aplikace ASP.NET využívají vícevláknového přístupu.



Obr. 4.4: Architektura aplikace.

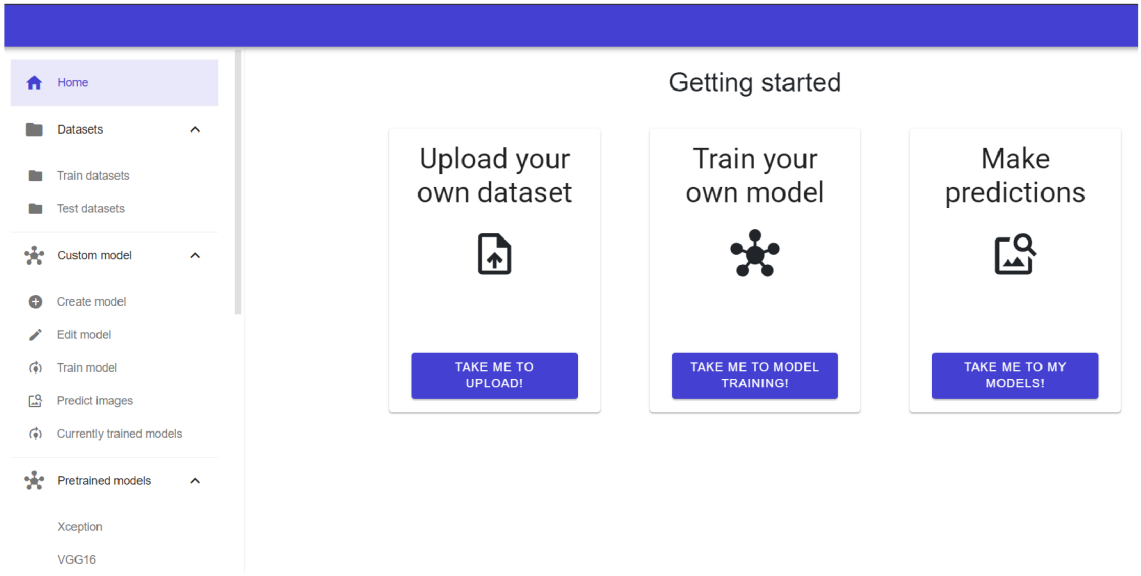
Z tohoto důvodu musí být veškerý kód, jenž využívá na pozadí knihovnu Numpy.Bare, opatřen zámkem GIL (Global Interpreter Lock). Tento zámek umožňuje pouze jednomu vláknu přistupovat k Python interpreteru, který knihovna Numpy.Bare používá, aby nedocházelo k uváznutím aplikace (anglicky Deadlock).

5 Vzhled a fungování aplikace

V předchozích kapitolách byl představen vývoj aplikace, v této kapitole bude prezentován její výsledný vzhled a funkčnost z pohledu uživatele.

5.1 Úvodní stránka

Úvodní stránka, ilustrovaná na obrázku 5.1, je první načtenou stránkou po spuštění aplikace. Je na ní patrné celkové rozložení aplikace s horním panelem a postranním panelem, který obsahuje menu.



Obr. 5.1: Úvodní stránka.

Úvodní stránka obsahuje malý rozcestník, který uživateli naznačuje, kde je vhodné začít. Pokud se rozhodne pro první možnost, bude přesměrován na stránku, kde může nahrát svůj dataset na server. V případě, že si vybere druhou možnost a bude chtít rovnou trénovat model, bude přesměrován na stránku, která mu umožní natrénovat model. V posledním případě bude přesměrován na stránku, která slouží pro klasifikaci snímků pomocí vybraného modelu neuronové sítě.

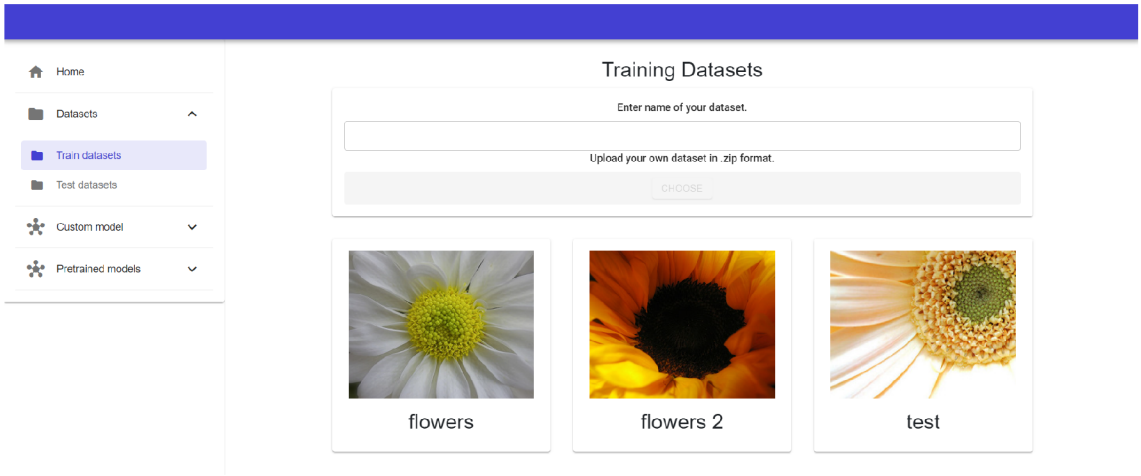
Na obrázku 5.2 je také možné vidět plně rozvinuté boční menu, které obsahuje oblasti:

- `Home`,
- `Datasets`,
- `Custom model`,
- `Pretrained models`.

Tlačítko `Home` vede na úvodní stránku, která je na obrázku 5.1. Oblast `Datasets` obsahuje dvě položky: `Training datasets` a `Test datasets`. Stránka `Training datasets` byla prezentována na obrázku 5.2. Stránka `Test datasets` je téměř identická jako stránka `Training datasets`. Rozdíl je pouze v tom, že požadavky jsou posílány na jinou adresu kontroléru a dataset poté také uložen na jiné místo na serveru a záznam o něm do jiné příslušné tabulky v databázi. Sekce `Custom model` a `Pretrained models` budou blíže popsány v následující kapitole.

5.2 Datasety

Pokud uživatel vybral první možnost a chce nejprve nahrát svůj dataset, je zobrazena stránka, která je na obrázku 5.2.



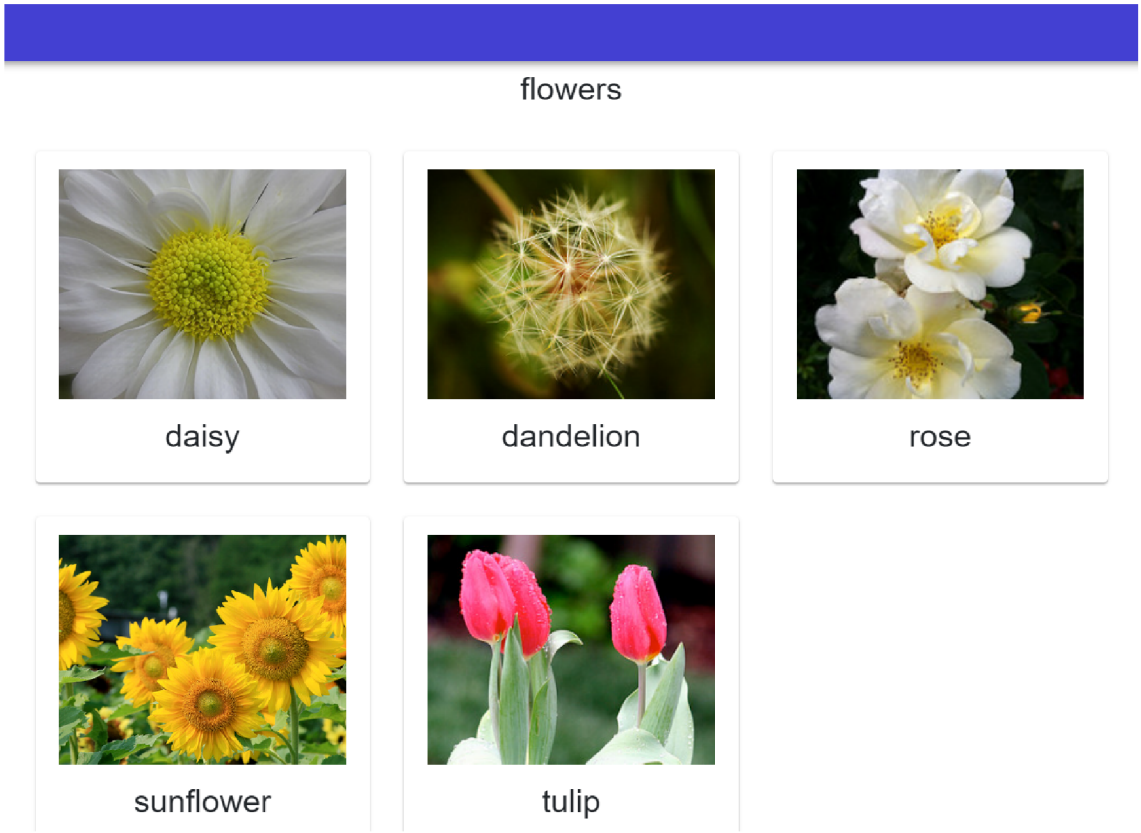
Obr. 5.2: Stránka s nahráním datasetu.

Zde je ve výchozím stavu nemožné kliknout na tlačítko `CHOOSE`. Uživatel musí nejprve vyplnit název svého datasetu a až poté je možné stisknout toto tlačítko. Po stisku se objeví klasické systémové vyskakovací okno, kde uživatel může vybrat jeden soubor, který bude nahrán na server. Po vybrání souboru se ve spodní části komponenty, jejíž součástí je textový vstup a tlačítko, objeví lišta s průběhem načítání. Objeví se také tlačítko `CANCEL`, kterým uživatel může přerušit nahrávání. Po vybrání souboru je poslán HTTP požadavek na příslušnou adresu kontroléru, jsou zkontrolovány všechny náležitosti a v případě úspěšného ověření je dataset uložen na server.

Pod komponentou, která umožňuje nahrávání, jsou ilustrovány všechny dostupné trénovací datasety. Po kliknutí na vybraný dataset se zobrazí stránka, kde jsou obdobně ilustrovány jednotlivé složky daného datasetu, tzn. skupiny obrázků patřící do jedné třídy.

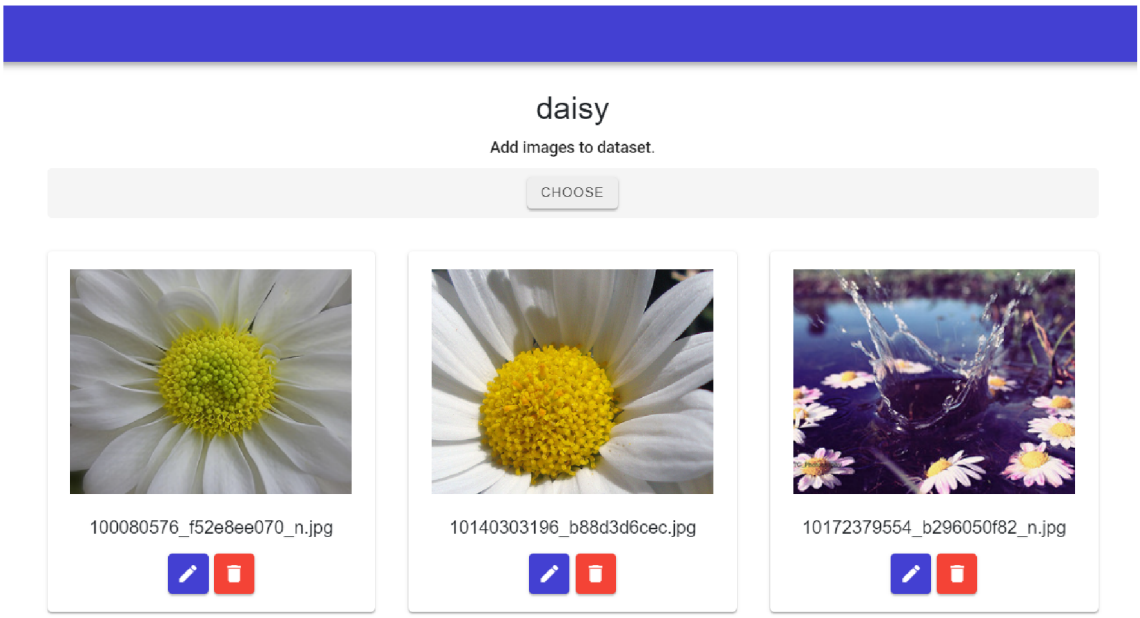
Pro ověření funkčnosti řešení byl na server nahrán dataset s názvem „Flowers Recognition“ [29]. Tento dataset obsahuje celkově 4242 obrázků květin, které lze použít pro natrénování neuronové sítě pro rozpoznávání druhů květin v obrázcích. Dataset je, stejně na obrázku 5.3, rozdělen do pěti složek, které reprezentují jednotlivé třídy:

- `daisy` – kopretina,
- `dandelion` – pampeliška,
- `rose` – růže,
- `sunflower` – slunečnice,
- `tulip` – tulipán.



Obr. 5.3: Dataset „Flowers recognition“.

Kliknutím na vybranou složku je zobrazen obsah této složky, jak je ilustrováno na obrázku 5.4. Uživatel má možnost nahrát do vybrané složky libovolný počet obrázků, upravit název obrázku po kliknutí na modrou ikonu tužky či obrázek smazat po kliknutí na červenou ikonu odpadkového koše.



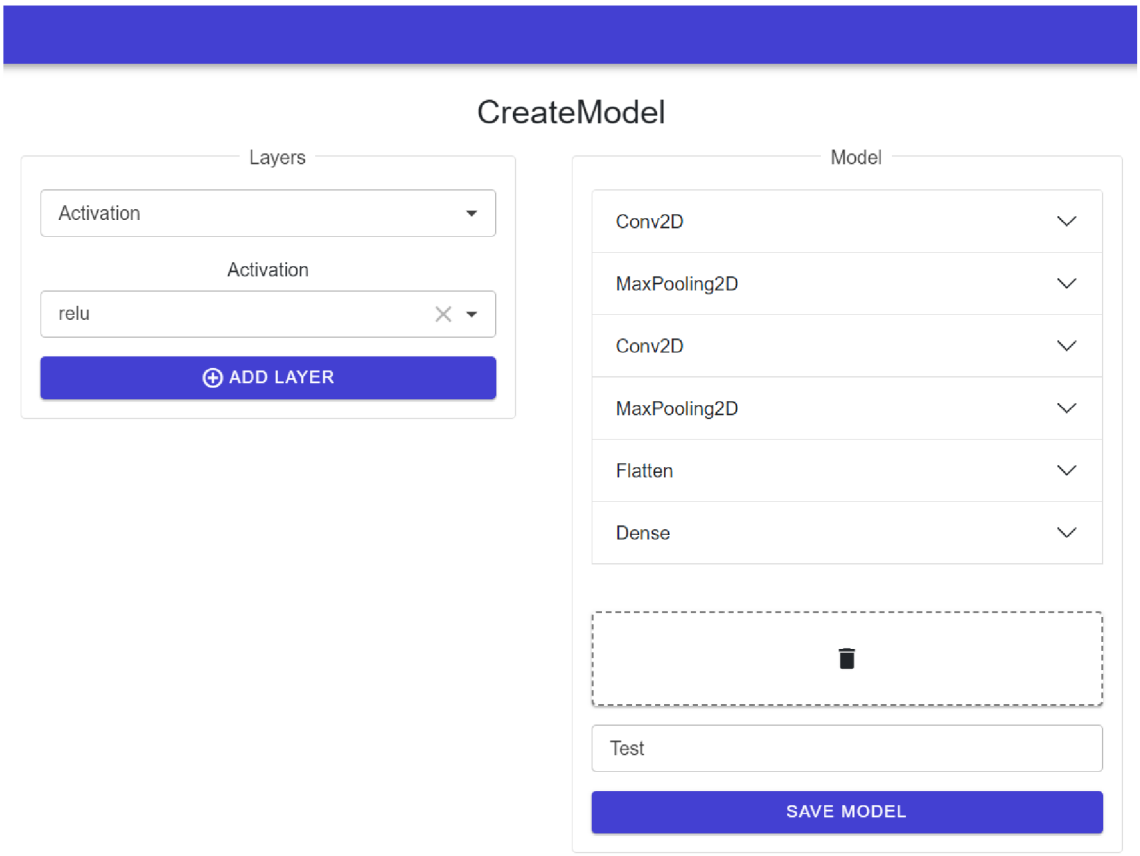
Obr. 5.4: Obsah datasetu.

5.3 Uživatelský model

V této kapitole budou představeny funkcionality související s uživatelským modelem, dostupné v menu `Custom model`.

5.3.1 Tvorba modelu

Po kliknutí na položku bočního menu `Custom model`, je pole rozvinuto a jsou zobrazeny další položky. Kliknutím na `Create model` je uživatel přesměrován na stránku, která je ilustrována na obrázku 5.5. V levé části uživatel z horní rolovací lišty nejprve vybere požadovanou vrstvu modelu a následně vyplní potřebné parametry. V ilustračním případě vybral vrstvu Activation a jako aktivační funkci zvolil ReLu. Po kliknutí na tlačítko `ADD LAYER` je vrstva přidána do pravé části, ve které je zobrazen vytvářený model. Vrstvy jsou skládány pod sebe jako HTML komponenty „Accordion“. Tuto komponentu lze po kliknutí rozvinout, tím zobrazit jednotlivé atributy dané vrstvy a upravit požadované hodnoty. Lze také měnit pořadí vrstev. Přetáhnutím vybrané vrstvy na vrstvu druhou je pořadí těchto vrstev zaměněno. Smazat lze požadovanou vrstvu přetáhnutím do oblasti s ikonou odpadkového koše, která se nachází pod poslední vrstvou modelu. Po zadání názvu v dolní části je aktivováno tlačítko `SAVE MODEL`, po jehož stisknutí je model uložen.



Obr. 5.5: Vytváření modelu.

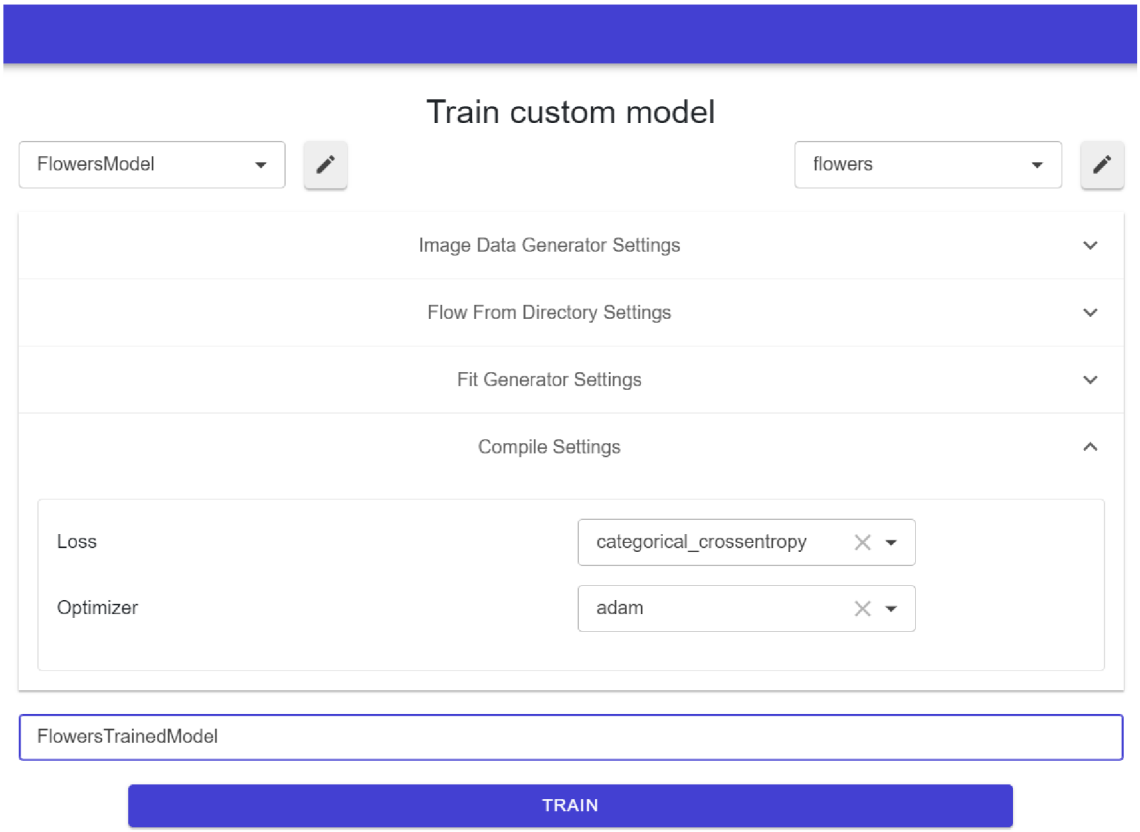
5.3.2 Úprava modelu

Upravovat vytvořené, dosud nenatrénované modely lze po kliknutí na položku menu `Edit model`. Zde uživatel vybere požadovaný model z rolovací lišty a následně je mu zobrazeno rozhraní velmi podobné tomu z obrázku 5.5. Po provedených úpravách má uživatel možnost zvolit mezi tlačítky `UPDATE`, nebo `SAVE AS NEW`.
Pokud si vybere první možnost, model bude uložen pod původním jménem se změnami, které uživatel provedl. Pokud se uživatel rozhodne pro druhou možnost, po zadání názvu bude model uložen jako nový unikátní model.

5.3.3 Trénování modelu

Po kliknutí na tlačítko `Train model` se uživateli zobrazí stránka, která je na obrázku 5.6. Nejprve uživatel vybere model, na kterém bude provedeno trénování. To umožňuje rolovací nabídka vlevo. Po kliknutí na rolovací komponentu vpravo, jež uživatele vybízí k vybrání datasetu, je možné zvolit vybraný dataset. V obou komponentách umožňujících výběr může uživatel vyhledávat podle jména. Pokud by před

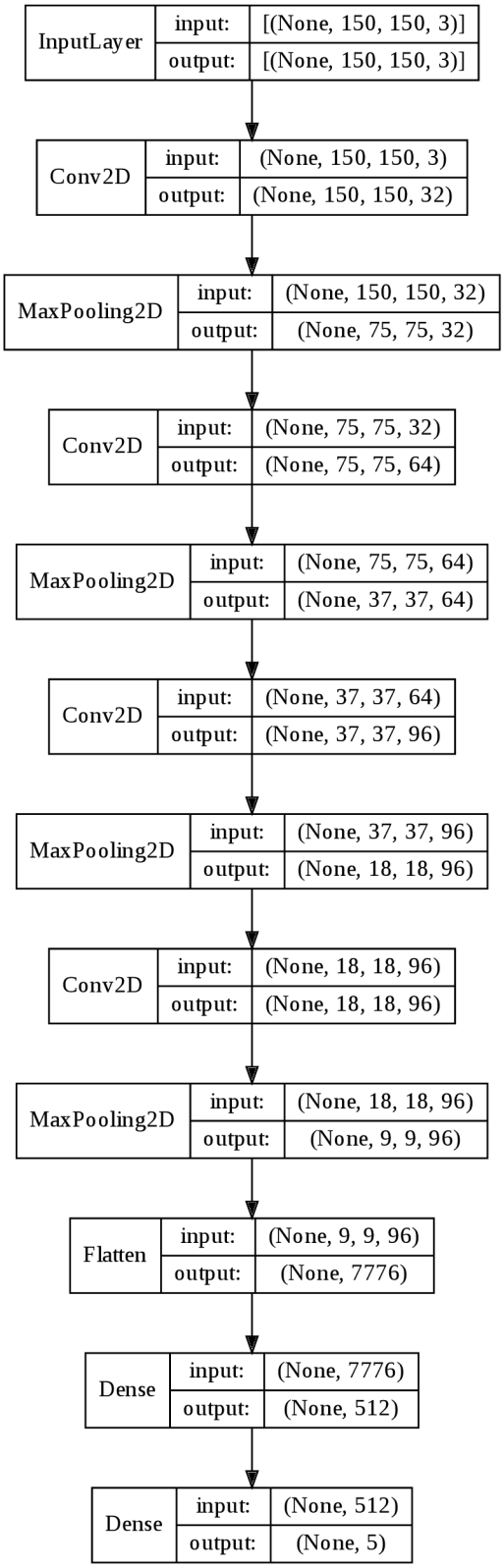
trénováním chtěl uživatel provést změny modelu, může tak učinit po kliknutí na ikonu tužky vedle výběru modelu. Následně bude přesměrován na stránku umožňující úpravu modelu, jenž byla popsána v předchozí kapitole, kde bude automaticky načten konkrétní model. Stejná ikona tužky se nachází i vedle výběru datasetu. V tomto případě je uživatel přesměrován na stránku s konkrétním datasetem, kde může provést úpravy obrázků v datasetu.



Obr. 5.6: Trénování modelu.

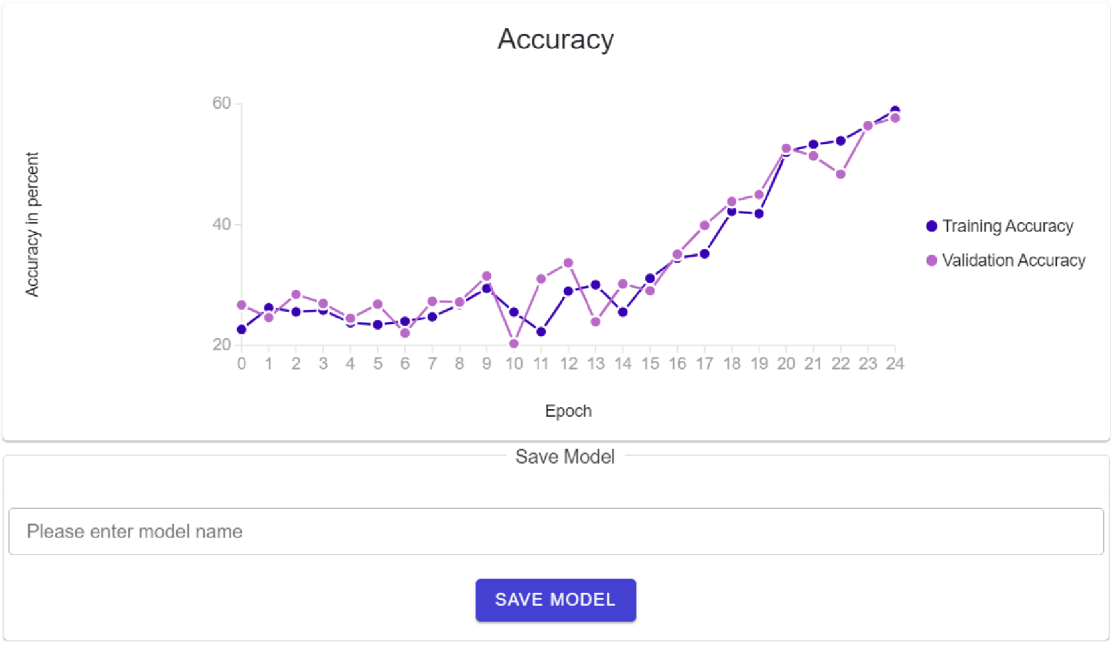
Jak bylo naznačeno na diagramu 4.3, dále je potřeba pomocí uživatelských vstupů nastavit jednotlivé objekty nebo metody. Stránka proto obsahuje lišty: `Image Data Generator Settings`, `Flow From Directory Settings`, `Fit Generator Settings`, `Compile Settings`.

Po kliknutí na příslušnou lištu se lišta rozbalí a je zobrazen formulář, do kterého je potřeba vyplnit hodnoty vstupních parametrů podle daného typu nastavení. Poslední povinnou položkou je textový vstup pod nastaveními, určený pro zadání názvu modelu. Po zadání je možné zahájit trénování kliknutím na tlačítko `TRAIN`. Samotný model vytvořený pro ověření funkčnosti řešení byl vytvořen podle příkladu, který je volně k dispozici u výše zmíněného datasetu „Flower recognition“. Jedná se o konvoluční neuronovou síť a její zjednodušená struktura je následující [30]:



Obr. 5.7: Zjednodušená struktura neuronové sítě.

Díky tomu, že byla zachována pojmenování, která používá nástroj Keras, je pro uživatele snadné v dokumentaci tohoto nástroje vyhledat jednotlivé vstupní veličiny a zjistit jejich význam a potřebné nastavení. Po spuštění trénování modelu je otevřeno dialogové okno informující uživatele, že trénování bylo zahájeno. Toto dialogové okno obsahuje tlačítko, po jehož stisknutí je uživatel přesměrován na stránku `Currently trained models` obsahující modely, na nichž momentálně probíhá trénování. Po kliknutí na vybraný model je spuštěna aplikace TensorBoard a uživatel je do této aplikace přesměrován v nové kartě prohlížeče. V případě, že uživatel během trénování nezavřel kartu prohlížeče, v níž bylo spuštěno trénování, jsou dole pod sekcí s nastaveními zobrazeny grafy s informacemi o trénování modelu. V nich jsou parametry přesnosti a hodnot chybové funkce na trénovacích a testovacích datech během jednotlivých epoch učení. Uživatel tak pomocí těchto parametrů může odvodit, zda potřebuje jeho model či nastavení trénování nějaké úpravy a tyto úpravy při příštím trénování také provést. Výsledky trénování jsou na obrázku 5.8. V případě, že uživatel v průběhu trénování zavřel kartu prohlížeče, může si informace o průběhu trénování zobrazit po výběru modelu pro predikci.

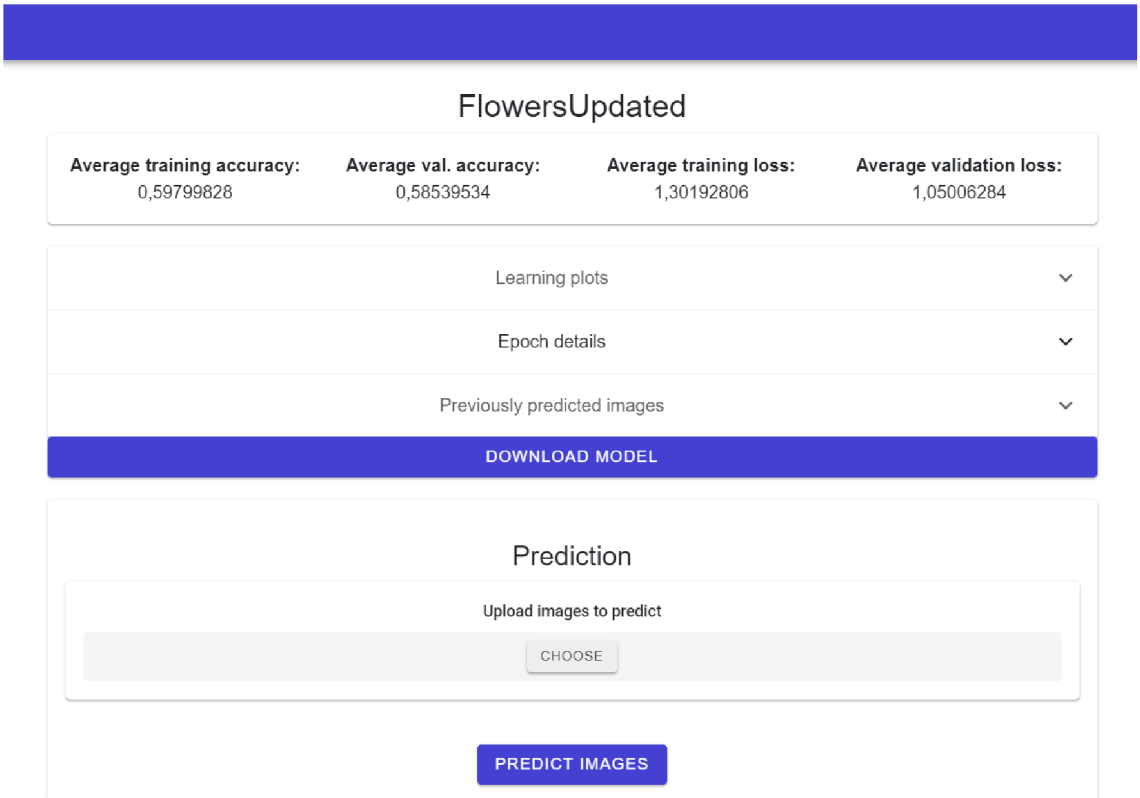


Obr. 5.8: Výsledky trénování modelu.

5.3.4 Predikce modelu

Po natrénování a uložení modelu je možné pomocí tohoto modelu klasifikovat libovolný obrázek. Kliknutím na položku `Predict images` v postranním menu se uži-

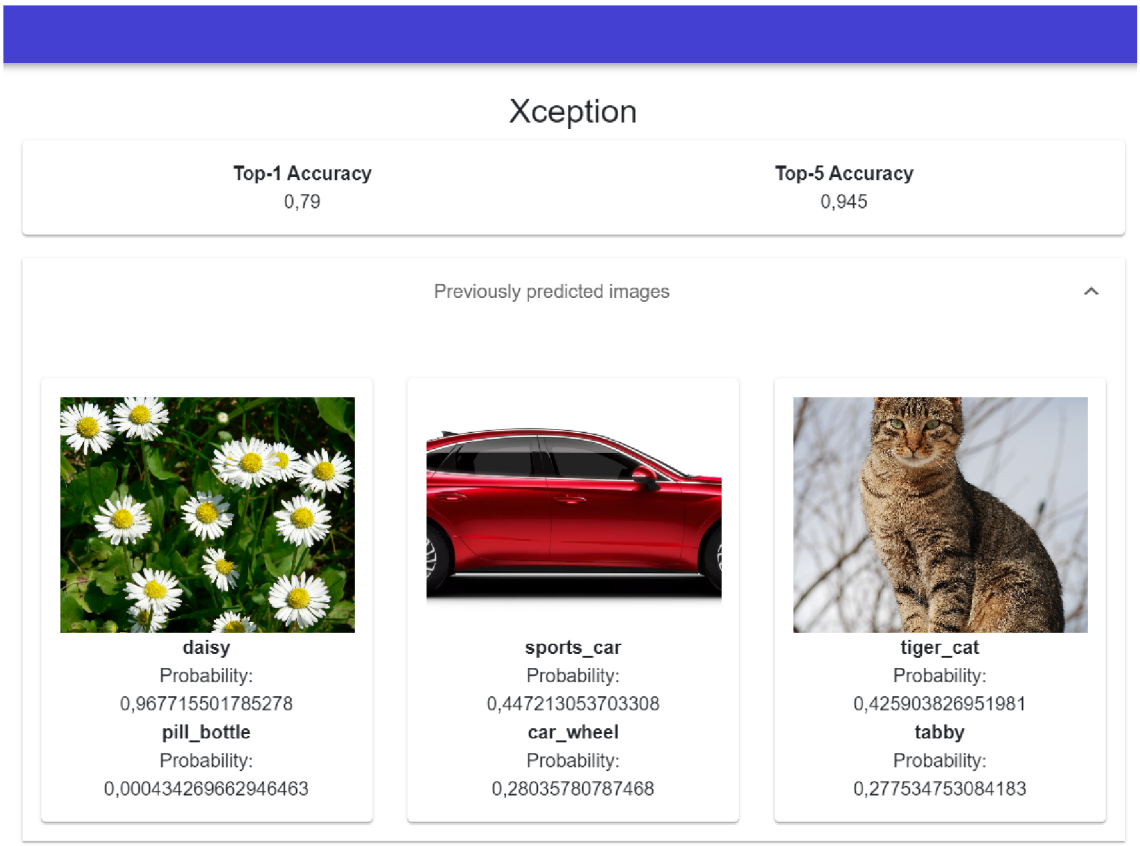
vatel dostane na stránku se všemi dostupnými natrénovanými modely. U každého modelu je uveden název, počet epoch trénování, které byly s modelem provedeny, průměrnou přesnost na trénovacích a validačních datech. Po kliknutí na vybraný model je uživateli zobrazena stránka jako na obrázku 5.9.



Obr. 5.9: Natrénovaný model.

V horní části jsou uvedeny základní parametry natrénovaného modelu. Po rozkliknutí záložky `Learning plots` jsou zobrazeny grafy vývoje přesnosti a hotnot loss funkce během jednotlivých epoch učení. V záložce `Epoch details` jsou tyto údaje uvedeny ještě v tabulce, aby si uživatel mohl vybrat preferovanou podobu dat. Záložka `Previously predicted images` po rozkliknutí zobrazí obrázky, s nimiž již predikce proběhla a výsledky této predikce. Natrénovaný model lze také vyexportovat a stáhnout do počítače. Po stisknutí tlačítka `DOWNLOAD MODEL` je model uložený ve formátu H5 zabalen do archivu ZIP a uživatel si jej pomocí vyskakovacího okna uložit na disk svého počítače. Nové obrázky lze nahrát po kliknutí na tlačítko `CHOOSE`. Po nahrání jsou obrázky zobrazeny pod komponentou pro nahrávání a je aktivováno tlačítko `PREDICT IMAGES`. Po jeho stisknutí následuje predikce modelu. Po provedení predikce je možné výsledky predikce zobrazit po rozkliknutí dříve zmíněné záložky `Previously predicted images`.

Kromě uživatelských modelů je možné provést predikce na 12 již předtrénovaných modelech, které obsahuje knihovna Keras.NET. Předtrénované modely jsou dostupné po rozvinutí pole `Pretrained models` v menu. Po kliknutí na vybraný model je uživateli zobrazena stránka velmi podobná jako v případě predikování pomocí uživatelského modelu. V horní části jsou informace o Top-1 a Top-5 přesnosti modelu, dále je možné nahrát obrázky určené ke klasifikaci a spustit predikci. Ukázka stránky s možností predikce pomocí modelu Xception, na které jsou vidět dříve klasifikované obrázky, je na obrázku 5.10.



Obr. 5.10: Predikce modelu Xception.

6 Závěr

V rámci této diplomové práce byla nejprve teoreticky popsána základní problematika umělé inteligence, strojového učení, neuronových sítí a jejich variant. Následně byly představeny stěžejní nástroje použité při vývoji aplikace. Poté byl vytvořen teoretický návrh možného řešení a na základě tohoto návrhu byla implementována webová aplikace pro trénování neuronových sítí. Toto řešení se skládá z databáze a samotné webové aplikace. Jedná se o Blazor Server aplikaci, která využívá knihovnu EF Core pro komunikaci s databází pomocí objektově relačního mapování. Hluboké učení bylo implementováno pomocí nástroje Keras.NET. Uživatelské rozhraní bylo implementováno pomocí komponent dostupných v knihovně Radzen, HTML a CSS. Aplikace umožňuje nahrání jak trénovacího, tak testovacího datasetu, přičemž je ověřeno dodržení správného formátu souborů. Dále je uživateli umožněna tvorba vlastního modelu skládáním jednotlivých vrstev. Po uložení tohoto modelu je možné tento model kdykoliv upravit. Před samotným trénováním je uživateli umožněn výběr konkrétního datasetu a modelu z databáze. Následně může nastavit parametry, které ovlivňují jak předzpracování dat, tak samotný průběh trénování modelu. Po natrénování je model uložen a uživateli jsou prezentovány výsledky trénování modelu během jednotlivých epoch. Výsledky jsou zobrazeny v přehledných grafech, případně v tabulkách, které obsahují hodnoty přesnosti a chybové funkce na trénovacích datech a validačních datech. Uložený natrénovaný model je možné použít pro zpracování uživatelem vybraného obrázku, který na server nahraje. Kromě uživatelem vytvořených natrénovaných modelů je možné provést predikci také na předtrénovaných modelech, které jsou dostupné v knihovně Keras.NET, mezi něž patří například model Xception či různé verze modelu VGG.

Literatura

1. ONGSULEE, Pariwat. Artificial intelligence, machine learning and deep learning. In: *2017 15th International Conference on ICT and Knowledge Engineering (ICTKE)*. 2017, s. 1–6. Dostupné z DOI: 10.1109/ICTKE.2017.8259629.
2. GUPTA, N.; MANGLA, R. *Artificial Intelligence Basics: A Self-Teaching Introduction*. 1. vyd. Mercury Learning & Information, 2020. ISBN 9781683925149. Dostupné také z: https://www.proquest.com/docview/2373544443/763EAB950E14420APQ/1?accountid=17115.
3. SHU-HSIEN LIAO. Expert system methodologies and applications—a decade review from 1995 to 2004. *Expert Systems with Applications*. 2005, roč. 28, č. 1, s. 93–103. ISSN 0957-4174. Dostupné z DOI: https://doi.org/10.1016/j.eswa.2004.08.003.
4. MUELLER, John Paul; MASSARON, Luca. *Machine Learning for Dummies*. 2. vyd. John Wiley & Sons, Incorporated, 2021. ISBN 9781119724063. Dostupné také z: https://www.proquest.com/docview/2478253935/19E9C5D3BF414217PQ/.
5. ZHOU, Lina; PAN, Shimei; WANG, Jianwu; VASILAKOS, Athanasios V. Machine learning on big data: Opportunities and challenges. *Neurocomputing*. 2017, roč. 237, s. 350–361. ISSN 0925-2312. Dostupné z DOI: https://doi.org/10.1016/j.neucom.2017.01.026.
6. MOHAMMED, Mohssen; KHAN, Muhammad Badruddin; BASHIER, Eihab Bashier Mohammed. *Machine Learning: Algorithms and Applications*. 1st Edition. CRC Press, 2016. ISBN 9781315371658. Dostupné také z: https://www.taylorfrancis.com/books/mono/10.1201/9781315371658/machine-learning-mohssen-mohammed-muhammad-badruddin-khan-eihab-bashier-mohammed-bashier.
7. *A typical neuron* [online]. [B.r.]. [cit. 2022-11-14]. Dostupné z: https://openbooks.lib.msu.edu/neuroscience/chapter/the-neuron/.
8. AGGARWAL, Charu C. *Neural Networks and Deep Learning: A Textbook*. Springer International Publishing AG, 2018. ISBN 978-3-319-94463-0. Dostupné také z: https://link.springer.com/book/10.1007/978-3-319-94463-0.
9. BUREŠ, Jaroslav. *Klasifikace obrazů planktonu s proměnlivou velikosti pomocí konvoluční neuronové sítě*. Brno, 2020. Dostupné také z: http://hdl.handle.net/11012/192511. Diplomová práce. Vysoké učení technické v Brně. Fakulta informačních technologií. Ústav počítačové grafiky a multimédií.

10. AGHDAM, Hamed Habibi; HERAVI, Jahani. *Guide to Convolutional Neural Networks*. 1. vyd. Springer Cham, 2017. ISBN 978-3-319-57550-6. Dostupné také z: https://link.springer.com/book/10.1007/978-3-319-57550-6.

11. SWAPNA, K. E. *Convolution Neural Network (CNN)* [online]. [B.r.]. [cit. 2023-05-02]. Dostupné z: https://developersbreach.com/convolution-neural-network-deep-learning/.

12. DENG, Jia; DONG, Wei; SOCHER, Richard; LI, Li-Jia; LI, Kai; FEI-FEI, Li. Imagenet: A large-scale hierarchical image database. In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, s. 248–255.

13. NAGDA, Rushabh. Evaluating models using the Top N accuracy metrics [online]. [B.r.] [cit. 2023-05-02]. Dostupné z: https://medium.com/nanonets/evaluating-models-using-the-top-n-accuracy-metrics-c0355b36f91b.

14. MATEEN, Muhammad; WEN, Junhao; NASRULLAH; SONG, Sun; HUANG, Zhouping. Fundus Image Classification Using VGG-19 Architecture with PCA and SVD. *Symmetry*. 2019, roč. 11, č. 1. ISSN 2073-8994. Dostupné z DOI: 10.3390/sym11010001.

15. SZEGEDY, Christian; LIU, Wei; JIA, Yangqing; SERMANET, Pierre; REED, Scott; ANGUELOV, Dragomir; ERHAN, Dumitru; VANHOUCKE, Vincent; RABINOVICH, Andrew. Going Deeper With Convolutions. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015.

16. RAJ, Bharath. *A Simple Guide to the Versions of the Inception Network*. [B.r.].

17. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. Deep Residual Learning for Image Recognition. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

18. CHOLLET, Francois. Xception: Deep Learning With Depthwise Separable Convolutions. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2017.

19. *Keras Applications* [online]. [B.r.]. [cit. 2023-05-02]. Dostupné z: https://keras.io/api/applications/.

20. *C#* [online]. [B.r.]. [cit. 2022-12-04]. Dostupné z: https://dotnet.microsoft.com/en-us/languages/csharp.

21. *ASP.NET* [online]. [B.r.]. [cit. 2022-12-04]. Dostupné z: https://dotnet.microsoft.com/en-us/apps/aspnet.

22. SPASOJEVIC, Marinko. *Asynchronous Programming with Async and Await in ASP.NET Core* [online]. [B.r.]. [cit. 2023-05-02]. Dostupné z: https://code-maze.com/asynchronous-programming-with-async-and-await-in-asp-net-core/.

23. *ASP.NET Core Blazor hosting models* [online]. [B.r.]. [cit. 2023-05-02]. Dostupné z: https://learn.microsoft.com/en-us/aspnet/core/blazor/hosting-models?view=aspnetcore-7.0.

24. *Radzen* [online]. [B.r.]. [cit. 2022-12-05]. Dostupné z: https://www.radzen.com/.

25. *Entity Framework Core* [online]. [B.r.]. [cit. 2022-12-04]. Dostupné z: https://learn.microsoft.com/en-us/ef/core/.

26. CHOLLET, François et al. *Keras* [https://keras.io]. 2015.

27. *Keras.NET* [online]. [B.r.]. [cit. 2022-12-04]. Dostupné z: https://scisharp.github.io/Keras.NET/.

28. Business Process Model and Notation (BPMN). In: [online]. December 2013 [cit. 2022-12-01]. Dostupné z: https://is.muni.cz/el/1433/jaro2014/PV165/um/46771256/pr_06_bpmn.pdf.

29. *Flowers Recognition* [online]. [B.r.]. [cit. 2022-12-04]. Dostupné z: https://www.kaggle.com/datasets/alxmamaev/flowers-recognition.

30. MEHROTRA, Raj. *Flower Recognition CNN Keras* [online]. [B.r.]. [cit. 2022-12-05]. Dostupné z: https://www.kaggle.com/code/rajmehra03/flower-recognition-cnn-keras/notebook.

Seznam symbolů a zkratek

AI	Artificial Intelligence
API	Application Programming Interface
BPMN	Business Process Model and Notation
CNN	Convolutional Neural Network
CPU	Central Processing Unit
CRUD	Create Read Upload Delete
CSS	Cascading Style Sheets
CT	Computed Tomography
GPU	Graphics Processing Unit
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
IoC	Inversion of Control
IoT	Internet of Things
JPEG	Joint Photographic Experts Group
ML	Machine Learning
MRI	Magnetic Resonance Imaging
MVC	Model View Controller
ORM	Object-Relational Mapping
PNG	Portable Network Graphics

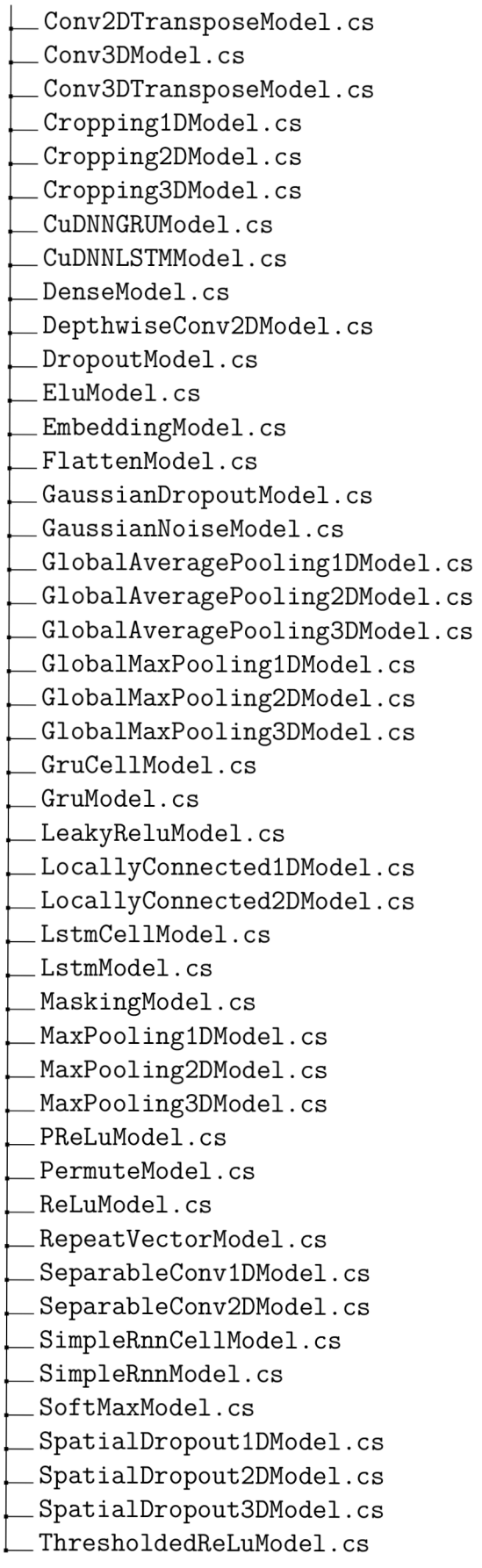
A Obsah elektronické přílohy

Elektronická příloha obsahuje projekt spolu vyvinutý v rámci této práce. K samotné aplikaci je přiložena také databáze `MLDatabase`. Před spuštěním je nutné v souboru `appsettings.json` nastavit umístění databáze vložením cesty do uvozovek v políčku `Path`, v sekci `DatabaseSettings`. Jako prerekvizity je nutné mít nainstalované následující aplikace:

- Python 3.8 – `https://www.python.org/downloads/`
- TensorFlow2 – `https://www.tensorflow.org/install`
- .NET SDK 6 – `https://dotnet.microsoft.com/en-us/download/dotnet/6.0`

Databáze obsahuje vzorový nenatrénovaný model, popsaný zde `https://www.kaggle.com/code/rajmehra03/flower-recognition-cnn-keras/notebook`. Dále také obsahuje informace o předtrénovaných modelech, které jsou nutné pro fungování aplikace. Dataset „Flowers Recognition“, jenž byl popisován v této práci je možné stáhnout zde: `https://www.kaggle.com/datasets/alxmamaev/flowers-recognition`. Projekt byl vyvíjen a testován na operačním systému Windows 10 verze 21H2.

```
MachineLearningWeb
└── KerasNet
    ├── DTOs
    │   └── EpochDto.cs
    ├── Enums
    │   └── PretrainedModelEnum.cs
    └── Models
        ├── LayerModelBases
        │   ├── ConvModelBase.cs
        │   ├── CuDNNBase.cs
        │   ├── GlobalPoolingBase.cs
        │   ├── GruBase.cs
        │   ├── LayerBase.cs
        │   ├── LstmBase.cs
        │   ├── MaxPoolingBase.cs
        │   ├── SeparableConvBase.cs
        │   └── SimpleRnnBase.cs
        └── LayerModels
            ├── ActivationModel.cs
            ├── ActivityRegularizationModel.cs
            ├── AlphaDropoutModel.cs
            ├── AveragePooling1DModel.cs
            ├── AveragePooling2DModel.cs
            ├── AveragePooling3DModel.cs
            ├── BatchNormalizationModel.cs
            ├── Conv1DModel.cs
            └── Conv2DModel.cs
```


Conv2DTransposeModel.cs
Conv3DModel.cs
Conv3DTransposeModel.cs
Cropping1DModel.cs
Cropping2DModel.cs
Cropping3DModel.cs
CuDNNGRUModel.cs
CuDNNLSTMModel.cs
DenseModel.cs
DepthwiseConv2DModel.cs
DropoutModel.cs
EluModel.cs
EmbeddingModel.cs
FlattenModel.cs
GaussianDropoutModel.cs
GaussianNoiseModel.cs
GlobalAveragePooling1DModel.cs
GlobalAveragePooling2DModel.cs
GlobalAveragePooling3DModel.cs
GlobalMaxPooling1DModel.cs
GlobalMaxPooling2DModel.cs
GlobalMaxPooling3DModel.cs
GruCellModel.cs
GruModel.cs
LeakyReluModel.cs
LocallyConnected1DModel.cs
LocallyConnected2DModel.cs
LstmCellModel.cs
LstmModel.cs
MaskingModel.cs
MaxPooling1DModel.cs
MaxPooling2DModel.cs
MaxPooling3DModel.cs
PReLuModel.cs
PermuteModel.cs
ReLuModel.cs
RepeatVectorModel.cs
SeparableConv1DModel.cs
SeparableConv2DModel.cs
SimpleRnnCellModel.cs
SimpleRnnModel.cs
SoftMaxModel.cs
SpatialDropout1DModel.cs
SpatialDropout2DModel.cs
SpatialDropout3DModel.cs
ThresholdedReLuModel.cs

```
│   │   │   ├── UpSampling1DModel.cs
│   │   │   ├── UpSampling2DModel.cs
│   │   │   ├── UpSampling3DModel.cs
│   │   │   ├── ZeroPadding1DModel.cs
│   │   │   ├── ZeroPadding2DModel.cs
│   │   │   └── ZeroPadding3DModel.cs
│   │   ├── SettingModels
│   │   │   ├── CompileSettings.cs
│   │   │   ├── FitGeneratorSettings.cs
│   │   │   ├── FlowFromDirectorySettings.cs
│   │   │   └── ImageDataGeneratorSettings.cs
│   │   └── TrainingResultModel.cs
│   ├── Services
│   │   ├── SystemProcessService.cs
│   │   └── TrainingDataService.cs
│   ├── KerasNet.csproj
│   └── KerasNetWrapper.cs
└── MachineLearningWebBlazor
    ├── Constants
    │   ├── CompileConstants.cs
    │   ├── FlowFromDirectoryConstants.cs
    │   ├── ImageDataGeneratorConstants.cs
    │   └── LayerConstants.cs
    ├── Controllers
    │   ├── DatasetController.cs
    │   └── ImageController.cs
    ├── Exceptions
    │   ├── EntityNotFoundException.cs
    │   └── UnsupportedFormatException.cs
    ├── Images
    │   └── blank_image.png
    ├── Mapping
    │   └── Mapper.cs
    ├── Models
    │   ├── DatasetFolderModel.cs
    │   ├── ImageModel.cs
    │   ├── LayerNameModel.cs
    │   └── NameModel.cs
    ├── Pages
    │   ├── CreateModel.razor
    │   ├── CurrentlyTrainedModels.razor
    │   ├── DatasetFolderContent.razor
    │   ├── DatasetFolderContent.razor.css
    │   ├── DatasetFolders.razor
    │   ├── DatasetFolders.razor.css
    │   └── EditModel.razor
```

```
│   ├── Error.cshtml
│   ├── Error.cshtml.cs
│   ├── Index.razor
│   ├── Index.razor.css
│   ├── PretrainedModel.razor
│   ├── TestDataset.razor
│   ├── TrainCustomModel.razor
│   ├── TrainedModel.razor
│   ├── TrainedModels.razor
│   ├── TrainingDataset.razor
│   ├── TrainingDataset.razor.css
│   ├── _Host.cshtml
│   └── _Layout.cshtml
├── Persistence
│   ├── Entities
│   │   ├── CurrentlyTrainedModel.cs
│   │   ├── DatasetEntityBase.cs
│   │   ├── EntityBase.cs
│   │   ├── EpochEntity.cs
│   │   ├── ImageEntityBase.cs
│   │   ├── LabelEntity.cs
│   │   ├── LayerEntity.cs
│   │   ├── PredictionEntity.cs
│   │   ├── PretrainedModelEntity.cs
│   │   ├── PretrainedModelImageEntity.cs
│   │   ├── TestDatasetEntity.cs
│   │   ├── TrainDatasetEntity.cs
│   │   ├── TrainedModelEntity.cs
│   │   ├── TrainedModelImageEntity.cs
│   │   └── UntrainedModelEntity.cs
│   ├── Repositories
│   │   ├── CurrentlyTrainedModelsRepository.cs
│   │   ├── IRepository.cs
│   │   ├── LayerRepository.cs
│   │   ├── PredictionRepository.cs
│   │   ├── PretrainedModelImagesRepository.cs
│   │   ├── PretrainedModelRepository.cs
│   │   ├── RepositoryBase.cs
│   │   ├── TestDatasetRepository.cs
│   │   ├── TrainDatasetRepository.cs
│   │   ├── TrainedModelImagesRepository.cs
│   │   ├── TrainedModelRepository.cs
│   │   └── UntrainedModelRepository.cs
│   └── DatabaseContext.cs
└── Properties
    └── launchSettings.json
```

```
├── Services
│   ├── DownloadService.cs
│   ├── GalleryService.cs
│   ├── LayerService.cs
│   ├── PredictionService.cs
│   ├── TrainedModelService.cs
│   ├── UntrainedModelService.cs
│   ├── UploadDatasetService.cs
│   └── UploadImagesService.cs
├── Shared
│   ├── AccuracyChart.razor
│   ├── AddLayerComponent.razor
│   ├── CompileSettingsForm.razor
│   ├── CompileSettingsForm.razor.css
│   ├── CreateModelComponent.razor
│   ├── DatasetCard.razor
│   ├── DatasetCard.razor.css
│   ├── DatasetFolder.razor
│   ├── DatasetFolder.razor.css
│   ├── DatasetPicker.razor
│   ├── DatasetPicker.razor.css
│   ├── FitGeneratorSettingsForm.razor
│   ├── FitGeneratorSettingsForm.razor.css
│   ├── FlowFromDirectorySettingsForm.razor
│   ├── FlowFromDirectorySettingsForm.razor.css
│   ├── ImageDataGeneratorSettingsForm.razor
│   ├── ImageDataGeneratorSettingsForm.razor.css
│   ├── IndexPageCard.razor
│   ├── LayerForm.razor
│   ├── LossChart.razor
│   ├── MainLayout.razor
│   ├── MainLayout.razor.css
│   ├── NavMenu.razor
│   ├── NavMenu.razor.css
│   ├── RenameDialog.razor
│   ├── SurveyPrompt.razor
│   ├── UploadDatasetComponent.razor
│   ├── UploadDatasetComponent.razor.css
│   └── UploadImagesComponent.razor
├── wwwroot
├── App.razor
├── MachineLearningWebBlazor.csproj
├── Program.cs
├── _Imports.razor
├── appsettings.Development.json
└── appsettings.json
```

```
├── MLDatabase
└── MachineLearningWebBlazor.sln
```