Jihočeská univerzita v Českých Budějovicích

Přírodovědecká fakulta

Segmentace včel na obrázcích plástů

Bakalářská práce

Sergei Filitov

Školitel: Ing. Miroslav Skrbek, Ph.D.

Konzultant: RNDr. Alena Bruce, Ph.D.

České Budějovice 2023

Bibliografické údaje:

Filitov, S., 2023: Segmentace včel na obrázcích plástů. [Bee segmentation on honeycomb images. Bc. Thesis, in Czech.] – 71 p., Faculty of Science, The University of South Bohemia, České Budějovice, Czech Republic.

Anotace:

Cílem bakalářské práce je návrh a implementace programového vybavení pro segmentaci včel na obrázcích plástů. V rámci této práce byly zohledněné existující segmentační řešení a technika dynamického okna spolu machine-learning modelem (klasifikátorem), a implementována obě řešení.

Klíčová slova:
YOLO, Dynamické okno, Hluboké konvoluční sítě, Hluboké reziduální sítě, Python, Segmentace, Včela.

Annotation:

The aim of the bachelor's thesis is the design and implementation of software for the segmentation of bees on images of honeycombs. As part of this work, the existing segmentation solution and the dynamic window technique together with a machine-learning model (classifier) were taken into account, and both solutions were implemented.

Keywords:
YOLO, Dynamic window, Deep convolutional networks, Deep residual networks, Python, Segmentation, Bee.

Prohlašuji, že jsem autorem této kvalifikační práce a že jsem ji vypracoval pouze s použitím pramenů a literatury uvedených v seznamu použitých zdrojů.

V Českých Budějovicích dne: _____ Podpis autora: _____

Poděkování:

Rád bych touto cestou poděkoval Ing. Miroslavu Skrbkovi, Ph.D. za jeho odborné vedení a konzultace ohledně implementace a RNDr. Aleně Bruce, Ph.D. za její cenné rady ve včelí problematice.

Samozřejmě bych rád poděkoval své rodině za podporu ve studiu.

Obsah

1 Úvod

Medonosné včely jsou zásadní pro zemědělství a hrají klíčovou roli v životním cyklu rostlin jako jeden z nejdůležitějších opylovačů. Více než 75 % ze 115 nejpopulárnějších druhů plodin na celém světě je závislých na opylování zvířaty nebo z něj alespoň těží. Včely jsou zodpovědné za opylení přibližně 70 % všech druhů plodin na celém světě. Především z tohoto důvodu existují vážné ekonomické důsledky, když včely bojují s patogeny, parazity, škůdci a dalšími faktory ovlivňujícími přežití včelstva. [14]

Existují nevysvětlené případy poruchy zhroucení včelstev (**colony collapse disorder** nebo **CCD**). Ztráty včelstev jsou považovány za multifaktoriální problém, kombinující vliv biotických a abiotických stresorů, např. výživy, pesticidů a změny klimatu. Znalost fyziologického a imunologického stavu včelstev je proto klíčová při pomoci včelařům čelit potenciálním problémům a chránit včely. [14]

Většina CCD se vyskytuje v zimě, což je pro včelstva nejrizikovější období. Proto je nanejvýš důležité studovat změny v organismu včely medonosné v zimním období. [14]

Automatizovaná segmentace včel na obrázcích plástů hraje významnou roli při monitorování zdraví včel. Zjištěním známek onemocnění, stresu nebo environmentálních faktorů ovlivňujících včely mohou výzkumníci a včelaři včas zasáhnout a zmírnit potenciální hrozby pro zdraví úlů.

Tato práce je zaměřena na průzkum segmentačních technik, přístupů a metod. Implementuje jednu z existujících segmentační řešení (YOLO) a snaží se navrhnout vlastní přístup segmentaci, na základě techniky dynamického okna a klasifikátoru.

1.1 Členění práce

Kapitola *3* se zabývá rozborem problematiky úlohy segmentace, popisuje možné přístupy segmentace. Tato část práce čerpá informace z odborné literatury a poskytuje potřebné znalosti pro návrh programového vybavení.

Kapitola *4* stanoví funkční požadavky a navrhuje architekturu aplikace.

Kapitola *5* zohledňuje programové vybavení použité při tvorbě aplikace.

Kapitola *6* popisuje implementační kroky, pro každou část aplikace.

Kapitola *7* popisuje provedené experimenty segmentace a jejich výsledky.

Kapitola *8* poskytuje čtenáři diskuzi ohledně výsledků práce.

2 Cíl

Hlavním cílem práce je navrhnout a implementovat programové vybavení pro segmentaci včel na obrázcích plástů. Nejprve je nutné se seznámit s problematikou segmentace a s obecnou odbornou literaturou, zabývající se stejnou problematikou. Na základě této teoretické informace, navrhnout řešení pro segmentaci včel na obrázcích plástů. Programové vybavení umožní trénování neuronové sítě na vlastních datech, obdržení výsledků tréninků a provedení inferenci. Inferenci se rozumí proces používání trénované neuronové sítě k předpovědím nebo rozhodnutím na základě nových, neviditelných před tím dat. Během inference neuronová síť přijímá vstupní data, zpracovává je prostřednictvím svých vrstev a vytváří výstup nebo předpověď bez dalšího školení nebo aktualizace svých vah a parametrů.

Dílčí cíle:

1. Provést rešerši přístupů pro segmentaci obrazu a klasifikaci textur neuronovými sítěmi, případě v kombinaci s klasickými přístupy.
2. Vybrat vhodný přístup pro segmentaci včel na fotografiích plástů.
3. Využit nebo upravit existující implementaci, případně přímo implementovat, ve známých frameworcích.
4. Provést sérii experimentů a zhodnotit výsledky z hlediska přesnosti stanovení obrysu včely.

3 Teorie

3.1 Segmentace instancí

Segmentace instancí je formou segmentace obrazu, která se zabývá detekcí instancí objektů a vymezením jejich hranic.

Tato technika kombinuje dvě úlohy počítačového vidění: detekci objektů a sémantickou segmentaci, která poskytuje pixelovou identifikaci objektů a přiřazuje label kategorie každého pixelu v obrazu.

Úkolem segmentace je rozdělení obrazu na části, které odpovídají objektům v obraze zachyceným za účelem vyčlenění zájmových oblastí od zbytku obrazu. V závislosti na míře úspěchu je možno hovořit o částečné, nebo úplné segmentaci. [7]

V rámci rešerši přístupů bylo zohledněno několik prací, zabývajících se stejnou problematikou [1–6]. Uvedené články se dělí, podle přístupu, na ty, co upravují existující řešení pod specifickou úlohu [1, 2, 3] a ty, co obecně zohledňují přístupy i metody segmentace [4, 5, 6] a jsou základem pro návrh vlastního řešení.

Učení segmentačních neuronových sítí potřebuje anotovaný dataset, který se produkuje manuálně. Pro segmentaci včel, vyprodukujeme potřebné anotace a použijeme k naučení poslední verzi YOLO [8].

Odborné práce [4, 5, 6] uvádějí alternativní metody segmentace, mimo nějaké verze YOLO. Zohledňují přístup dynamického okna [4] s kombinaci s nějakým klasifikátorem [5, 6], který, při každé iterace dynamického okna, ohodnotí jeho obsah a poskytne **confidence score**, že v okně je objekt a pokud překročí předem definovaný práh, je považováno za oblast zájmu (**Region of Interest**) a souřadnice bounding boxu tohoto okna jsou zaznamenány jako odhadovaná poloha objektu.

Na zaklade teorie z těchto článku, se pokusíme, navrhnout a implementovat segmentační řešení, které by nezáviselo na anotacích a bylo by možné provést učení a segmentaci instancí objektů jenom na základě neanotovaných obrazů.

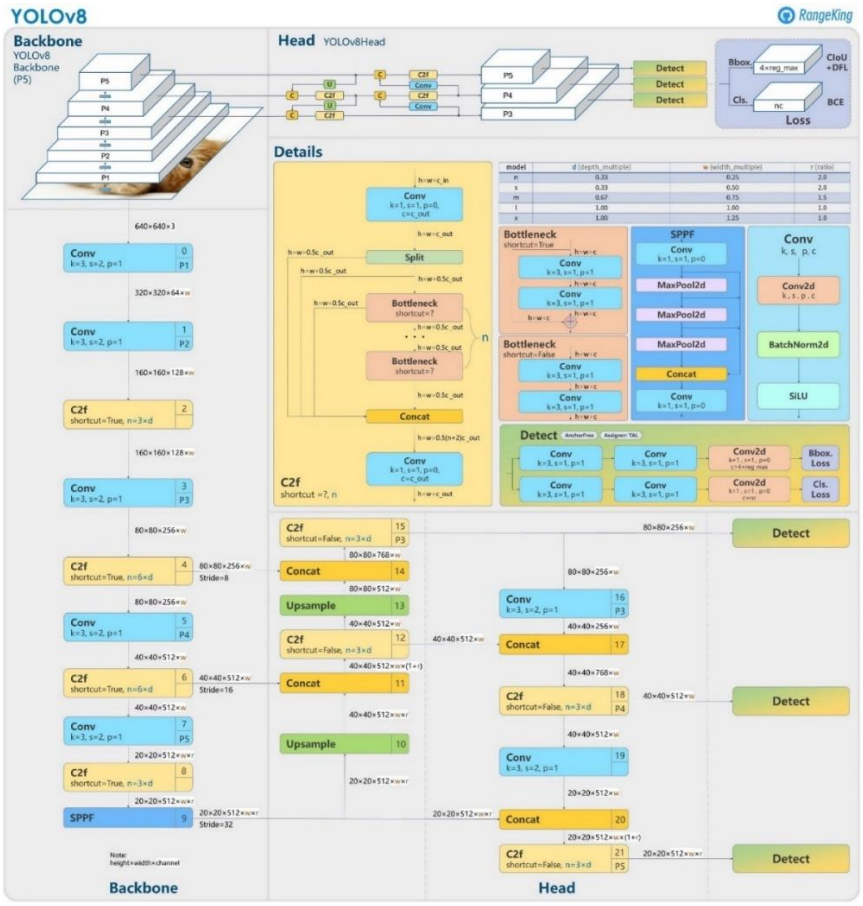
Jako architektury klasifikátorů se nabízí Hluboká Konvoluční Síť [5] a Hluboká Reziduální Síť [6].

3.2 YOLO

Detekce objektů je kombinací úloh klasifikace obrazu a lokalizace objektů, kde ohraničující rámeček (**bounding box**) doprovází předpokládanou třídu objektu. Algoritmy lokalizace objektů se obecně používají k lokalizaci přítomnosti objektu v obraze a reprezentují jeho umístění pomocí bounding boxu. [3]

YOLO framework používá konvoluční vrstvy k predikci bounding boxu a tříd pravděpodobnosti všech objektů zobrazených na obrázku. Protože algoritmus YOLO je jednorázový detektor, podívá se na obraz pouze jednou. YOLO vypočítá skóre spolehlivosti (**confidence score**) pro každý bounding box vynásobením pravděpodobnosti, že objekt bude v podkladové mřížce obsahovat průnik přes spojení (**Intersection over Union**) mezi základní pravdou a předpokládaným bounding boxem. Následně nemaximálního potlačení (**Non-Maximum Suppression** nebo **NMS**) odstraní překrývající se bounding boxy obklopující detekovaný objekt výběrem boxu s nejvyšší konfidenci. [3].

Tato architektura nabízí efektivní řešení pro úlohy segmentace, detekce a klasifikaci obrazu v reálném čase.

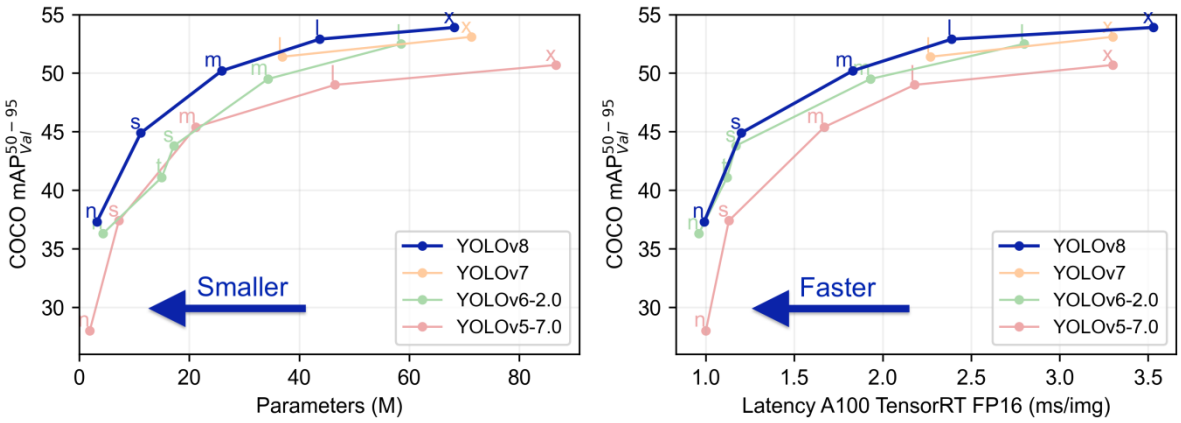


Obrázek 1 - YOLOv8 architektura [3]

Modely YOLO jsou trénovány na velkých datasetech ve stejném formátu. Poskytují vysoce přesné předpovědi ve třídách, ve kterých jsou naučené. Rychle se trénují a produkují výsledky vysoké přesnosti s menšími velikostmi modelů.

YOLOv8 je nejnovější iterací řady modelů YOLO (2023). Prošel několika změnami, jako *Anchor-free detection a Mosaic Augmentation*. Je významně výkonnější podle benchmarku Ultralytics [8].

Obraz 2 ukazuje porovnání střední průměrné přesnosti masky pro validační COCO dataset s konfidenci v rozmezí od 0.5 do 0.95 v závislosti na počtu parametrů sítě (její velikosti) vlevo a latenci v milisekundách na jeden obrázek vpravo.



Obrázek 2 - YOLO benchmarky Ultralytics [8]

Tato hodnota se vypočítá jako průměrná přesnost pro každou třídu.

$$mAP = \frac{1}{num\_classes} \times \left(AP_1 + AP_2 + \cdots + AP_{num\_classes}\right) \tag{1}$$

Samostatná průměrná přesnost vypočítá jako plocha pod křivkou **precision-recall**.

$$AP = \int_0^1 p(r)dr \tag{2}$$

Pro výpočet křivky **precision-recall** jsou předpovědi modelu seřazeny podle jejich konfidencí a následně je aplikován práh, který určí, jaké předpovědi jsou považovány za pozitivní nebo negativní.

Precision je poměr skutečných pozitivních předpovědí k celkovému počtu předpokládaných pozitivních odpovědí: skutečně pozitivních (TP) a falešně pozitivních (FP). Měří přesnost pozitivních předpovědí provedených modelem.

$$Precision = \frac{TP}{TP + FP} = \frac{TP}{all\_detections} \tag{3}$$

Recall je poměr skutečně pozitivních předpovědí k celkovému počtu skutečně pozitivních a falešně negativních (FN). Měří, kolik skutečných pozitiv může model identifikovat.

$$Recall = \frac{TP}{TP + FN} = \frac{TP}{all\_ground\_truths} \tag{4}$$

Skutečně Pozitivní Odpověď (TP) nastává, když model předpovídá pozitivní výsledek a skutečný výsledek je skutečně pozitivní.

Falešně Pozitivní Odpověď (FP) nastává, když model předpovídá pozitivní výsledek, ale skutečný výsledek je negativní.

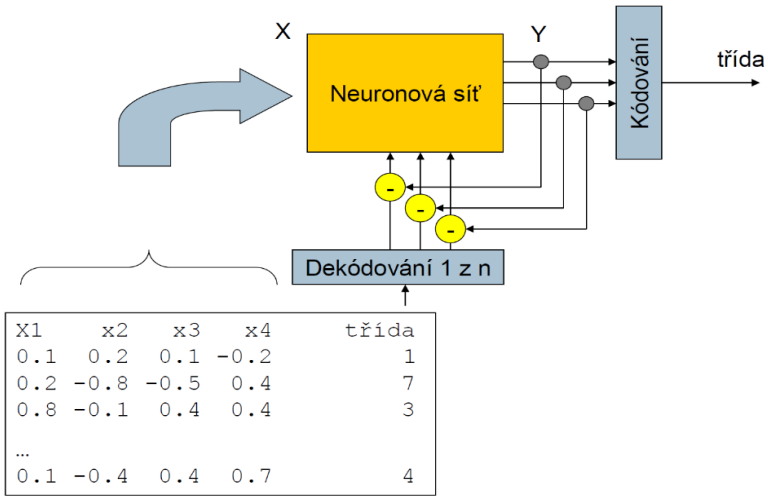
Falešně Negativní Odpověď (FN) nastává, když model předpovídá negativní výsledek, ale skutečný výsledek je pozitivní.

Střední průměrné přesnosti nad 0.4 se považuje za dobrou a znamená, že síť odhadla nad 40 % masek přítomných na validačním obraze. Přesnost 1 je skoro nedosažitelná, jak by síť měla vrátit výsledek do pixelu identický validačnímu obrazu.

3.3 Vlastní řešení

Vlastní segmentační řešení založíme na kombinací dvou odlišných klasifikátorů a dynamickém oknu.

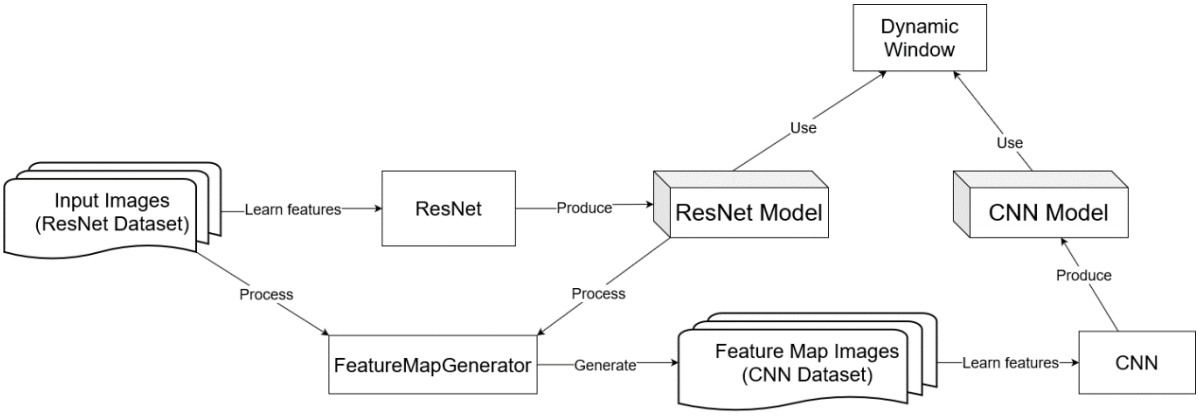
Cílem klasifikátorů je naučit se vzory a vztahy v datech, které umožňují predikovat třídu pro vstupní data.



Obrázek 3 – Princip fungování klasifikátoru [28]

Klasifikační neuronové sítě modelují složité, nelineární vztahy v datech. Přebírají nezpracovaná vstupní data, transformují je prostřednictvím řady vrstev a vytvářejí výstup, který představuje předikovaný label třídy.

Dva klasifikátory se budou učit v tandemu, kde jeden z klasifikátorů přispívá k naučení vlastnosti jednotlivé včely (tvar těla, velikost, pruhy atd.) a druhý se učí rozlišovat mapy vlastností vyprodukované předchozím klasifikátorem.



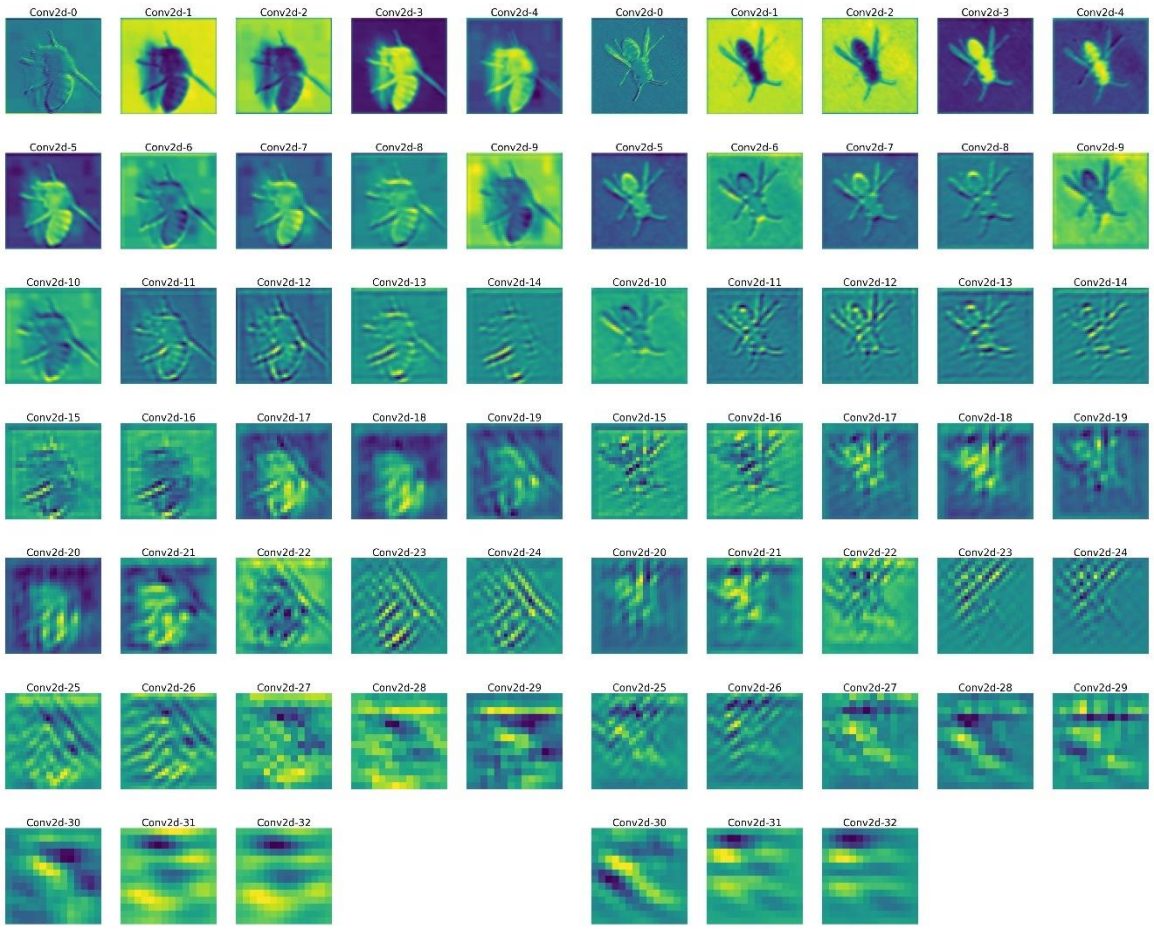
Obrázek 4 – Princip fungování vlastního řešení

K naučení vlastnosti využijeme ResNet [6], jako složitější a pokročilejší architekturu. Pro rozlišení využijeme CNN [5].

Po naučení ResNet, obdržíme mapu vlastnosti, z vrstvy před plně propojenou vrstvou (**FC Layer**), a pomocí FeatureMapGenerator provedeme inferenci mapy na každý z obrazů původního datasetu.

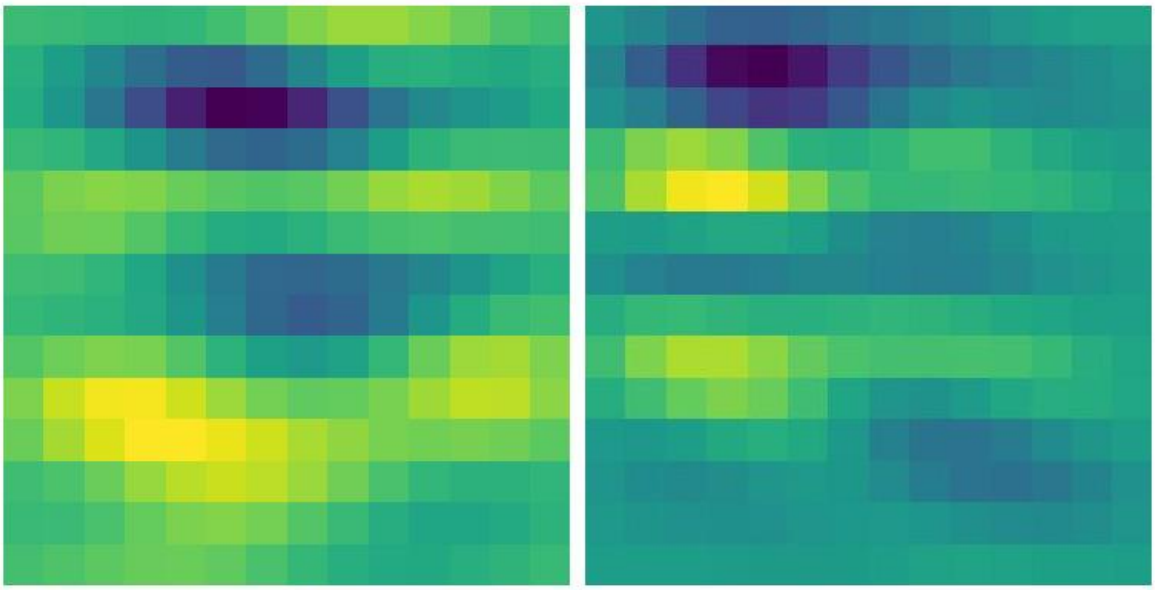
Výsledkem inference budou mapy vlastnosti jednotlivých včel a ne včel, ve formátu tensoru (tři rozměrové matici). Tensory se uloží jako samostatné obrazy.

Obrázek 5 znázorňuje inferenci mapy vlastností přes všechny konvoluční vrstvy ResNet.
V každé vrstvě, ResNet si zaznamenává nějaké vlastnosti a předává je do další vrstvy.



Obrázek 5 – Inference mapy vlastnosti ResNet. Vlevo: včela, vpravo: osa

Přes každou konvoluční vrstvu velikost výsledné mapy je jiná a postupně se zmenšuje. Mapa vlastností z poslední konvoluční vrstvy obsahuje v sobě veškery vlastnosti, kterým se naučil ResNet a má velikost 14×14. Světlejším jsou vyznačené nejdůležitější vlastnosti.



Obrázek 6 – Finální mapa vlastnosti inference. Vlevo: včela, vpravo: osa

Tyto obrazy jednotlivých map vlastností včel slouží vstupem do CNN, kde se konvertují na tesnory a mění svou velikost na 34×34. CNN se učí rozhodnout komu patří mapa vlastnosti včele nebo ne včele.

Vlastnosti produkované ResNet a konfidence od CNN jsou dále využitá dynamickém oknem při segmentaci.

Při každé iterace dynamického okna, jeho obsah, z formátu **nparray** se konvertuje do tensoru, který se předává na vstup FeatureMapGenerator s před naučeným ResNet. FeatureMapGenerator provádí inferenci mapy vlastnosti ResNet na předaný tensor. Výsledkem inference je nparray finální mapa vlastnosti, která se také konvertuje na tensor, upravuje velikost a předává na vstup před naučené CNN, která rozhoduje, do jaké kategorie, s jakou konfidenci, tensor patří.

Jestli konfidence CNN bude vyšší než definovaný práh spolehlivosti, tak souřadnice okna jsou zaznamenaná jako informace o bounding boxu objektu.

Mapa vlastností vygenerována FeatureMapGenerator pomocí mapy vlastností ResNet se využije pro vyděleni segmentační masky, na zaklade hodnot v tensoru.

3.3.1 Dynamické okno

Technika dynamického okna je rozšířením klasického posuvného okna. Dynamické okno, během své iterace po obrazu, může měnit svou velikost a zachycovat více nebo méně informaci v závislosti na velikosti obrazu a instancích objektů.

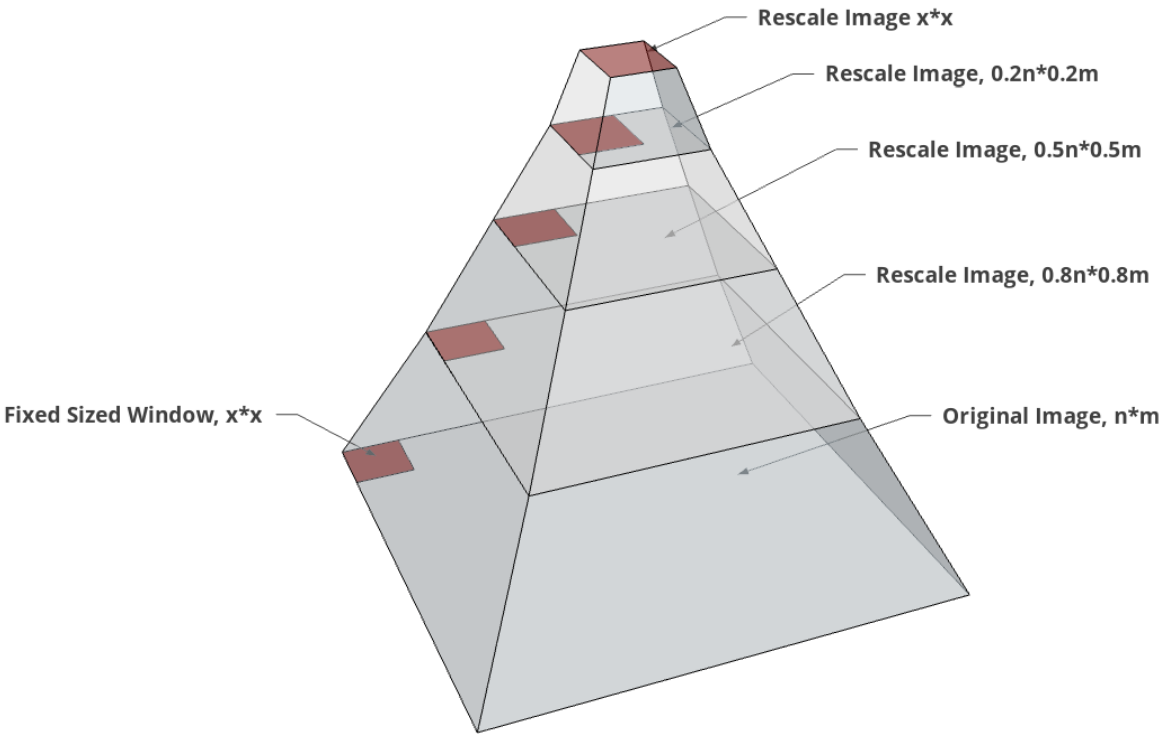
Velikosti, se kterými dynamické oknu má projít obraz se mohou zadávat výčtem, např.: 34, 100, 224 atd. V tomto případě pro každou velikost se vytvoří vlastní instance dynamického okna, která projde vstupním obrazem.

Velikosti se také mohou zadávat pomocí postupného přeškálování vstupního obrazu. V tomto případě máme konstantní velikost okna a několik obrazů různé velikosti. Čím menší je obraz, tím více informace zachytí okno a naopak. Iterace okna probíhají obraz o různých velikostech. Přeškálování probíhá do až do velikosti okna.

Metoda přeškálování obrazů má výhodu, že celkový počet iterací okna závisí na velikosti segmentačního obrazů a škalovacím koeficientu oproti výčtu, kde počet iterací je omezen předem definovanými velikostmi okna.

Přeškálování, efektivně tvoří ze vstupního obrazu pyramidu, skládající se s jeho přeškálovaných obrazů.

Obrázek 7 znázorňuje pyramidu obrazů spolu s oknem konstantní velikosti.



Obrázek 7 – Pyramida obrazů s oknem fixované velikosti

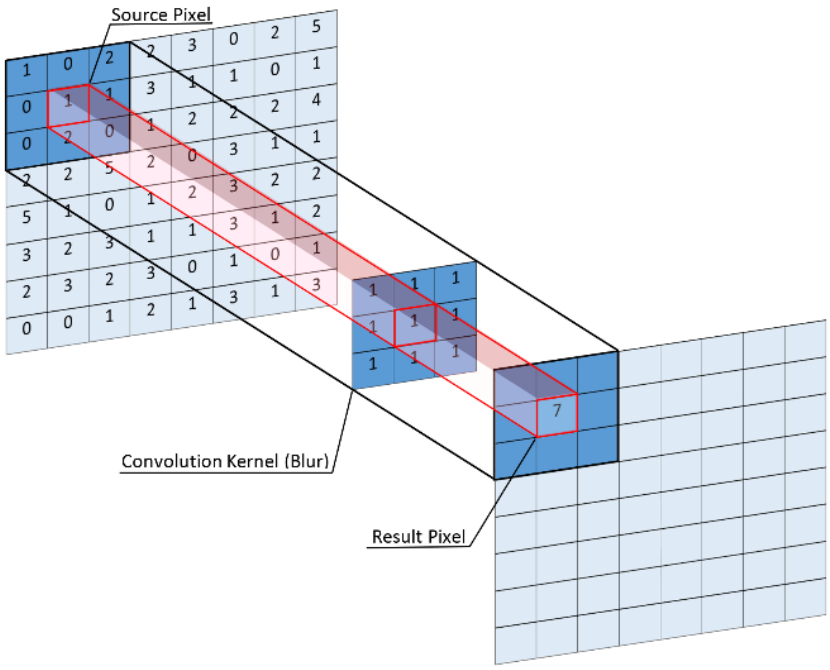
3.3.2 Hluboká konvoluční síť

Hluboké Konvoluční Sítě (**CNNs**) se dobře prokázaly v široké řadě úkolů vizuálního rozpoznávání včetně klasifikace textur. [5]

CNN je schopná se učit hierarchické prvky z vizuálních dat. Nižší vrstvy zachycují jednoduché prvky, zatímco vyšší vrstvy zachycují abstraktnější a složitější vzory.

Vstupní vrstva CNN přebírá nezpracovaná data, kde každý pixel v obrazu odpovídá uzlu v této vrstvě a tvoří mřížku neuronů, které představují obsah obrazu. Data ze vstupní vrstvy přecházejí do následujících konvolučních vrstev.

Konvoluční vrstvy zachycují vlastnosti nízké úrovně (**low-level features**), jako jsou hrany, rohy a textury. Konvoluce spočívá v aplikaci sady naučitelných filtrů (**kernels**) na vstupní obraz, vynásobením kernelu na vstupní matici a vytvořením nové mapy vlastností.



Obrázek 8 - Kernel [29]

Kernely se posouvají přes vstupní data, detekují vzory a prvky v různých částech vstupní sekvence. Počet kernelů v každé konvoluční vrstvě určuje počet vyprodukovaných map vlastností. Hlubší vrstvy obvykle zachycují složitější vlastnosti.

Konvoluční vrstvy jsou následované aktivačními funkcemi.

Aktivační funkce zavádějí nelinearitu do neuronové sítě. Toto umožňuje zachytit složitější vztahy mezi vlastnostmi objektů.

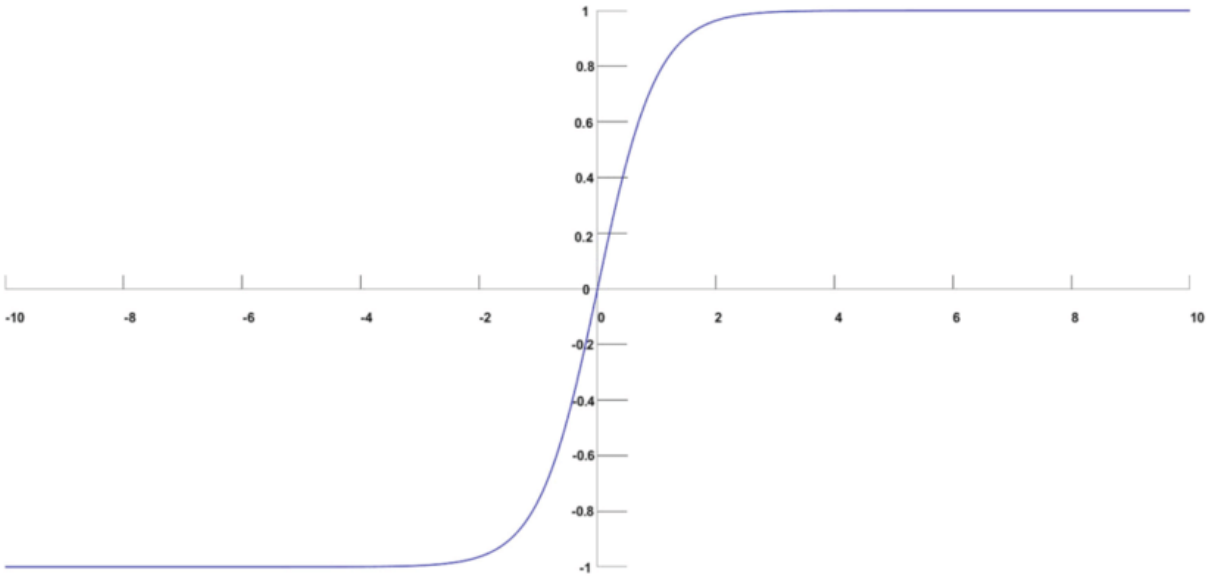
SoftMax, Sigmoida, RELU (The Rectified Linear Unit) jsou nejčastější používané aktivační funkce.

Softmax funkce je hladká křivka začínající od 0 a končící na 1, která odpovídá součtu pravděpodobností všech prvků vstupního vektoru. Softmax má vlastnost, že se zvýšením pravděpodobnosti jakéhokoliv prvku ve vektoru, hodnoty ostatních prvků se snižují, což umožňuje použití funkce pro klasifikaci s více třídami.

Softmax transformuje vektor vstupních prvků $z = [z_1, z_2, \dots z_n]$ na rozklad pravděpodobnosti Pro každý prvek vektoru z se počítá $p = [p_1, p_2, \dots p_n]$ se počítá pravděpodobnosti v rozmezí od 0 do 1, jako:

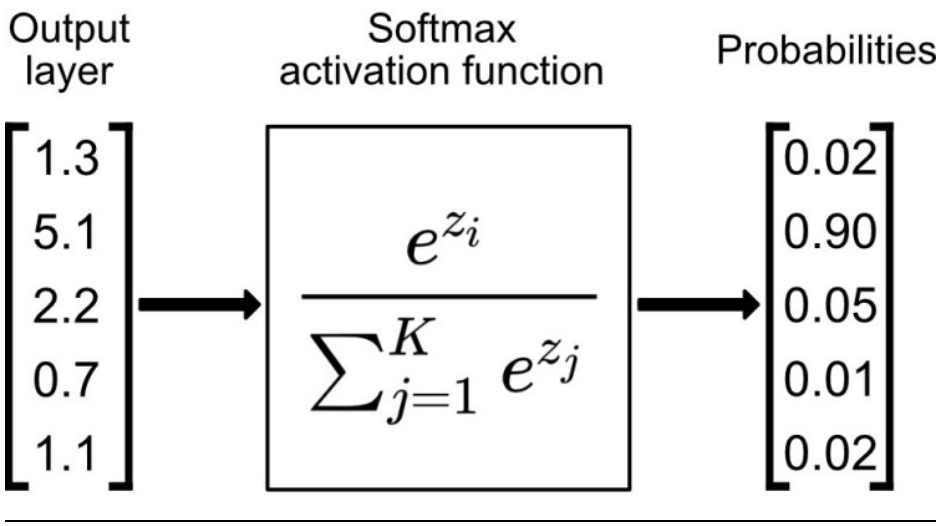
$$p_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{5}$$

kde e je základ přirozeného logaritmu (Eulerovo číslo), z_i je vstupní hodnota pro třídu i a jmenovatel $\sum_{j=1}^{n} e^{z_j}$ sečítá exponenciální hodnoty všech pravděpodobností pro všechny třídy.



Obrázek 9 - Softmax funkce [30]

Softmax umocňuje vstupní hodnoty a normalizuje je dělením součtem všech umocněných hodnot. Tato normalizace zajišťuje, že výsledné pravděpodobnosti se sčítají do 1.



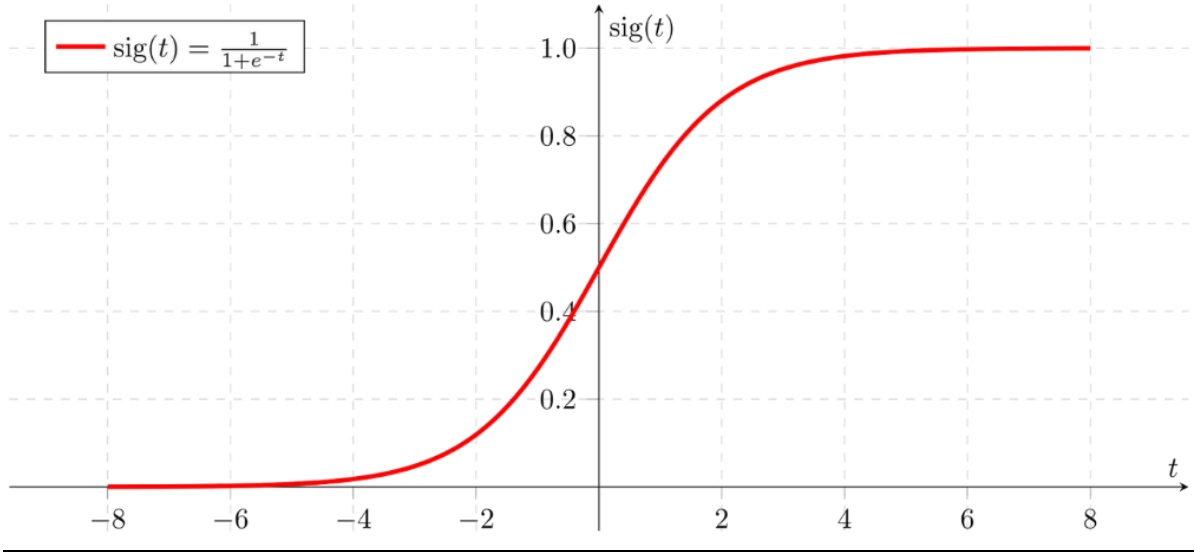
Obrázek 10 – Softmax funkce aplikace [31]

Sigmoida je aktivační funkce, ve tvaru S, která mapuje vstupní hodnoty do rozsahu mezi 0 a 1. Sigmoida se používá v úlohách binární klasifikace.

Sigmoida se definuje jako:

$$f(t) = \frac{1}{1 + e^{-t}} \tag{6}$$

kde e je základ přirozeného logaritmu (Eulerovo číslo) a t je vstupní hodnota



Obrázek 11 – Sigmoida funkce [32]

Nevýhodou sigmoidy je problém mizejícího gradientu při použití v CNN, což znesnadňuje proces učení. Když gradient se stává příliš malým, váhy sítě se také aktualizují pomalu a učení se zpomaluje.

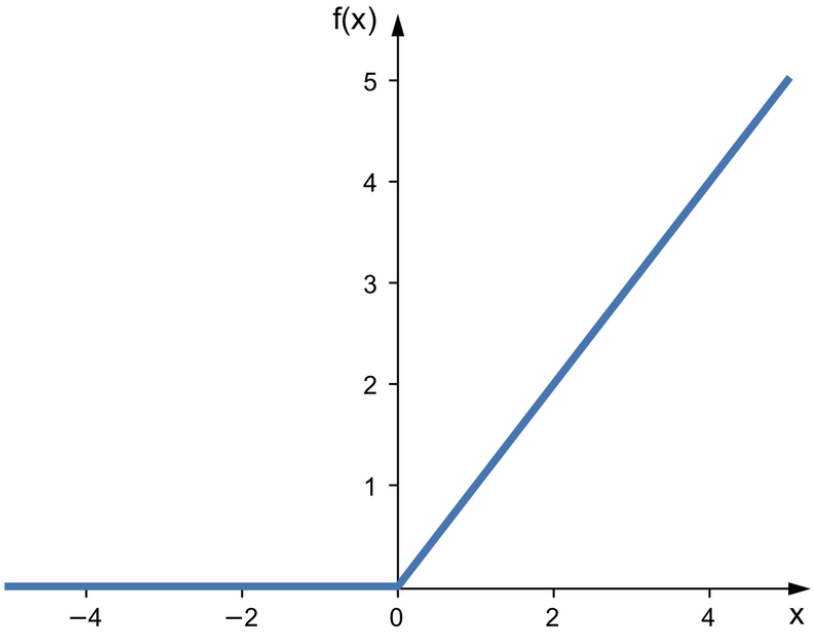
Toto omezení vedlo k použití dalších aktivačních funkcí, jako je ReLU (Rectified Linear Unit).

ReLU nahrazuje záporné hodnoty nulami, což pomáhá síti zachytit složité vztahy v datech, zaměřit se na nejrelevantnější vlastností a snížit riziko nadměrného vybavení (**overfitting**).

ReLU se definuje jako:

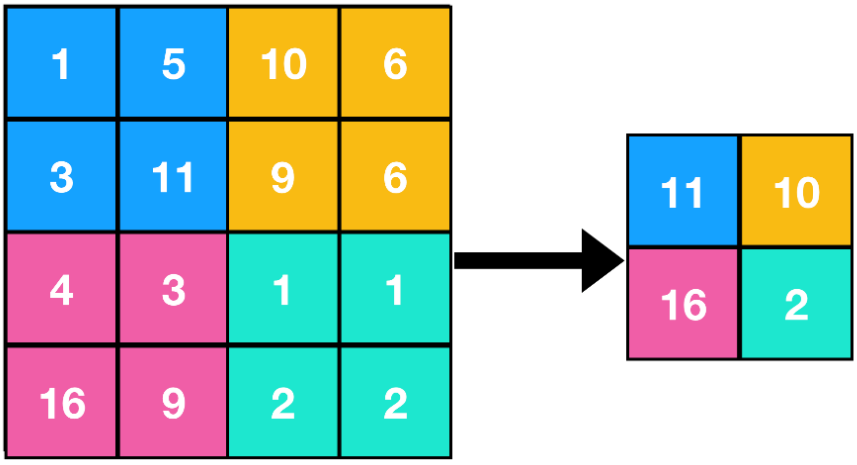
$$f(x) = max\,(0, x) \tag{7}$$

Kde x je vstupní hodnota a max je funkce, která se používá k porovnání vstupu x s 0. Pokud je vstup x větší nebo roven 0, výstup $f(x)$ se rovná x. Pokud je vstup x menší než nula, výstup $f(x)$ je nastaven na 0.



Obrázek 12 – ReLU funkce [33]

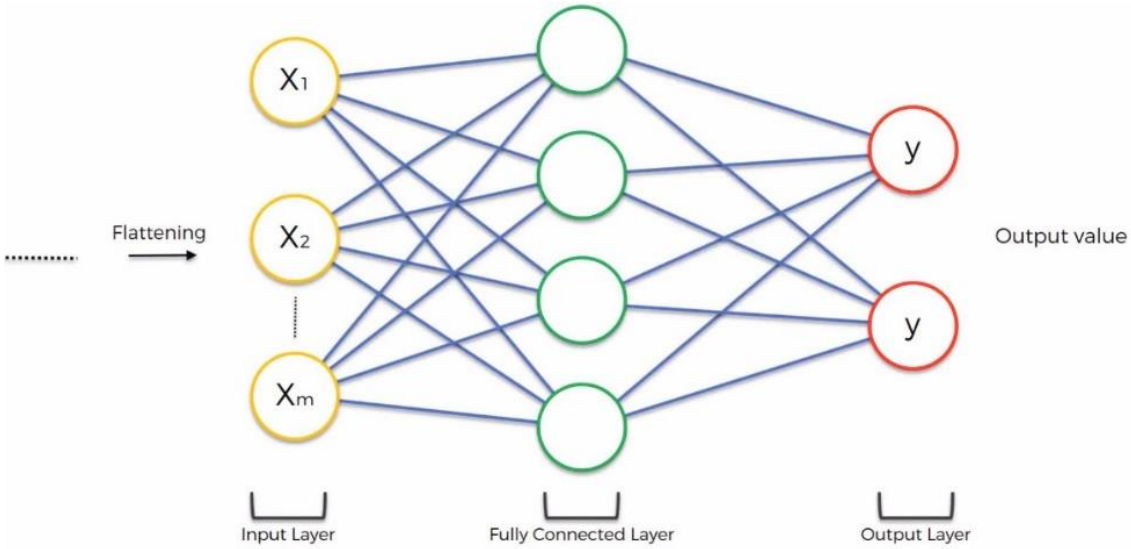
Pooling vrstvy snižují prostorové rozměry (**spatial dimensions**) feature map vyprodukovaných konvolučními vrstvami. Tímto přispívají ke koncentraci sítě na důležitějších vlastnostech objektu. Běžné pooling operace jsou max-pooling (výběr maximální hodnoty) a average-pooling (výpočet průměrné hodnoty).



Obrázek 13 – Max Pooling příklad [34]

Flatten vrstva přetváří feature mapy na jednorozměrný vektor před přechodem z konvolučních vrstev na plně propojené (**Fully Connected**) vrstvy. Plně propojené vrstvy vyžadují plochý vstup.

Fully Connected vrstva slouží jako neuronová síť, spojující všechny neurony v jedné vrstvě se všemi neurony v další vrstvě. Tyto vrstvy mapují vlastnosti vysoké úrovni (**high-level features**) extrahované z předchozích vrstev do výstupní vrstvy.

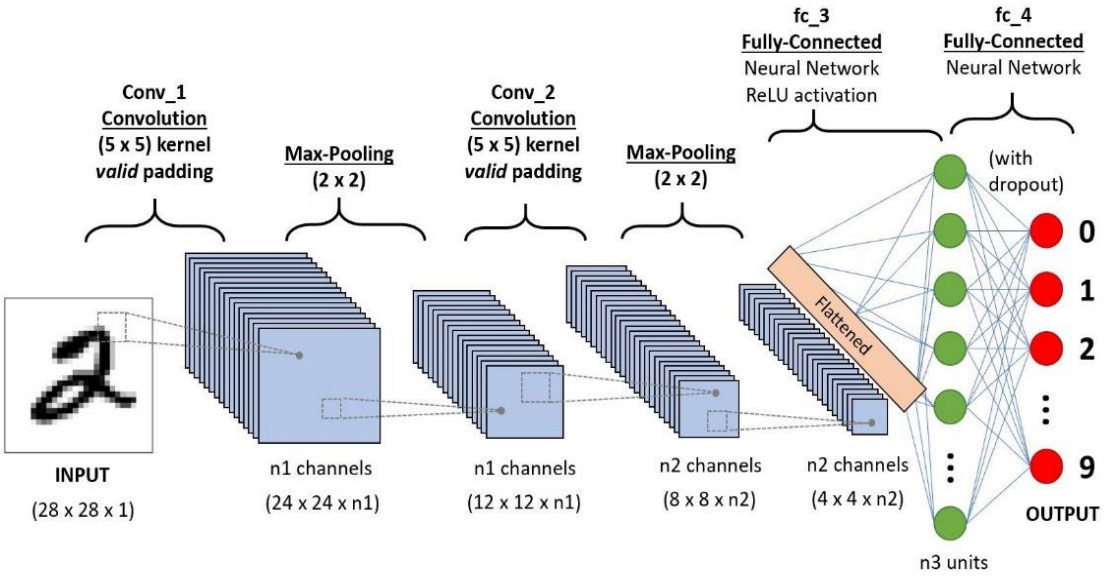


Obrázek 14 – FC vrstva [35]

Dropout vrstva náhodně deaktivuje část neuronů během učení, čímž vynucuje síť spoléhat na větší počet vlastností a přispívá ke schopnosti generalizace sítě.

Výstupní vrstva poskytuje konečné předpovědi nebo výsledky.

Obrázek 15 znázorňuje klasickou CNN architektu, skládající z popsaných prvků.



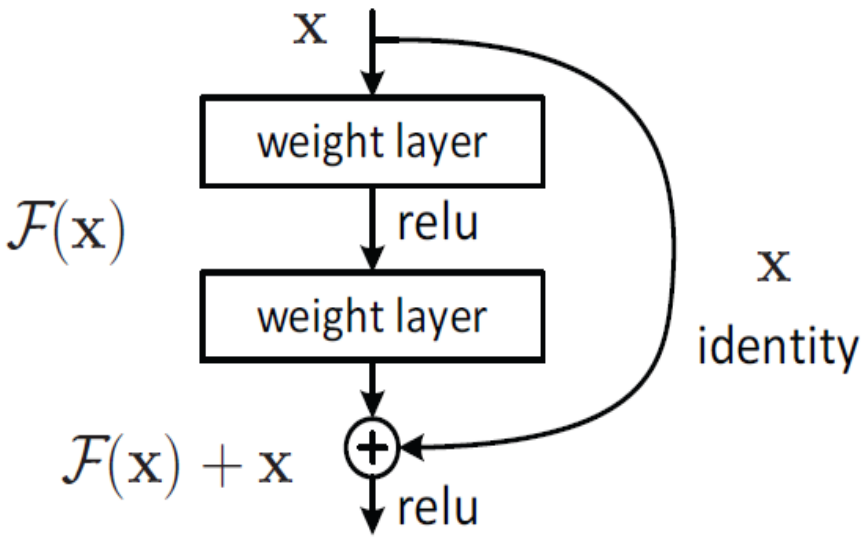
Obrázek 15 – CNN architektura [36]

3.3.3 Hluboká reziduální síť

ResNet je typ architektury CNN, který zavádí nový architektonický prvek zvaný **reziduální propojení** nebo **skip connection**. S rostoucí hloubkou CNN se potýkají s problémy, jako je mizející gradienty [6].

ResNet zahrnuje reziduální propojení, mezi konvolučními vrstvy (**weight layer**), která umožňují tok informací přímo z jedné vrstvy do druhé. To pomáhá zmírnit problém mizejícího gradientu.

Reziduální propojení poskytují přechody gradientu přímo z výstupních vrstev do vstupních vrstev. Toto umožňuje síti se naučit požadované transformace a mapování identity [6].



Obrázek 16 – Reziduální propojení [6]

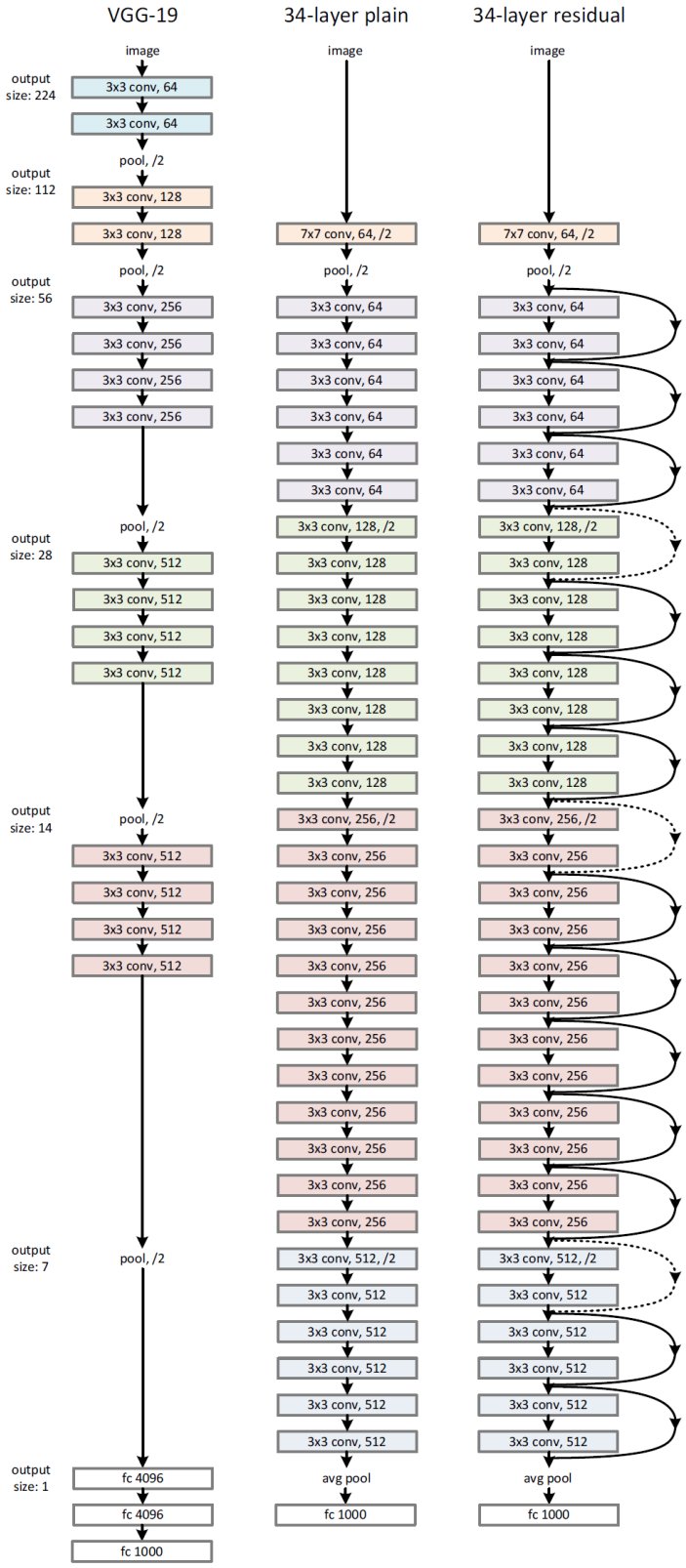
Reziduální propojení se skládá ze dvou cest: identity a transformace.

Cesta identity předává vstup přímo výstupu. Toto umožňuje síti naučit se funkci identity.

Cesta transformace obsahuje jednu nebo více konvolučních vrstev následovaných normalizačními a aktivačními funkcemi. Je zodpovědná za učení transformace, kterou je třeba aplikovat na vstup.

ResNet může mít stovky nebo tisíce vrstev, tím je vhodná pro úkoly vyžadující abstrakce na vysoké úrovni.

Obrázek 17 znázorňuje architekturu 34vrstevní ResNet v porovnání s 34vrstevní „plochou“ architekturu bez reziduálních propojení a VGG-19 (typ CNN).



Obrázek 17 - Porovnání ResNet a CNN architektur [6]

Vlevo: model VGG-19 (19,6 miliard FLOP) jako reference. Uprostřed: jednoduchá síť s 34 vrstvami parametrů (3,6 miliardy FLOP). Vpravo: reziduální síť s 34 vrstvami parametrů (3,6 miliardy FLOP).

3.4 Dataset

Každé z řešení potřebuje přípravu vlastního datasetu. YOLO vyžaduje vytvoření konfiguraci a anotace pro učební data. Pro vlastní řešení je třeba sestavit dataset, ze kterého se klasifikátor bude schopen naučit včelí vlastnosti.

3.4.1 YOLO

Konfigurace YOLO definuje strukturu datasetu (kořenový adresář, relativní cesty k adresářům trénovacích/validačních obrázků nebo souborů *.txt obsahujících cesty k obrázkům a anotacím tříd).

```
path: data
train: train/images
val: val/images

names:
  0: bee
```

Zdrojový kód 1 – YOLO dataset konfigurace

Labely jsou exportovány do formátu YOLO s jedním souborem *.txt na obrázek. Pokud v obrázku nejsou žádné objekty, není vyžadován žádný soubor *.txt.

```
data
|
|__train
|   |
|   |__images
|   |
|   |__labels
|__val
|   |
|   |__images
|   |
|   |__labels
|__coco128-seg.yaml
```

Zdrojový kód 2 – YOLO dataset struktura

YOLO je textový formát, kde každý textový soubor odpovídá obrázku v datasetu a každý řádek v textovém souboru představuje anotovanou instanci objektu ve formátu:

```
(class_id, x_center, y_center, width, height)
```

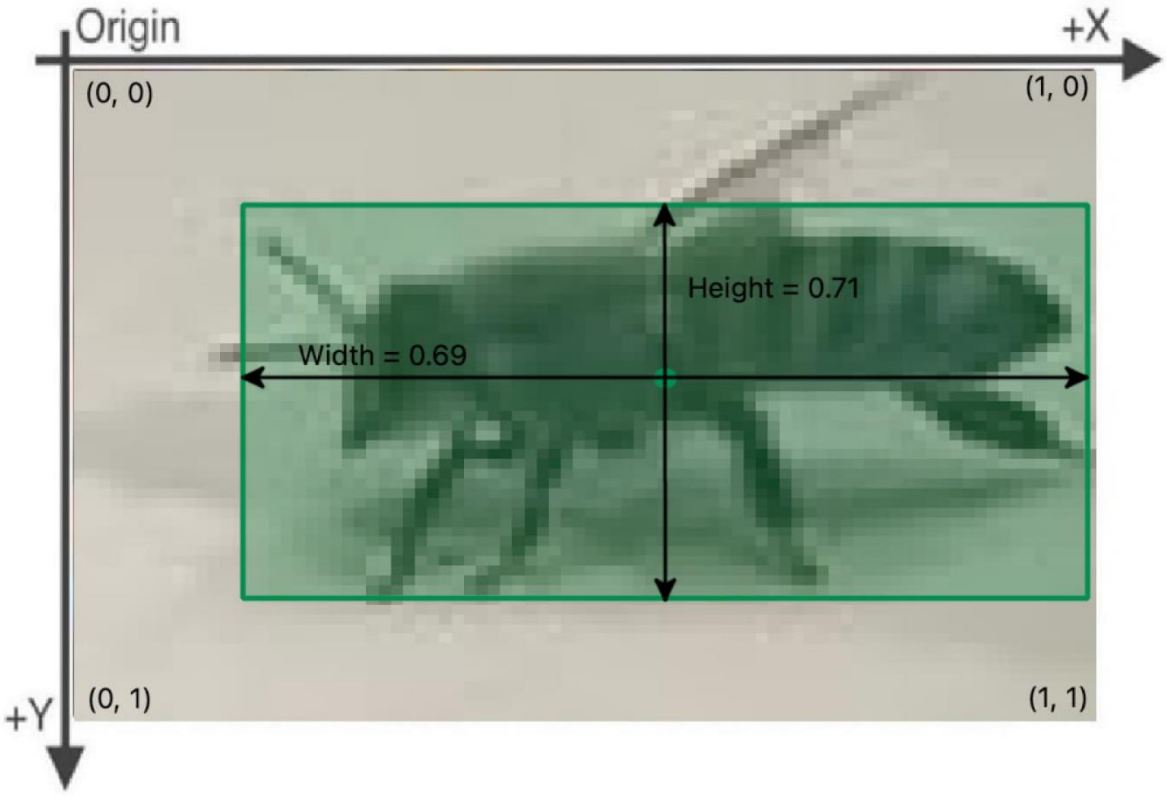
Zdrojový kód 3 – YOLO anotace struktura

```
0, 0.481719, 0.634828, 0.690625, 0.713278
```

Zdrojový kód 4 – YOLO anotace příklad

YOLO používá normalizované souřadnice, díky tomuto je ten formát nezávislý na velikosti obrazu.

Obr. 18 vizualizace YOLO anotace. Soubor s labely odpovídající uvedenému obrázku obsahuje 1 včelu (třída 0).



Obrázek 18 – YOLO anotace vizualizace

Anotace se produkují nástroji na labelování obrazu, jako CVAT [9] nebo VoTT [10].

Textové formáty mají obtíž reprezentace složitých anotačních struktur. Z těchto důvodů se široce používá alternativní formát COCO.

COCO (Common Objects in Context) je **JSON-based** anotační formát, který poskytuje informace o instancích objektů v obrázku, jejich bounding boxy, kategoriích a segmentačních maskách. Anotace zahrnuje:

- **images**: Informace o obrazu:
    - ID obrazu
    - Šířka obrazu
    - Výška obrazu
    - Jméno souboru

- **annotations**: Informace o instancích tříd objektů:
    - ID instanci (**id**)
    - ID kategorie (**category_id**)
    - Souřadnice bounding boxu (**bbox**)
    - Segmentační maska (**segmentation**)
- **categories**: Definice kategorie:
    - ID kategorie (**id**)
    - Jméno kategorie

```
{
  "images": [
    {"id": 1, "width": 608, "height": 608, "file_name": "19.jpg"},
    ...,
    {"id": 15, "width": 416, "height": 416, "file_name": "zelizone.jpg"}
  ],
  "annotations": [
    {
      "id": 1, "image_id": 1, "category_id": 1,
      "segmentation": [[46.54, 275.76, ..., 277.25]],
      "area": 252.0,
      "bbox": [32.7, 275.76, 16.14, 23.33],
      "iscrowd": 0,
      "attributes": {"occluded": false}
    },
    ...,
    {
      "id": 1302, "image_id": 15, "category_id": 1,
      "segmentation": [[386.4, 130.05, ..., 130.05]],
      "area": 1197.0,
      "bbox": [371.99, 51.15, 44.01, 78.9],
      "iscrowd": 0,
      "attributes": {"occluded": false}
    }
  ],
  "categories": [{"id": 1, "name": "bee", "supercategory": ""}]
}
```

Zdrojový kód 5 – COCO anotace příklad

Souřadnice bounding boxů mohou být zadané v normalizovaných nebo absolutních hodnotách. Při tvorbě anotaci zadáme absolutní hodnoty.

COCO je konvertováno do YOLO formátu s normalizaci pomocí scriptu A v příloze.

3.4.2 Vlastní řešení

Pro naučení klasifikátoru vytvoříme vlastní dataset kombinací včelích a nevčelích obrazů (pozitivních a negativních instancí).

Negativní instance pomáhají klasifikátoru odlišit unikátní vlastností pozitivních instancí a vypočíst metriky, jako je přesnost (**accuracy**), preciznost (**precision**), zapamatovatelnost (**recall**) atd. pro hodnocení kvality a efektivity učení.

"Bee or wasp?" [11], „Honey Bee pollen" [12], „The BeeImage Dataset" [13] budou základem pro sestavení vlastního datasetu.

Z každého datasetu vybereme náhodné obrazy jednotlivých včel. Kombinování obrazu různých formátů s různými druhy včel zajistí schopnost klasifikátoru se naučit abstraktní vlastnosti.

Obrázek 19 – Včely z datasetů:
"Bee or wasp?", "Honey Bee pollen", „The BeeImage Dataset"

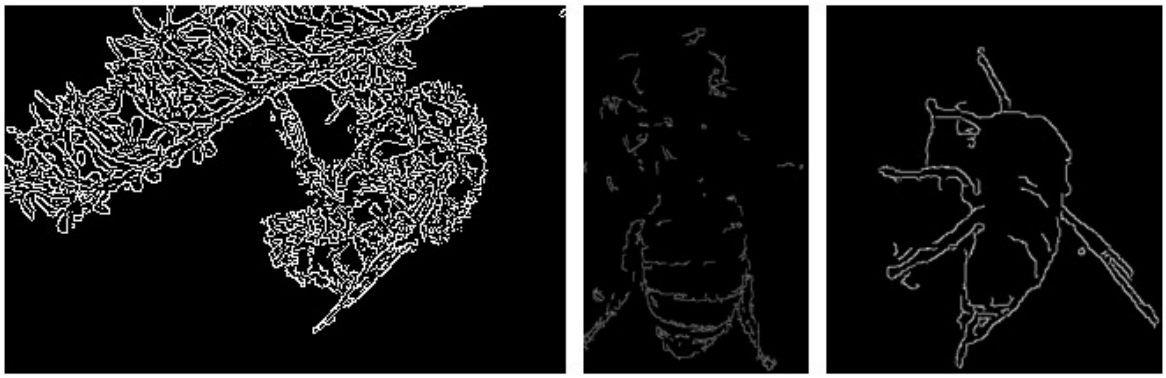
Bee or wasp [11] poskytuje fotografie os, sršňů a jiných hmyzů, z nichž si sestavíme dataset negativních obrazů.

Obrázek 20 – Negativní instance: Osa, Sršeň, Moucha z „Bee or wasp" datasetu

Originální obrazy zkonvertujeme na jejich **edge** formu, tzn. opustíme barevné informace o všem obrazu a zanecháváme jenom informace o hranách objektů.

Edge obraz je binární obraz, kde každý pixel je klasifikován jako hrana (**edge**) – bílým nebo ne hrana (**non-edge**) – černým.

Redukcí barev zmenšíme počet vlastností, potřebné pro naučení ResNet klasifikátoru. Bez barevných informací se klasifikátor zaměří na samostatný obrys včely.



Obrázek 21 – Edge obraz 18

Obrázek 22 – Edge obraz 19

Učení provedeme na obou typech datasetů.

Struktura datasetu bude se skládat z **root** složky, 2 složek **train** a **val**, kde každá z těchto 2 složek bude mít složky s obrazy, které se mají jmenovat podle třídy obrazů.

```
data
|
|__train
|   |
|   |__bee
|   |
|   |__not_bee
|__val
   |
   |__bee
   |
   |__not_bee
```

Zdrojový kód 6 – ResNet dataset struktura

4 Návrh

Programové vybavení navrhneme jako web aplikaci pro trénování a inferenci neuronové sítě na vlastních datech. Implementujeme oba segmentační řešení jako separátní moduly a zpřístupněme jejich funkcionalitu pomocí Web API.

Cílem aplikace je zprovoznit přistup k funkcionalitě neuronových síti, aniž by si uživatel měl lokálně instalovat síť a provádět její učení na svém zařízení, které, nejspíše, není zaměřeno na práci s neuronovými síti.

Aplikace se umístí na výkonní GPU server, na který uživatel zašle vlastní data pro úkol a následně obdrží výsledky.

4.1 Funkční požadavky

Aplikace umožní:

1. Trénování vybrané neuronové sítě
2. Stažení a mazaní výsledků trénovaní (z databáze)
3. Inference oproti vlastním datům

Uživatel zadá parametry (typ sítě, velikost, počet epoch, velikost batche, dataset) a spustí požadovanou akci.

Proces učení může trvat několik hodin nebo několik dní a musí probíhat na serveru ve vlastním vlákně, aby aplikace nebyla blokovaná po dobu učení sítě. Uživatel tím bude moc pouštět více učení najednou.

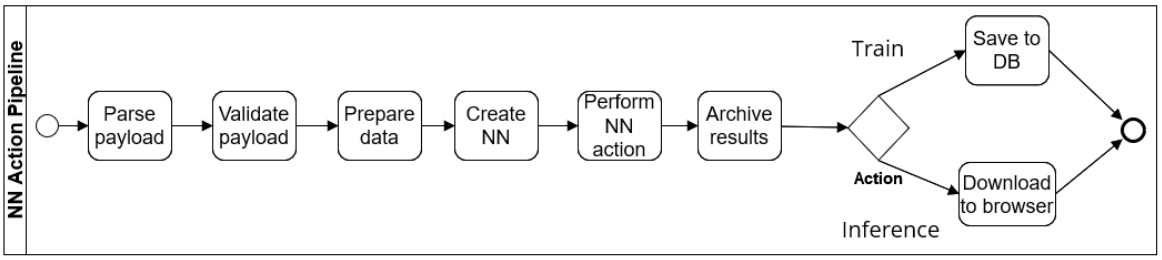
Informace o aktuálních běžících sítích se zobrazí na dashboardu. Tato informace se aktualizuje při každém startu a ukončení tréninků.

Při obdržení požadavku na akci (trénink nebo inference) neuronové sítě, proběhne:

1. Parsing bytového obsahu payloadu
2. Validace parsovaných hodnot
3. Příprava dat (uložení, dearchivace)
4. Vytvoření modelu pro úkol
5. Vykonání úkolu
6. Archivace a export výsledků

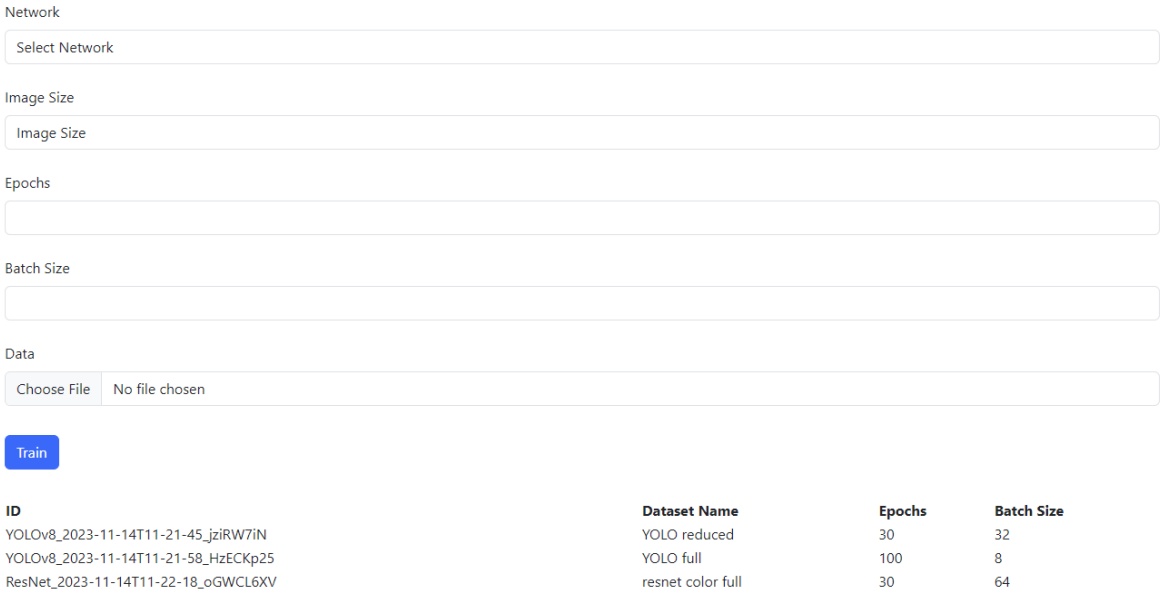
Pro požadavek na učení, zapíšeme výsledky do databáze, na inferenci je stáhneme do uživatelského prohlížeče.

Obrázek 23 zobrazuje **Business Process Model and Notation (BPNM)** diagram pro konání akce neuronové sítě na serveru.



Obrázek 23 – BPMN diagram akce neuronové sítě

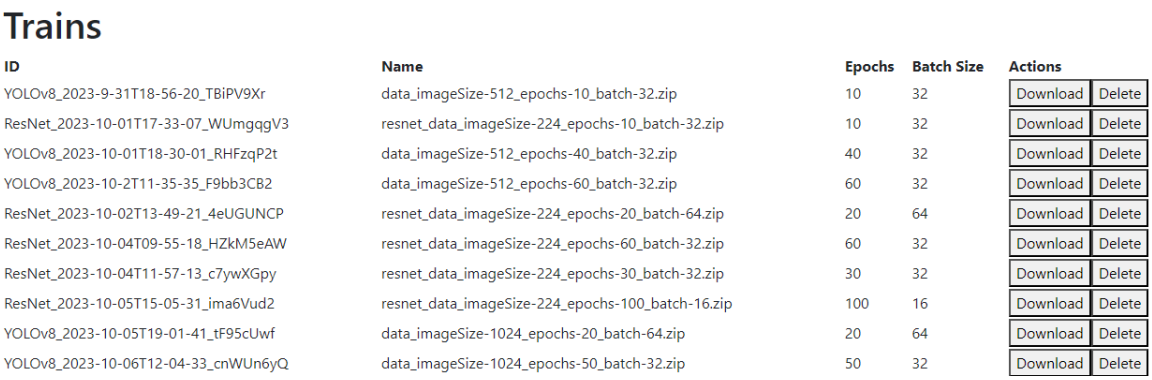
Obrázek 24 znázorňuje stránku pro zadaní tréninku neuronové sítě.



Obrázek 24 – Train page s dashboardem

Výsledky tréninku (váhy a grafiky průběhu učení) se uloží do databáze. Uživatel po stranách vylistuje všechny výsledky a bude moct je stáhnout nebo smazat.

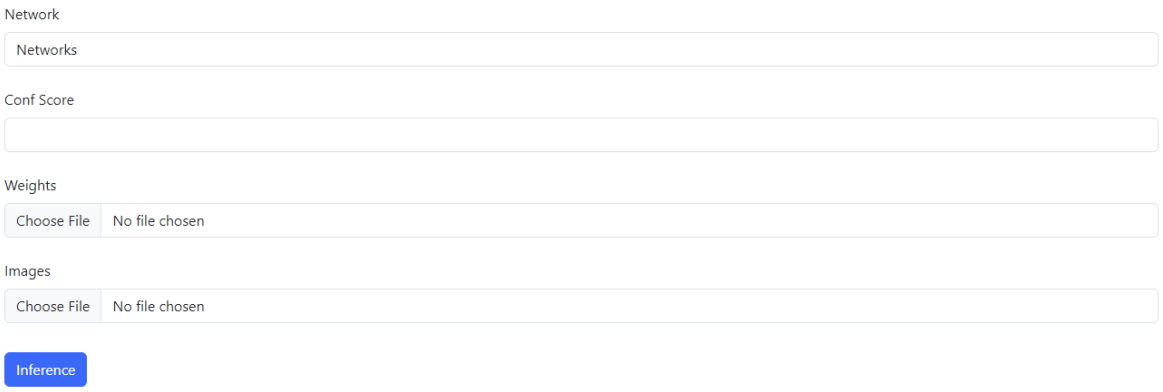
Obrázek 25 znázorňuje stránku pro zobrazení výsledků učení neuronové sítě.



Obrázek 25 – List page

Stažené váhy se využívají při inferenci. Uživatel zadá konfidenci, nahraje soubor s váhami, a složku s obrazy. Výsledné obrazy se stáhnou přímo do prohlížeče bez ukládání do databáze.

Obrázek 26 znázorňuje stránku pro inferenci.



Obrázek 26 – Inference page

4.2 Architektura

Aplikaci založíme na MVC architektuře, však upravíme ji podle našich potřeb.

V MVC architektuře Web 2.0, klient je implementován jako samostatná aplikace pomocí nějakého JavaScript frameworku nebo knihovny a je oddělen od serveru, který může běžet na vlastním serveru. Typicky, komunikace mezi klientem a serverem probíhá přes **HTTP požadavky**, které jsou inicializované klientem.

Tato skutečnost klade velké omezení na interaktivitu naší aplikace, jelikož pro zobrazení **real-time** informace o trénincích běžících na serveru, potřebujeme uvědomovat klienta, kdy trénink začne a skončí, abychom aktualizovali informací na dashboardu.

Předejít tomuto omezení lze pomocí několika technik:

1. **HTTP Long Polling** – je technika, kdy klient odešle HTTP požadavek na server a server ponechá požadavek otevřený, dokud nebudou k dispozici nová data. Jakmile jsou data dostupná nebo je splněna určitá podmínka, server odpoví a klient okamžitě odešle další požadavek. To vytváří iluzi požadavku iniciovaného serverem.
2. **Server-Sent Events** (SSE) – je standard pro vytvoření jednosměrného kanálu ze serveru ke klientovi přes jediné připojení HTTP. Klient naváže připojení SSE odesláním http požadavku na server a server pak může klientovi posílat aktualizace, kdykoli jsou k dispozici nová data.
3. **AJAX Polling** – zahrnuje zasílání pravidelných HTTP požadavků klientem na server pro kontrolu aktuálního stavu.
4. **WebSockets** – je komunikační protokol, který poskytuje plně duplexní, obousměrné komunikační kanály přes jediné spojení TCP (Transmission Control Protocol). Ve chvílí, kdy klient odešle požadavek a server na něj odpoví, WebSocket naváže nepřetržitou komunikaci mezi klientem a serverem v reálném čase.
5. **Message Queues** –Fronty zpráv (např. RabbitMQ, Apache Kafka), pomocí nichž, server může publikovat zprávy do fronty a klient se může přihlásit k odběru aktualizací.

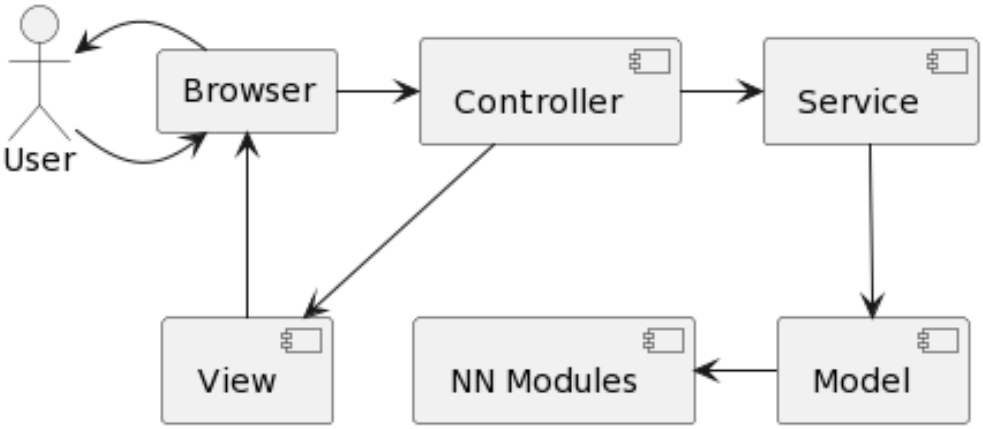
Žádná z **polling** technik nezajišťuje požadovanou real-time komunikaci.

Standart **SSE** je navržen pro jednosměrnou komunikaci, což dělá ho nevhodným pro implementaci v naši aplikaci.

Fronta zpráv je uvažováno za nadbytečné řešení pro dashboard update.

WebSockets, díky své schopnosti obousměrné real-time komunikace mezi klientem a serverem, jsou uznána jako nejvhodnější technologií pro implementaci stanovených funkčních požadavků.

Pro interakci s funkcionalitou obou modulů neuronových sítě, navrhneme další **Service** vrstvu, které také delegujeme práci s **Modelem**. V tomto případě, **Controller** bude záviset na **Service**, **Service** záviset na **Modelu** a **Model** na **NN Modulech**



Obrázek 27 – Architektura aplikace

5 Využité programové vybavení

Python (3. 10. 0) – programovací jazyk pro serverovou části a neuronové sítě. [15]

- Flask (2.3.3) – Python framework pro vývoj na stráně serveru. [16]
- Flask-Cors (4.0.0) – modul pro zpracování Cross-Origin Resource Sharing (CORS), které umožňuje komunikaci mezi webovými aplikacemi umístěných na různých doménách (klient a server). [17]
- Flask-PyMongo (2.3.0) – modul pro práci s databázemi MongoDB. [18]
- Flask-SocketIO (5.3.6) – modul pro WebSocket komunikaci na stráně serveru. [19]
- Eventlet (0.33.3) – knihovna pro souběžné síťové programování pomocí korutin. [20]
- PyTorch (2.0.1) – framework pro implementaci obou modulů neuronových sítí. [21]

JavaScript – programovací jazyk pro klientskou část aplikace. [22]

- Node.js (18.17.1) – JavaScript runtime pro spouštění JS kódu na stráně serveru, přístup k Node Packet Manager (NPM). [23]
- React.js (18.2.0) – knihovna pro vytváření uživatelských rozhraní s architekturou založenou na komponentách [24]
- Socket.IO-Client (4.7.2) – knihovna pro implementaci WebSocket komunikace na straně klienta. [25]

MongoDB – NoSQL dokumentová databáze pro ukládání výstupů neuronových sítí. [26]

CUDA Toolkit (11.8) – nástroj pro vytvoření environmentu pro **GPU-accelerated** aplikace. Využívá se pro spouštění neuronových sítě na GPU. [27]

6 Implementace

6.1 YOLO

Pomocí softwarového nástroje CVAT [9] vytvoříme anotaci ve formátu COCO a zkonvertujeme ji před učením do YOLO [8].

Celková velikost vytvořeného datasetu je 50 obrazů pro učení a 15 pro validaci. Celkový počet instanci včel pro učení je 5269, pro validaci – 1302.

Obrázek 28 – Včely na plastů, vizualizace anotace

Pomocí skriptu B v příloze, rozdělíme tyto 65 obrazů na obrázky jednotlivých včel, podle vyprodukovaných anotací, čímž zvýšíme objem trénovacích dat.

Obrázek 29 – Jednotlivé včely, vizualizace anotace

Provedeme instalaci YOLO jako `pip install ultralytics`

Inicializujeme model a spustíme trénink s parametry, kde **data** je cesta k YOLO konfigurace z 3.4.1. Písmeno za **yolov8** určuje velikost sítě: **n**ano, **s**mall, **m**edium, **l**arge a e**x**tra-large.

```
model = YOLO('yolov8l.yaml')
model.train(
    data=data,
    epochs=100,
    batch=4,
    imgsz=1024,
    project=result_save_dir,
    name=folder_name
)
```

Zdrojový kód 7 – Spouštění YOLO tréninku

6.2 Vlastní řešení

Na základě oficiální dokumentace PyTorch [21], navrhneme 2 modely neuronových sítě. ResNet34 pro naučení vlastnosti a CNN pro klasifikaci vlastností.

```
import torch.nn as nn


class ResNet34(nn.Module):
    def __init__(self, block, layers, num_classes):
        super(ResNet34, self).__init__()
        self.inplanes = 64
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3),
            nn.BatchNorm2d(64),
            nn.ReLU())
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.layer0 = self._make_layer(block, 64, layers[0], stride=1)
        self.layer1 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer2 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer3 = self._make_layer(block, 512, layers[3], stride=2)
        self.avgpool = nn.AvgPool2d(7, stride=1)
        self.fc = nn.Linear(512, num_classes)

    def forward(self, x):
        ...

    def _make_layer(self, block, planes, blocks, stride=1):
        ...
```

Zdrojový kód 8 – ResNet Model

Úroveň ResNet (34) je určen součtem konvolučních a FC vrstev v modelu. Layers je pole, které definuje počet reziduálních bloků, v každé skupině konvolučních vrstev, které se vytváří v metodě _make_layer.

Metoda **_make_layer** tvoří konvoluční vrstvy sítě s normalizaci na základě ResidualBlock.

```
def _make_layer(self, block, planes, blocks, stride=1):
    downsample = None
    if stride != 1 or self.inplanes != planes:
        downsample = nn.Sequential(
            nn.Conv2d(self.inplanes, planes, kernel_size=1, stride=stride),
            nn.BatchNorm2d(planes),
        )
    layers = []
    layers.append(block(self.inplanes, planes, stride, downsample))
    self.inplanes = planes
    for i in range(1, blocks):
        layers.append(block(self.inplanes, planes))

    return nn.Sequential(*layers)
```

Zdrojový kód 9 – ResNet _make_layer metoda

ResidualBlock kombinuje konvoluční vrstvy a skip connections z 3.3.3.

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, kernel_size=3,
stride=stride, padding=1),
            nn.BatchNorm2d(out_channels),
            nn.ReLU())
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1,
padding=1),
            nn.BatchNorm2d(out_channels))
        self.downsample = downsample
        self.relu = nn.ReLU()
        self.out_channels = out_channels

    def forward(self, x):
        ...
```

Zdrojový kód 10 – ResidualBlock

Metoda **forward** přijímá na vstup obraz (jeho tensor) a pouští ho přes všechny vrstvé sítě.

```
def forward(self, x):
    x = self.conv1(x)
    x = self.maxpool(x)
    x = self.layer0(x)
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)

    x = self.avgpool(x)
    x = x.view(x.size(0), -1)
    x = self.fc(x)

    return x
```

Zdrojový kód 11 – ResNet forward metoda

CNN má pouze 2 konvoluční a 3 lineární vrstvy. Taková malá hloubka sítě je podmíněna tím, že při učení korelaci, operujeme s mapou vlastnosti. Předpokládáme, ze konvoluční vrstvy ResNet již extrahovaly nejdůležitější vlastnosti a nepotřebujeme mnoho konvolučních vrstev na stráně CNN.

```
class CNN(nn.Module):
    def __init__(self):
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        ...
```

Zdrojový kód 12 – CNN Model

NetworkTrainer je třída pro spouštění procesu učení každé sítě. Přijímá **model** a **data_loader** pro načtení datasetu.

```
class NetworkTrainer:

    def __init__(self, data_loader, model):
        super().__init__()
        self.data_loader = data_loader
        self.model = model
```

Zdrojový kód 13 – NetworkTrainer konstruktor

Metoda **train** v NetworkTrainer zajišťuje trénink a následnou validaci.

```
def train(self, image_size, epochs, batch_size, result_dir, data,
model_name):
```

Zdrojový kód 14 – NetworkTrainer metoda train signatura

Data loader načte dataset a vrátí **train** a **val** loadery, které budou dávkovaně natahovat data.

```
train_loader, val_loader = self.data_loader(data=data, batch_size=batch_size,
imgsize=image_size)
```

Zdrojový kód 15 – Volání data_loaderu

```
train_images, train_labels = load_data(os.path.join(data, 'train'))
train_dataset = CustomTensorDataset(tensors=(train_images, train_labels),
transform=transform)
train_indices = list(range(len(train_dataset)))
if shuffle:
    np.random.shuffle(train_indices)
train_sampler = SubsetRandomSampler(train_indices)
train_loader = torch.utils.data.DataLoader(
    train_dataset, batch_size=batch_size, sampler=train_sampler)
```

Zdrojový kód 16 – Vytvoření train_loaderu

Trénink a následná validace se pouští s příslušnými loadery.

```
for epoch in range(epochs):
    correct = 0
    total = 0
    epoch_train_loss = 0
    epoch_val_loss = 0

    for i, (images, labels) in enumerate(train_loader):
        ...
```

Zdrojový kód 17 – Počátek tréninku

TandemTrainer zprovozní, na GPU, sekvenční učení ResNet, generaci datasetu pro CNN a učení CNN korelací.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')


class TandemTrainer:

    def __init__(
            self,
            num_classes,
            data
    ):
        super().__init__()
        self.resnet_model = ResNet34(ResidualBlock, [3, 4, 6, 3],
num_classes=num_classes).to(device)
        self.cnn_model = CNN(num_classes=num_classes).to(device)
        self.resnet_data = data
        ...
```

Zdrojový kód 18 – TandemTrainer konstruktor přehled

TandemTrain má reference na modely, které bude pouštět pro naučení vlastností a korelace a další potřebné parametry.

TandemTrain sekvenčně pouští učení ResNet, generuje dataset pro CNN pomocí FeatureMapGenerator a pouští další učení CNN na generovaném datasetu.

```
def train(self, params):
    resnet34_trainer = NetworkTrainer(
        model=self.resnet_model,
        data_loader=data_loader,
    )
    cnn_trainer = NetworkTrainer(
        model=self.cnn_model,
        data_loader=data_loader
    )

    resnet34_trainer.train(...)

    feature_maps_generator = FeatureMapGenerator(
        ...
    )
    feature_maps_generator.generate()

    cnn_trainer.train(...)
```

Zdrojový kód 19 – TandemTrainer metoda train přehled

ResNet a CNN, po naučení, produkují soubory s rozlišením *.pth, které jsou výslednými váhami. Soubor generovaný ResNet obsahuje požadované mapy vlastností.

FeatureMapGenerator extrahuje konvoluční vrstvy, provádí inferenci výsledné mapy vlastnosti na původní dataset a ukládá výslednou mapu vlastnosti poslední vrstvy pro každý obrázek.

```
class FeatureMapGenerator:
    def __init__(self, model, path_to_resnet_weights, folder_to_process,
folder_to_save_results):
        self.model = model
        self.loaded_weights = torch.load(path_to_resnet_weights)
        self.model.load_state_dict(self.loaded_weights['model_state_dict'])
        self.conv_layers = []
        self.extract_conv_layers()
        self.transform = set_transforms()
        self.folder_to_process = folder_to_process
        self.folder_to_save_results = folder_to_save_results

    def get_layers_outputs(self, image):
        ...

    def get_feature_maps(self, outputs):
        ...

    def plot_final_feature_map(self, image, folder, input_folder, name):
        ...

    def extract_conv_layers(self):
        ...

    def process_image(self, image):
        ...

    def generate(self):
        ...
        for name, image in zip(images_names, images):
            image = self.transform(image)
            image = image.unsqueeze(0)
            image = image.to(device)

            outputs, names = self.get_layers_outputs(image)
            processed = self.get_feature_maps(outputs)

            final_image = processed[-1]

            self.plot_final_feature_map(final_image, folder, input_folder,
name)
```

Zdrojový kód 20 – FeatureMapGenerator přehled

Následně, stejným způsobem se pouští učení CNN. Po obdržení obou výsledků CNN a ResNet, můžeme provést segmentaci pomocí dynamického okna.

Dynamické okno se inicializuje s mapy vlastností obou klasifikátorů, inicializuje CNN model pro předpovědi a předává ResNet model do FeatureMapGeneratoru pro provádění inference a extrakce vlastností, které se využijí při kreslení masky.

```
class DynamicWindow:

    def __init__(self, resnet_weights, cnn_weights, num_classes):
        self.model_cnn = CNN(num_classes=num_classes).to(device)
        self.loaded_cnn_weights = torch.load(cnn_weights)

self.model_cnn.load_state_dict(self.loaded_cnn_weights['model_state_dict'])
        self.feature_maps_class = FeatureMapGenerator(resnet_weights)
```

Zdrojový kód 21 – Dynamic window konstruktor

Při volání mu se předává cesta do obrazu a další parametry. Obraz se načte, určí jeho velikost a sestaví pyramidu obrazů z 3.3.1.

```
def __call__(self, img_path, min_conf, window_size, window_step, resnet_size,
result_dir):
    img = read_image(img_path)
    img = self.preprocess_image(img, window_size)

    pyramid = self.image_pyramid(np.asarray(img),
scale=self.kwargs['PYR_SCALE'], minSize=window_size)
    rois, locs = self.get_rois_and_locs(pyramid, window_size, window_step,
resnet_size)

    preds, labels, feature_maps_masks = self.get_preds(rois, locs, min_conf)
    nms_labels = self.apply_nms(labels)

    if self.kwargs['VISUALIZE']:
        img_name = os.path.basename(img_path)
        self.visualize_preds(np.array(img), nms_labels, feature_maps_masks,
img_name, result_dir)

    return nms_labels
```

Zdrojový kód 22 – Dynamic window volání

Metoda **image_pyramid** pomocí PYR_SCALE postupně škaluje obraz až do minimální velikosti. Počet přeškálování závisí na velikosti vstupního obrazu a minimální velikosti. Čím větší obraz, tím více vrstev bude mít pyramida.

```
def image_pyramid(self, image, minSize, scale):
    yield image
    while True:
        w = int(image.shape[1] / scale)
        image = imutils.resize(image, width=w)
        if image.shape[0] < minSize[1] or image.shape[1] < minSize[0]:
            break
        yield image
```

Zdrojový kód 23 – Image pyramid

Metoda **get_rois_and_locs** se používá k získání oblastí zájmů a jejich souřadnic přes veškeré vrstvy pyramidy procházením oknem fixované velikosti s krokem zadaným uživatelem.

```
def get_rois_and_locs(self, pyramid, window_size, window_step, resnet_size):
    rois = []
    locs = []
    for image in pyramid:
        scale = self.W / float(image.shape[0])
        for (x, y, roiOrig) in self.sliding_window(image, window_step, win-
dow_size):
            x = int(x * scale)
            y = int(y * scale)
            w = int(window_size[0] * scale)
            h = int(window_size[1] * scale)

            roi = cv2.resize(roiOrig, resnet_size)

            rois.append(roi)
            locs.append((x, y, x + w, y + h))
    return rois, locs
```

Zdrojový kód 24 – Metoda get_rois_and_locs

Pro každý region se zavolá metoda **get_preds.** FeatureMapGenerator, na základě předané oknu mapy vlastností, provede inferenci oblastí zájmů, vrátí tensor pro předání CNN. CNN ohodnotí obdržený tensor a vrátí konfidenci pro každou třídu, na kterou je naučen. Tato pravděpodobnost se porovnává s **conf_score** dá odpověď včela nebo ne.

```
def get_preds(self, rois, locs, min_conf):
    rois = np.array(rois, dtype="float32")
    preds = []
    feature_maps_mask = []
    for roi in rois:
        feature_map, feature_map_mask = self.feature_maps_class.process_im-
age(roi)
        image_from_feature = Image.fromarray(np.uint8(feature_map)).con-
vert('RGB')
        tensor_image = transform(convert_image_to_tensor(image_from_feature))
        self.model_cnn.eval()
        with torch.no_grad():
            pred = self.model_cnn(tensor_image.view(1, 3, 34, 34).to(device))
        preds.append(np.asarray(pred.cpu()[0]))
        feature_maps_mask.append(feature_map_mask)

    preds = list(zip(self.normalize_data(np.as-
array(preds)).argmax(axis=1).tolist(),
                     self.normalize_data(np.as-
array(preds)).max(axis=1).tolist()))

    return self.filter_preds(
        min_conf,
        locs,
        feature_maps_mask,
        preds
    )
```

Zdrojový kód 25 – Metoda get_preds

Pro každé okno, ve kterém byla zaznamenaná včela, zbavíme se překrývajících bounding boxu metodou NMS. Tato metoda sloučí se překrývající boxy.

```
def apply_nms(self, labels):
    nms_labels = {}
    for label in sorted(labels.keys()):
        boxes = np.array([p[0] for p in labels[label]])
        proba = np.array([p[1] for p in labels[label]])
        boxes = non_max_suppression(boxes, proba, overlap-
Thresh=self.kwargs['OVERLAP_THRESH'])
        nms_labels[label] = boxes.tolist()
    return nms_labels
```

Zdrojový kód 26 – Metoda apply_nms

Závěrem, proběhne vizualizace masek a bounding boxů.

```
def visualize_preds(self, img, nms_labels, feature_maps_masks, img_name,
result_dir):
    bee_key = [key for key, value in zip(list(label_dict.keys()),
list(label_dict.values())) if value == 'bee']
    if len(bee_key) > 0:
        label = bee_key[0]
    clone = img.copy()
    fig, ax = plt.subplots(figsize=(self.W, self.H))

    if label in feature_maps_masks:
        f_maps = feature_maps_masks[label]
        clone = self.visualize_masks(f_maps, img, clone)

        boxes = nms_labels[label]
        clone = self.visualize_boxes(clone, boxes, label)

        ax.imshow(clone, cmap='gray')
        os.makedirs(result_dir, exist_ok=True)
        img_path = str(os.path.join(result_dir, img_name))
        plt.savefig(img_path)
```

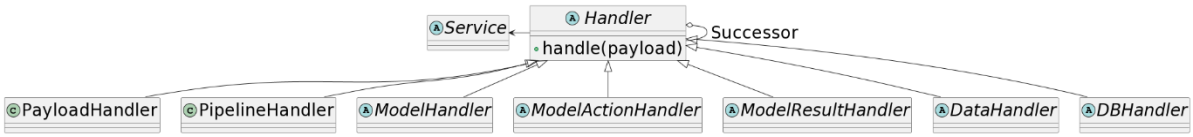
Zdrojový kód 27 – Metoda visualize_preds

6.3 Web app

6.3.1 Server

Pattern **Chain-of-Resposibility** je nejvhodnější cesta implementace aplikace, jak se očekává zpracování různých druhů požadavků různými způsoby.

Navrhneme sadu handlerů pro zpracování kroků požadavku. Handlery mohou záviset na servisech pro vykonání požadované funkcionality.



Obrázek 30 – UML class diagram, handler vrstva

Základní třída **Handler** poskytuje metodu **handle**.

```
from abc import ABC


class Handler(ABC):
    def __init__(self, successor=None):
        self.successor = successor

    def handle(self, payload):
        if self.successor:
            self.successor.handle(payload)
```

Zdrojový kód 28 – Handler třída

Potomci Handleru mohou tuto metodu přetížit pro přidání do payloadu dodatečných hodnot, jako, např. položky potřebné pro další zpracování pipeliny nebo výsledek volání metody služeb, na které specifický handler závisí.

PipelineHandler je počáteční handler pro řetězce všech akcí aplikace. Vkládá do payloadu hodnoty specifické pro akci a ošetřuje výjimky po cestě, voláním funkce **on_exception** a ukončením pipeliny pomocí klíčového slova **return**.

```
class PipelineHandler(Handler):
    def handle(
            self,
            payload,
            on_exception,
            socket=None,
            sid=None,
            on_start=None,
            on_finish=None,
            db_action=None
    ):
        if on_start: payload['on_start'] = on_start
        if on_finish: payload['on_finish'] = on_finish
        if socket: payload['socket'] = socket
        if sid: payload['sid'] = sid
        if db_action: payload['db_action'] = db_action

        try:
            super().handle(payload)
        except PipelineException as e:
            on_exception(str(e))
            return
```

Zdrojový kód 29 – PipelineHandler třída

PinelineHandler obdrží své dodatečné hodnoty z **kontroléru** aplikace. V našem případě, **event-listenery** propagují využité dále k zpracování hodnoty.

```
@socket.on('train')
def train(payload):
    sid = request.sid
    socket.start_background_task(
        target=train_pipeline_handler.handle,
        payload=payload,
        on_start=lambda: socket.emit('train_start'),
        on_finish=lambda: socket.emit('train_finish'),
        on_exception=lambda args: socket.emit('train_exception', {'args':
args}, room=sid)
    )
```

Zdrojový kód 30 – Train event listener

Metoda **start_background_task** pouští celou pipeline ve vlastním vlákně, čímž se umožní paralelní volání funkcí aplikace.

PayloadHandler provede parsing a validaci hodnot payloadu. Validace může vyvolat výjimku, pokud dataset pro úkol či síť nemá správnou strukturu nebo nějaká hodnota je mimo povolený rozsah. Volání výjimky efektivně ukončí zpracování pipeliny, tím, že bude zachycena v **expect** v PipelineHandler, který zpracování pipeliny terminuje.

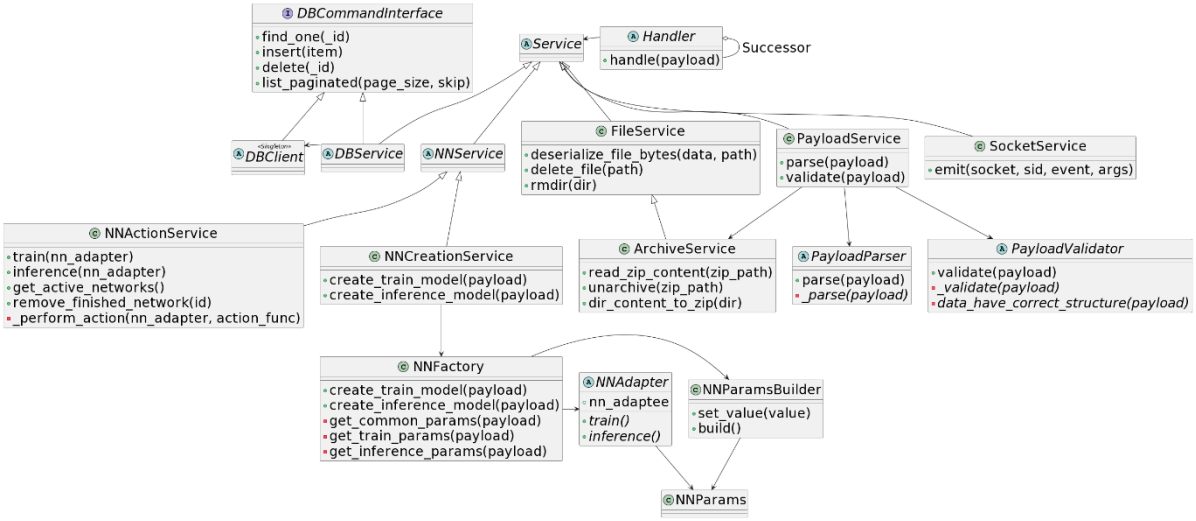
DataHandler připraví data pro budoucí úkol.

ModelHandler vytvoří potřebný pro úkol model.

ModelActionHandler zprovozní vykonání úkolu.

ModelResultHandler zarchivuje výsledky a vynaloží se s archivem příslušným způsobem, dle typu provedeného úkolu.

Pro konkrétní implementaci logiky aplikace, navrhneme sadu servisů.



Obrázek 31 – UML class diagram, servisní vrstva.

PayloadService je závislostí PayloadHandleru, a zprovozňuje parsing a validaci hodnot payloadu. Proto, PayloadService využívá třídy PayloadParser, PayloadValidator a ArchiveService.

PayloadParser a **PayloadValidator** oba implementují pattern **Template Method**. Provádí základní parsing a validaci hodnot, ve svých **public** metodách a rovnou volají své **abstraktní private** metody, které si musí implementovat jejich potomci pro specifický úkol.

ArchiveService je třída pro všechny náklady se soubory během provozu aplikace. Dokáže deserializovat byty do souboru, vymazat samostatný soubor nebo složku se soubory, dearchivovat *.zip archive a archivovat obsah cele složky do *.zip archivu.

NNCreationService zpřístupňuje interface NNFactory.

NNFactory poskytuje interface pro vytvoření modelu pro určitý úkol. Typ modelu je definován uživatelem při zadání parametrů úkolu a je převzat z payloadu.

Vytvoření modelu v NNFactory se skládá ze 3 kroků: 1) vytvoření NNParams pro úkol, 2) instanciování neuronové sítě z knihovny třetí strany, 3) provázání modelu třetí strany a parametrů přes NNAdapter.

NNAdapter umožní fungování mezi sebou objekty s odlišnými interface. Nadefinuje 2 **abstraktní public** metody **train** a **inference**, kde train slouží k účení modelu a inference k předpovědi nebo rozhodování na základě předem naučeného modelu a dat, které síť před tím neviděla.

```
class NNAdapter(ABC):
    def __init__(self, nn_adaptee, params):
        self.nn_adaptee = nn_adaptee
        self.params = params

    @abstractmethod
    def train(self):
        pass

    @abstractmethod
    def inference(self):
        pass
```

Zdrojový kód 31 – NNAdapter třída

Specifický adapter pod specifickou neuronovou síť třetí strany implementuje volání metod svého adaptee.

```
class YoloV8Adapter(NNAdapter):

    def train(self):
        self.nn_adaptee.train(
            data=self.params.data,
            epochs=self.params.epochs,
            batch=self.params.batch_size,
            imgsz=self.params.image_size,
            project=self.params.static_result_dir,
            name=self.params.id
        )

    def inference(self):
        for image in os.listdir(self.params.dataset_dir):
            self.nn_adaptee(
                os.path.join(self.params.dataset_dir, image),
                save=True,
                exist_ok=True,
                conf=self.params.conf_score,
                project=self.params.static_result_dir,
                name=self.params.id
            )
```

Zdrojový kód 32 – YoloV8Adapter

NNParams se tvoří pomocí NNParamsBuilder na základě hodnot z payloadu.

NNParamsBuilder krok za krokem vytváří NNParams. Pattern Builder umožňuje tvořit různé typy a reprezentace objektů s komplexní strukturou.

NNActionService se zachází s vytvořeným adapterem jako se skutečným modelem neuronové sítě a volá jeho metody. NNActionService, také, udržuje informaci o všech aktuálních úkolech, běžících na serveru.

```
class NNActionService(NNService):
    def __init__(self):
        super().__init__()
        self._active_networks = []

    def _perform_action(self, nn_adapter, action_func):
        self._active_networks.append(nn_adapter)
        if nn_adapter.params.on_start:
            nn_adapter.params.on_start()
        action_func(nn_adapter)
```

Zdrojový kód 33 – NNActionService třída

NNActionService vkládá adapter do svého poolu aktivních síti, volá **on_start** funkcí, pokud je přítomná a volá action funkci adapteru.

Pool aktivních sítí je přístupný přes metodu **get_active_networks()**.

Odstránění z poolu probíhá pomoci metody **remove_finished_network(id)**

Funkce **on_start** a **on_finish** slouží k uvědomění klienta o začátku a ukončení akce na serveru.

Volání funkci **on_finish**, odstranění sítě z poolu a mazání dat využitých pro úkol proběhne až po exportu výsledků do databáze nebo do uživatelského prohlížeče.

Za tuto funckionalitu je zodpovědný **ModelResultHandler** a využívá proto **Archive** a **NN Servisy**. Samostatný export je implementován v potomcích tyto třídy.

```
class ModelResultHandler(Handler, ABC):

    def __init__(self, archive_service, nn_service, successor=None):
        super().__init__(successor)
        self.archive_service = archive_service
        self.nn_service = nn_service

    def handle(self, payload):
        params = payload['model'].params
        self._handle(payload)
        self.archive_service.rmdir(params.dataset_parent_dir)
        self.archive_service.rmdir(params.model_result_dir)
        self.nn_service.remove_finished_network(params.id)
        if params.on_finish:
            params.on_finish()
        super().handle(payload)

    @abstractmethod
    def _handle(self, payload):
        pass
```

Zdrojový kód 34 – ModelResultHandler třída

SocketService a **DBService** implementují logiku stažení výsledků do prohlížeče nebo uložení do databáze a jsou závislostmi pro specifické ModelResultHandlery.

DBClient je závislost pro DBService, která ji umožní komunikovat s databází a oba sdílejí společný **DBCommnadInterface**.

Inicializace veškerých servisů, handlerů, provázání potřebných závislosti a sestavení handler pipeline pro úkoly probíhá v souboru **container.py**

```
archive_service = ArchiveService()
train_payload_parser = TrainPayloadParser()
train_payload_validator = TrainPayloadValidator()

train_payload_service = PayloadService(
    file_service=archive_service,
    payload_parser=train_payload_parser,
    payload_validator=train_payload_validator
)
...

train_pipeline_handler = PipelineHandler()
train_payload_handler = PayloadHandler(payload_service=train_payload_service)
...

train_pipeline_handler.successor = train_payload_handler
train_payload_handler.successor = train_data_handler
train_data_handler.successor = train_model_handler
train_model_handler.successor = train_handler
train_handler.successor = train_result_handler
```

Zdrojový kód 35 – container.py. Inicializace tříd, propojení závislosti a sestavení řetězce handlerů

6.3.2 Klient

Klientská strana aplikace se skládá ze 4 stránek: **Landing**, **TrainForm**, **TrainList** a **InferenceFrom**.

```
function App() {
    return (
        <div className="App">
            <Router>
                ...
                <Routes>
                    <Route path="/" element={<Landing/>}/>
                    <Route path="/train" element={<TrainForm/>}/>
                    <Route path="/trains" element={<TrainList/>}/>
                    <Route path="/inference" element={<InferenceForm/>}/>
                </Routes>
            </Router>
        </div>
    );
```

Zdrojový kód 36 – App.js

Landing obsahuje statické informace o použití aplikace.

TrainForm přijímá data a parametry pro trénink

```
<div>
    <select name="network" onChange={(event) =>
setNetwork(event.target.value)}>
        <option>Select Network</option>
        {networks.map((type) => (
            <option key={type} value={type}>
                {type}
            </option>
        ))}
    </select>

    {network === 'YOLOv8' ? (
        <select name="networkSize" onChange={(event) =>
setNetworkSize(event.target.value)}>
            <option>Select Network Size</option>
            {networkSizes.map((size) => (
                <option key={size} value={size}>
                    {size}
                </option>
            ))}
        </select>
    ) : null}

    <select name="imageSize" onChange={(event) =>
setImageSize(event.target.value)}>
        <option>Image Size</option>
        {imageSizes.map((type) => (
            <option key={type} value={type}>
                {type}
            </option>
        ))}
    </select>

    <input type="number" name="epochs" onChange={(event) =>
setEpochs(event.target.value)}/>

    {network === 'ResNet' ? (
        <input type="number" name="epochsCNN" onChange={(event) =>
setCnnEpochs(event.target.value)}/>
    ) : null}

    <input type="number" name="batchSize" onChange={(event) =>
setBatchSize(event.target.value)}/>
    <input type="file" name="data" onChange={(event) =>
setFile(event.target.files[0])}/>
    <button onClick={handleTrain}>Train</button>
</div>
```

Zdrojový kód 37 – TrainForm.js, nastavení hodnot parametrů tréninku

TrainForm validuje hodnoty parametrů a strukturu archivu, než je zasílá na server. Validace zajistí, aby uživatel nezatěžoval server požadavkem, který nemůže být vykonán kvůli nesprávným parametrům.

```
const validateContent = (archiveContent) => {
    if (network === 'YOLOv8') {
        try {
            validateYOLO(archiveContent);
            setValidationError(null);
        } catch (error) {
            setValidationError(error.message);
        }
    }
    /*Resnet validace*/
};
```

Zdrojový kód 38 – TrainForm.js, validace struktury archivu

Archiv s daty nahrány uživatelem prohází dodatečnou kompresi před zasíláním. Tento krok, efektivně, zmenšuje objem data, které bude potřeba přenést po síti, o dvě třetiny.

```
const submitTrain = () => {
    if (file) {
        const zip = new JSZip();
        zip.loadAsync(file).then((archive) => {
            const archiveContent = Object.keys(archive.files).map((fileName)
=> archive.files[fileName].name);
            validateContent(archiveContent);

            if (validationError === null) {
                const zip = new JSZip();
                zip.file(file.name, file);
                zip.generateAsync({type: 'blob'}).then((compressedBlob) => {
                    socket.emit('train', {
                        data: compressedBlob,
                        network: network,
                        networkSize: networkSize,
                        imageSize: imageSize,
                        epochs: epochs,
                        cnnEpochs: cnnEpochs,
                        batchSize: batchSize,
                        datasetName: file.name
                    });
                });
            }
        });
    }
};
```

Zdrojový kód 39 – TrainForm.js, zasílání uživatelských dat na server

InferenceForm je shodná s TrainForm z hlediska funkcionality, ale emituje jiný **inference** event na server a spolu s daty pro inferenci zasílá předem naučené váhy neuronové sítě. Pro YOLO inferenci potřebujeme 1 soubor, pro Tandem 2 soubory s váhy ResNet a CNN

```
zip.generateAsync({type: 'blob'}).then((compressedBlob) => {
    socket.emit('inference', {
        data: compressedBlob,
        network: network,
        confScore: confScore,
        datasetName: images.name,
        weightsName: weights.name,
        cnnWeightsName: cnnWeights ? cnnWeights.name : '',
    });
});
```

Zdrojový kód 40 – InferenceForm.js, zasílání uživatelských dat na server

TrainList zobrazuje, po stránkách, výsledky učení sítí, uložené do databáze. Umožní tyto výsledky stáhnout nebo smazat.

```
<tbody>
{trains.slice(0, pageSize).map((item) => (

    <tr key={item._id}>
        <td>{item._id}</td>
        <td>{item.filename}</td>
        <td>{item.epochs}</td>
        <td>{item.batch_size}</td>
        <td>
            <button onClick={() =>
handleDownload(item._id)}>Download</button>
            <button onClick={() => handleDelete(item._id)}>Delete</button>
        </td>
    </tr>
))}
</tbody>
```

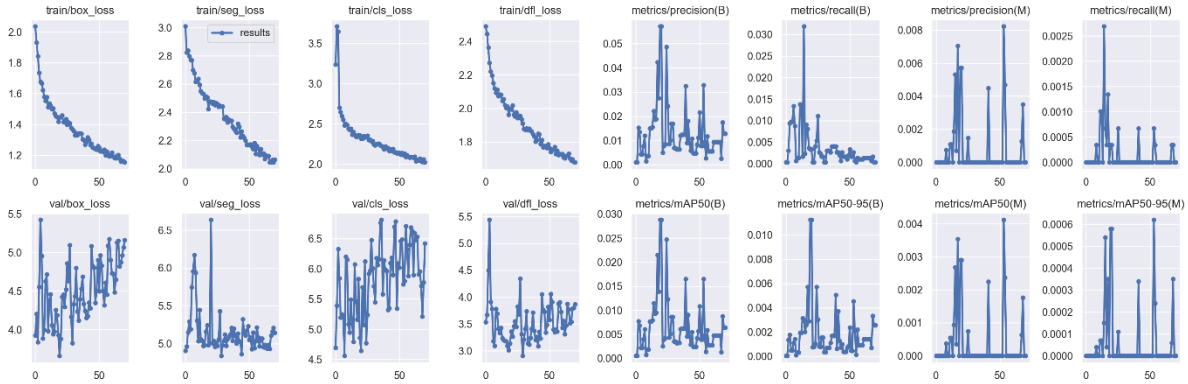
Zdrojový kód 41 – TrainList.js, zobrazení položek z databáze

7 Experimenty

7.1 YOLO

V rámci YOLO experimentů bylo provedeno učení sítě všech velikosti od **nano** do **extra-large**. Nejlepší segmentační výsledky, pro náš objem dat, byly naměřené ze sítě velikosti **large**. Dále jsou uvedené průběhy učení s různým **learning rate** (LR).

Obrázek 32 znázorňuje průběh učení s LR 0.1 na 100 epoch.



Obrázek 32 – YOLO průběh učení, LR 0.1, 100 epoch

Z grafů precision, recall a mAP, na obrazu 32 je vidět, že v průběhu učení tyto hodnoty začaly klesat. Toto naznačuje, že síť se přestává učit. Dodatečně to se zobrazuje v přesnost (**precision**) a zapamatovatelnost (**recall**) pro boxy a masky.

Tabulka č 1. zobrazuje metriky naměřené po učení s LR 0.1.

mAP50(B)	mAP50-95(B)	mAP50(M)	mAP50-95(M)
0.00644	0.00258	0	0

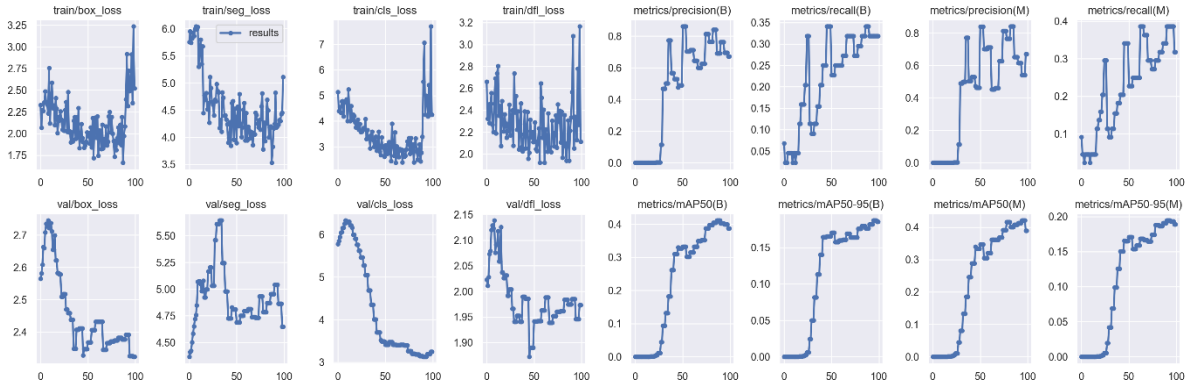
Tabulka 1 – mAP(B), mAP(M) pro LR 0.1

Metriky **mAP(B)** a **mAP(M)** jsou nejdůležitější metriky pro pochopení kvality naučení segmentační sítě. Říkají informaci o střední průměrné přesnosti pro bounding boxy a segmentační masky pro instance s konfidenci 50 %, mAP50-95 pro instance s konfidenci v rozmezí 50–95 %.

Přestože, učení bylo puštěno na 100 epoch, samostatný graf končí před 100. epochou – na 71. Tomuto je tak kvůli YOLO **EarlyStopping** mechanismu, který předčasně ukončí učení sítě, pokud během 50 epoch nebude zaznamenán žádný kvalitní pokrok. Počet epoch pro EarlyStopping se nastavuje pomocí argumentu **patience** při začátku tréninku.

Výsledně, tato síť nebyla schopna segmentovat ani jednu včelu.

Obrázek 33 znázorňuje průběh učení s LR 0.001 na 100 epoch.



Obrázek 33 – YOLO průběh učení, LR 0.001, 100 epoch

Z grafů precision, recall a mAP, na obrazu 33 je vidět, že v průběhu tréninku, síť se dostála do lokální extrému. Je to patrné z preciznosti a zapamatovatelnosti. Nabývají hodnotu pro lokální extrém, a pak začínají klesat. Poté, síť již není schopná překonat ten extrém a zůstává v něm. LR 0.001 je moc pomalý.

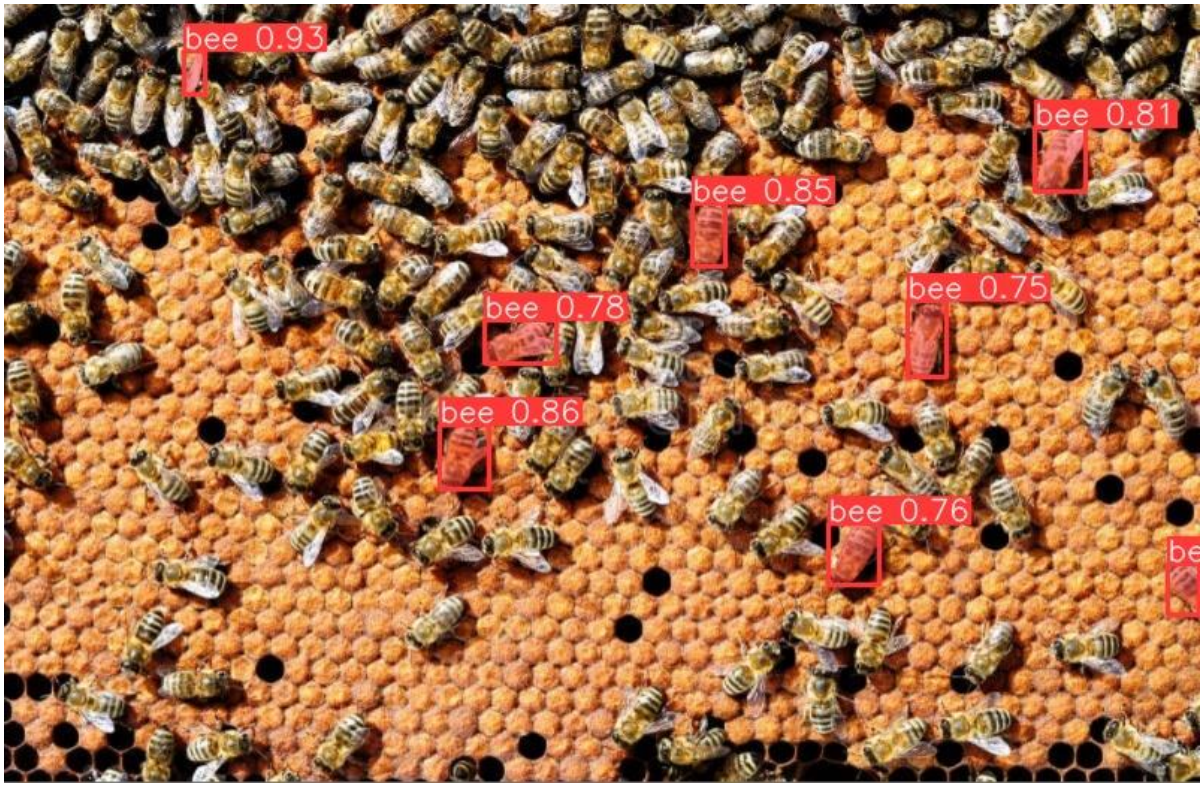
Tabulka č 2. zobrazuje metriky naměřené po učení s LR 0.001.

mAP50(B)	mAP50-95(B)	mAP50(M)	mAP50-95(M)
0.38772	0.18574	0.3893	0.18887

Tabulka 2 – mAP(B), mAP(M) pro LR 0.001

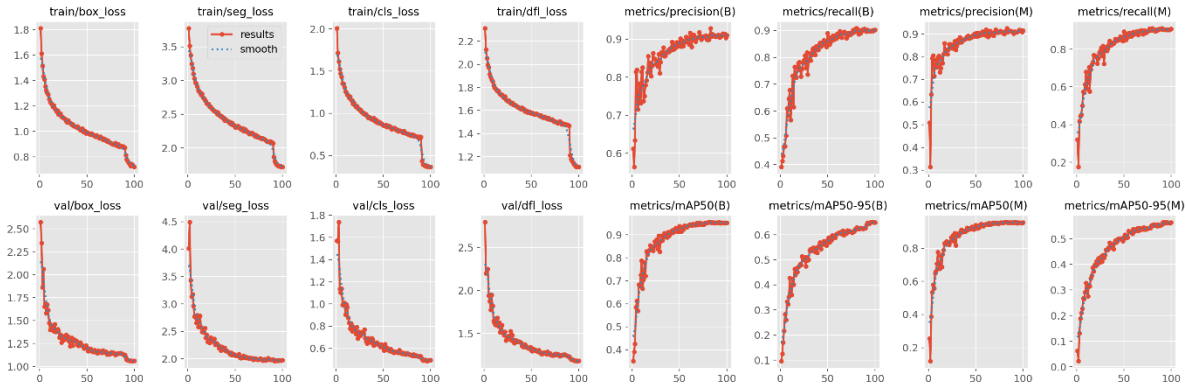
Obrázky 34 a 35 ukazují výsledky segmentace s LR 0.001. Síť je schopna segmentovat velmi malý počet včel. Navíc, je vidět, že síť přijímá samostatné včelí křídlo za včelu. Tato síť není aplikovatelná pro kvalitní segmentaci.

Obrázek 34 – Včely na plástu 1, výsledky segmentace LR 0.001. YOLOv8



Obrázek 35 – Včely na plástu 2, výsledky segmentace LR 0.001. YOLOv8

Obrázek 36 znázorňuje průběh učení s LR 0.01 na 100 epoch.



Obrázek 36 – YOLO průběh učení, LR 0.01, 100 epoch

Z grafů precision, recall a mAP, na obrazu 36 je vidět, že toto je nejlepší z otestovaných LR pro síť. Metriky naznačují, že se síť dostala do globálního extrému čili naučila se.

Tabulka č 3. zobrazuje metriky naměřené po učení s LR 0.01.

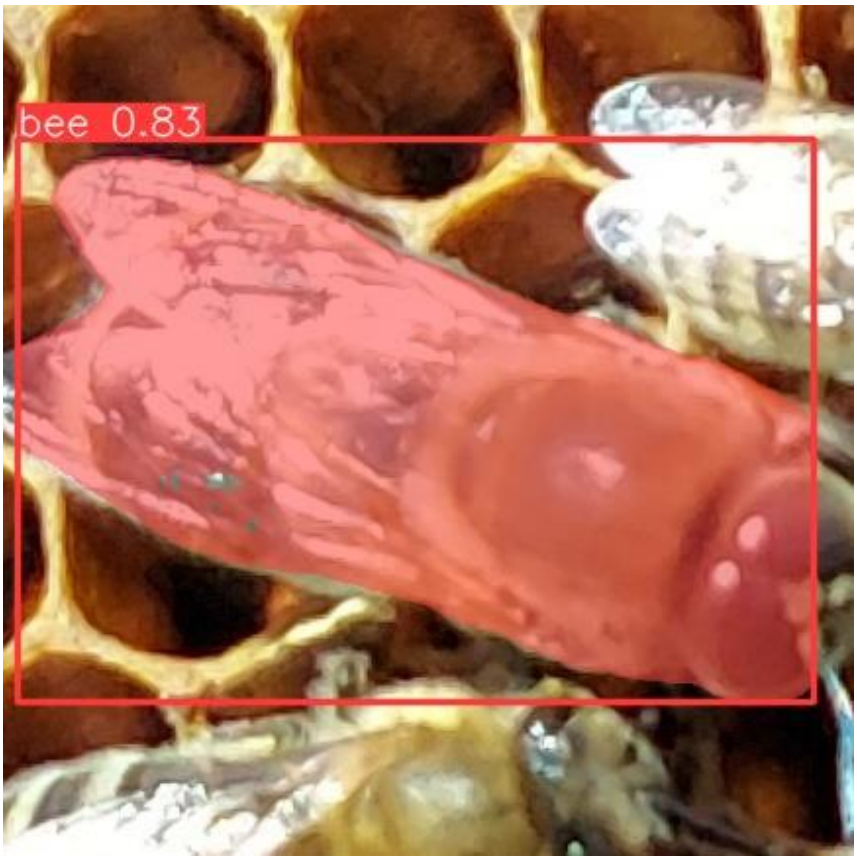
mAP50(B)	mAP50-95(B)	mAP50(M)	mAP50-95(M)
0.95311	0.64956	0.95621	0.56367

Tabulka 3 – mAP(B), mAP(M) pro LR 0.01

Obrázky 37, 38 ukazují výsledky segmentace s LR 0.01 pro více včel na plástu a obrázek 39 pro jednu včelu.

Obrázek 37– Včely na plástu 1, výsledky segmentace LR 0.01. YOLOv8

Obrázek 38– Včely na plástu 2, výsledky segmentace LR 0.01. YOLOv8



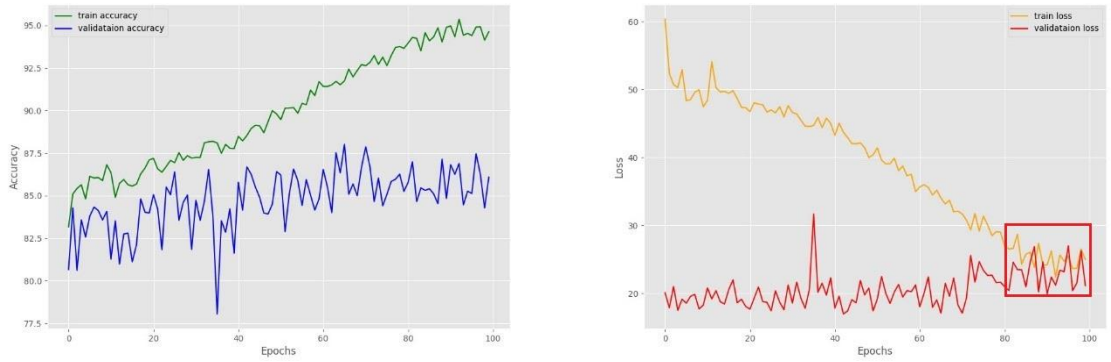
Obrázek 39– Jednotlivá včela, výsledek segmentace LR 0.01. YOLOv8

7.2 Vlastní řešení

V první řádě provedeme učení ResNet na edge (hranových) a barevným datasetu pro naučení včelích vlastností. Všechny ResNet byly naučené na velikost obrazu 224×224. Následně provedeme učení CNN a segmentaci.

7.2.1 ResNet Hranové Obrazy

Obrázek 40 znázorňuje průběh učení ResNet na hranových datasetu, na 100 epoch s LR 0.01



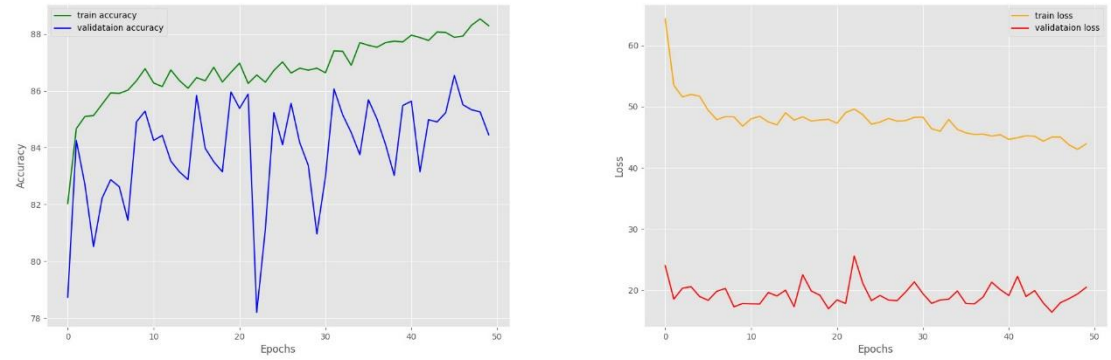
Obrázek 40– ResNet průběh učení hranové obrazy, LR 0.01, 100 epoch

Na grafu pro accuracy, vlevo, je vidět, že po 40. epoše nastává velká divergence mezi train a val přesnostmi. Train neustálé stoupá, mezitím val zůstává na místě, až začíná klesat k 70 epoše.

Na grafu pro loss, vpravo, train a val loss se dokonce kříží.

Síť se přeučuje na train dataset.

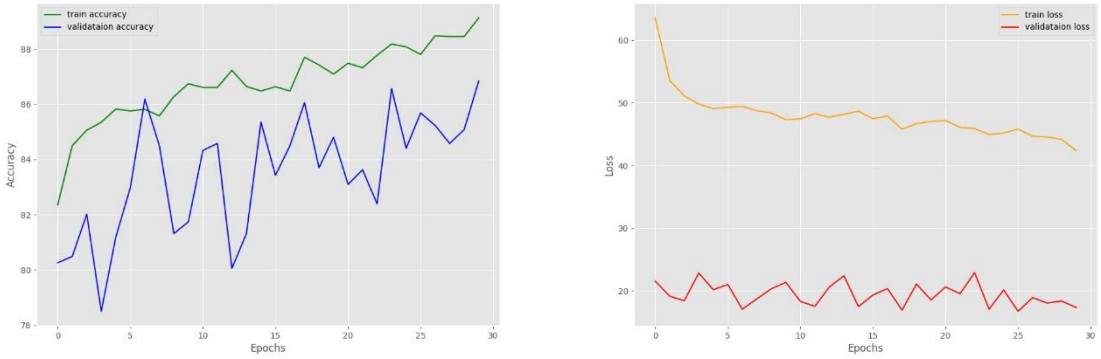
Obrázek 41 znázorňuje průběh učení ResNet na hranovém datasetu, na 50 epoch s LR 0.01



Obrázek 41– ResNet průběh učení hranové obrazy, LR 0.01, 50 epoch

Na grafu pro accuracy, vlevo, je vidět, že mezi train na val, také po 40. epoše začíná divergence, ale není natolik velká, abychom řekli, že se síť přeučuje. Samostatný průběh vypadá dobře. Na grafu pro loss, vpravo, train loss se postupně snižuje a val loss je poměrně nízký a nestoupá.

Obrázek 42 znázorňuje průběh učení ResNet na hranovém datasetu, na 30 epoch s LR 0.01



Obrázek 42– ResNet průběh učení hranové obrazy, LR 0.01, 30 epoch

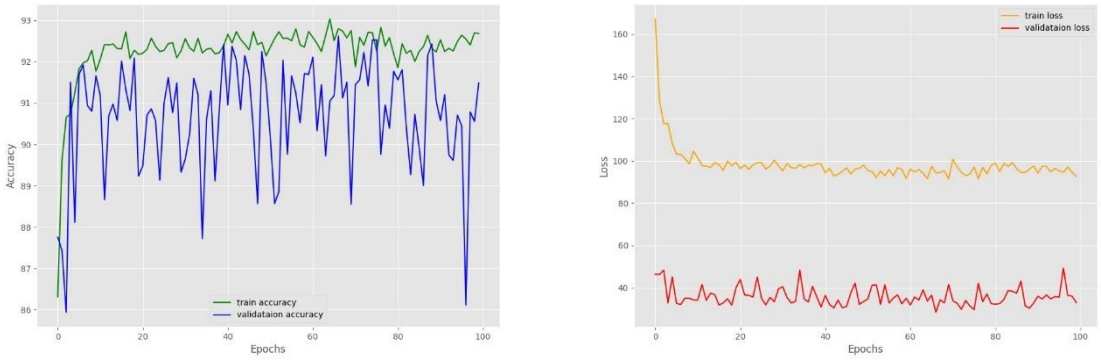
Na grafu pro accuracy, vlevo, je vidět, že train a val accuracy stoupají během celého učení. Maximální dosažená hodnota train je 89 a 87 pro val.

Na grafu pro loss, vpravo, train a val loss postupně klesá.

V porovnání se síti naučenou na 50 epoch, je vidět, že dosažené accuracy je skoro stejné a není žádná divergence mezi train a val.

7.2.2 ResNet Barevné Obrazy

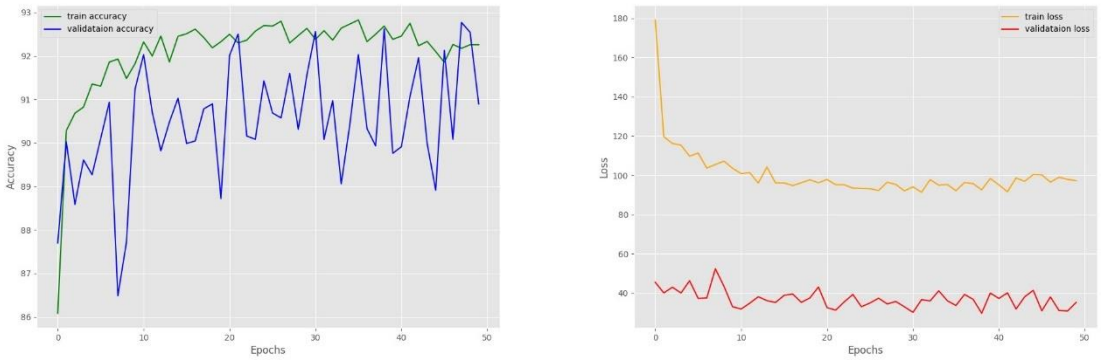
Obrázek 43 znázorňuje průběh učení ResNet na barevném datasetu, na 100 epoch s LR 0.01



Obrázek 43– ResNet průběh učení barevné obrazy, LR 0.01, 100 epoch

Na grafu pro accuracy, je vidět, že se síť nepřeučuje, Z 15. epochy hodnota train accuracy ResNet nabývá svých maximálních hodnot v rozmezí 92–93 a zůstává v tomto rozmezí přes celý trénink a nestagnuje od přeučení.

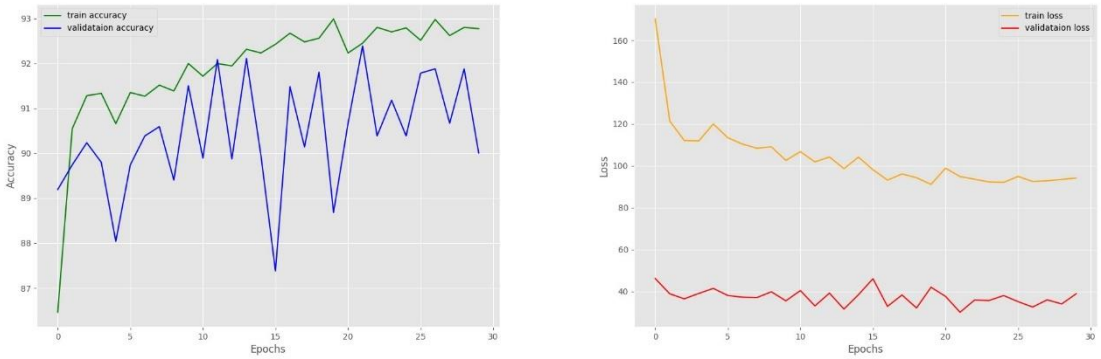
Obrázek 44 znázorňuje průběh učení ResNet na barevným datasetu, na 50 epoch s LR 0.01



Obrázek 44– ResNet průběh učení barevné obrazy, LR 0.01, 50 epoch

Je vidět, že průběh učení na 50 epoch se moc neliší od 100. Učení proběhlo bez závad. Val accuracy je méně volatilnější. Toto není působeno počtem epoch, ale váhami sítě při její inicializace.

Obrázek 45 znázorňuje průběh učení ResNet na barevným datasetu, na 30 epoch s LR 0.01

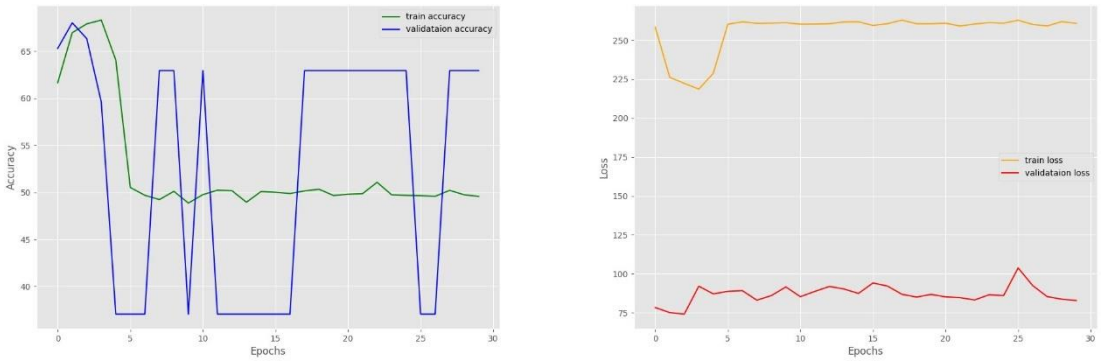


Obrázek 45– ResNet průběh učení barevné obrazy, LR 0.01, 30 epoch

ResNet na 30 epoch dosahuje těch samých výsledků, co dva předchozí modely, ale dělá to za menší dobu. V tomto případě, budeme preferovat ResNet naučený za 30 epoch, jak není kvalitní rozdíl mezi 3 modely a menší počet epoch ušetří čas při učení.

7.2.3 CNN Hranové Obrazy

Obrázek 46 znázorňuje průběh učení CNN, na základě mapy vlastností ResNet naučeného na hranových obrazech, na 30 epoch s LR 0.01.

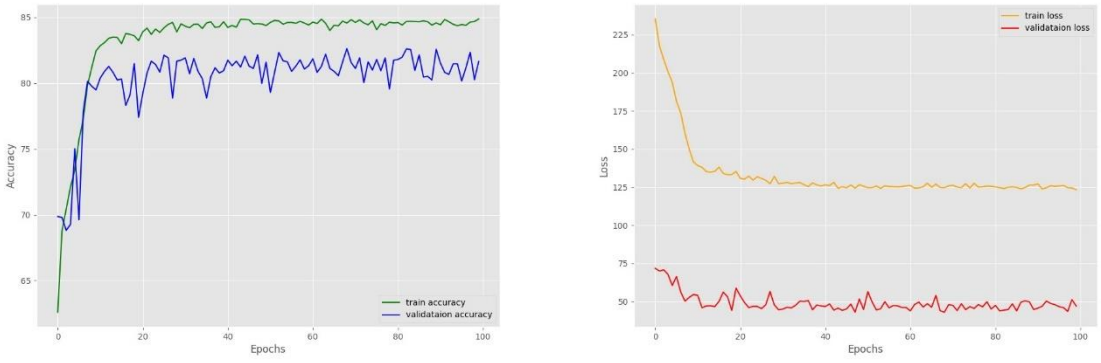


Obrázek 46– CNN průběh učení hranové obrazy, LR 0.01, 30 epoch

Z grafů je vidět, že se CNN vůbec neučí a nebude schopna rozlišit mapu vlastností včely od ne včely. Toto je působeno malým objemem příznaků u hranových obrazů, na kterých byl naučen ResNet.

7.2.4 CNN Barevné Obrazy

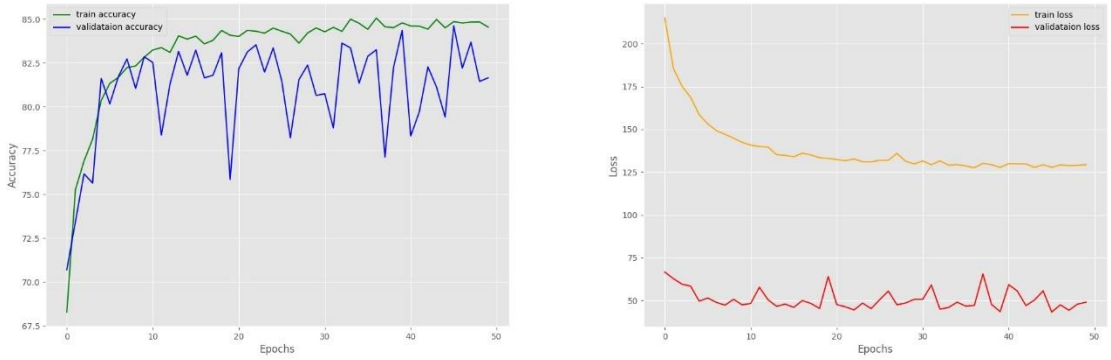
Obrázek 47 znázorňuje průběh učení CNN na základě mapy vlastností ResNet naučeného na barevných obrazech, na 100 epoch s LR 0.01.



Obrázek 47– CNN průběh učení barevné obrazy, LR 0.01, 100 epoch

Na grafu pro accuracy, je vidět, že z 24. epochy hodnota train accuracy CNN nabývá svých maximálních hodnot v rozmezí 82–85 a zůstává v tomto rozmezí přes celé učení a nestagnuje od přeučení.

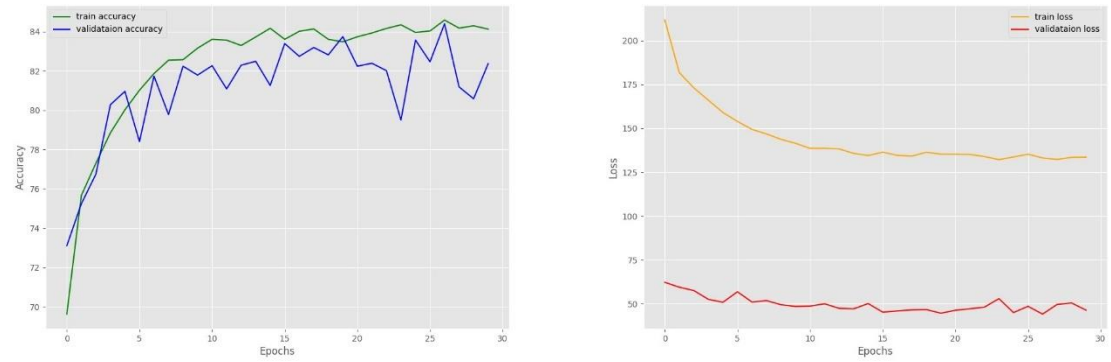
Obrázek 48 znázorňuje průběh učení CNN na základě mapy vlastností ResNet naučeného na barevných obrazech, na 50 epoch s LR 0.01.



Obrázek 48– CNN průběh učení barevné obrazy, LR 0.01, 50 epoch

Průběh učení a accuracy pro 50 epoch je shodný s průběhem pro 100 epoch. Train a val accuracy dosahují stejných hodnot.

Obrázek 49 znázorňuje průběh učení CNN na základě mapy vlastností ResNet naučeného na barevných obrazech, na 30 epoch s LR 0.01.



Obrázek 49– CNN průběh učení barevné obrazy, LR 0.01, 30 epoch

CNN na 30 epoch dosahuje těch samých výsledků, co dva předchozí modely, ale dělá to za menší dobu. V tomto případě, budeme preferovat CNN naučený za 30 epoch pro segmentaci.

7.2.5 Segmentace

Pro segmentaci vezmeme ResNet a CNN naučené na barevnách obrazech za 30 epoch.
Obrázky 50, 51 ukazují výsledky segmentace pro více včel na plástu a obrázek 51 pro 7 jednotlivých včel.

Obrázek 50– Včely na plástu 1, výsledky segmentace. ResNet, CNN, Dynamické okno

Obrázek 51– Včely na plástu 2, výsledky segmentace. ResNet, CNN, Dynamické okno

Obrázek 52– Jednotlivé včely, výsledky segmentace. ResNet, CNN, Dynamické okno

8 Výsledky a diskuze

YOLO úspěšně segmentuje jak včely na obrázcích plástů, tak i jednotlivé včely a zvýrazňuje jejich těla a křídla. Metriky jako mAP50-95(M) jsou vysoké – 0.56367, což ukazuje na celkově dobrý výkon segmentace detekovaných včel. Však je přítomná malá chyba, kde při segmentaci včely jsou zachyceny sousední části včel.

YOLO nedetekuje všechny včely na obrazcích plástů. Pro zlepšení výsledků segmentace je potřeba zvýšit úplnost detekce přidáním nových dat a vytvořením anotací.

Vlastní řešení, na obrázcích s velkým počtem včel, ukazuje nízkou přesnost kvůli falešně pozitivním segmentům, kde nejsou žádné včely, ale přesto jsou rozpoznány jako včely. To je pravděpodobně způsobeno tím, že klasifikátor byl trénován na obrázcích s jednotlivými včelami a pomocí dat map objektů, která nebyla součástí trénovací sady.

Velikosti okna se liší během segmentace a zachycují pouze části včel, jako jsou křídla nebo hlava, namísto celého hmyzu. To se liší od tréninkových dat, což vede k velkému počtu falešně pozitivních výsledků.

Na snímcích samostatných včel vlastní řešení efektivně extrahuje segmenty, aniž by potřebovalo anotaci, což naznačuje jeho účinnost při práci s tímto typem dat.

9 Závěr

Hlavním cílem práce bylo navrhnutím a implementace programové vybavení pro segmentaci včel na obrázcích plástů.

Exitující řešení, YOLO, prokázalo svou velkou efektivitu v segmentaci a má potenciál k vylepšení výsledků při zvětšení objemů trénovacích dat.

Vlastní řešení, založené na dynamickém okně a klasifikátorů, nedokázalo provést úplnou spolehlivou segmentaci včel na obrazcích plástů. Však, vlastní řešení, dosáhlo segmentací instancí shodných s trénovacími daty.

Toto lze považovat za úspěch, jak pokus o segmentaci něčeho bez anotovaných dat je netriviální úkol. V neuronových sítích, zatím, nebylo vyvinuto žádné funkční řešení, které by umožnilo sémantickou segmentaci bez vytvoření anotací. V rámci vlastního řešení byl proveden pokus o vyřešení právě tohoto problému.

Obě segmentační řešení jsou implementovaná zvlášť a jejich funkcionalita je zpřístupněna pomocí web aplikaci. Aplikace umožňuje zveřejnit funkcionalitu i dalších neuronových sítí připojením příslušné závislosti k aplikaci a implementací adapteru.

10 Seznam použité literatury, obrázků, tabulek a rovnic

10.1 Seznam použité literatury

[1] LIU, Haotian, Rafael A. RIVERA SOTO, Fanyi XIAO a Yong Jae LEE. YolactEdge: Real-time Instance Segmentation on the Edge [online]. [cit. 2023-09-18]. Dostupné z: doi:https://doi.org/10.48550/arXiv.2012.12259

[2] PAUL, Sudipto, Dr. Md Taimur AHAD a Md. Mahedi HASAN. Brain Cancer Segmentation Using YOLOv5 Deep Neural Network [online]. [cit. 2023-09-18]. Dostupné z: doi:https://doi.org/10.48550/arXiv.2212.13599

[3] HASAN, Md. Mahedi, Aritejh Kr GOIL, Henryk CHAN, et al. A Novel Application for Real-time Arrhythmia Detection using YOLOv8 [online]. [cit. 2023-09-18]. Dostupné z: doi:https://doi.org/10.48550/arXiv.2305.16727

[4] GUPTA, Lalit, Utthara Gosa MANGAI a Sukhendu DAS. Integrating region and edge information for texture segmentation using a modified constraint satisfaction neural network. Image and Vision Computing [online]. [cit. 2023-09-18]. ISSN 0262-8856. Dostupné z: doi:https://doi.org/10.1016/j.imavis.2007.12.004

[5] BU, Xingyuan, Zhi GAO a Yunde JIA. Deep convolutional network with locality and sparsity constraints for texture classification. Image and Vision Computing [online]. [cit. 2023-09-18]. ISSN 0031-3203. Dostupné z: doi:https://doi.org/10.1016/j.patcog.2019.02.003

[6] HE, Kaiming, Xiangyu ZHANG, Shaoqing REN a Jian SUN. Deep Residual Learning for Image Recognition. [online]. [cit. 2023-09-18]. Dostupné z: doi:https://doi.org/10.48550/arXiv.1512.03385

[7] HLAVÁC, Václav a Milan ŠONKA. Pocítacové videní. Praha: Grada, 1992. ISBN 80-85424-67-3

[8] YOLO od Ultralytics [online]. [cit. 2023-09-18]. Dostupné z: https://github.com/ultralytics/ultralytics

[9] CVAT [online]. [cit. 2023-09-16]. Dostupné z: https://www.cvat.ai/

[10] VoTT [online]. [cit. 2023-09-16]. Dostupné z: https://github.com/microsoft/VoTT

[11] Bee or wasp? [online]. [cit. 2023-09-12]. Dostupné z: https://www.kaggle.com/datasets/jerzydziewierz/bee-vs-wasp

[12] Honey Bee pollen [online]. [cit. 2023-09-12]. Dostupné z: https://www.kaggle.com/datasets/ivanfel/honey-bee-pollen

[13] The BeeImage Dataset [online]. [cit. 2023-09-12]. Dostupné z: https://www.kaggle.com/datasets/jenny18/honey-bee-annotated-images

[14] KUNC, Martin, Pavel DOBEŠ, Jana HURYCHOVÁ, et al. The Year of the Honey Bee (Apis mellifera L.) with Respect to Its Physiology and Immunity: A Search for Biochemical Markers of Longevity [online]. [cit. 2023-09-30]. Dostupné z: doi:10.3390/insects10080244

[15] Python oficiální stránka [online]. [cit. 2023-10-01]. Dostupné z: https://www.python.org/

[16] Flask 2.3.x dokumentace [online]. [cit. 2023-10-01]. Dostupné z: https://flask.palletsprojects.com/en/2.3.x/changes/

[17] Flask Cors dokumentace [online]. [cit. 2023-10-01]. Dostupné z: https://flask-cors.readthedocs.io/en/v1.1/

[18] Flask PyMongo dokumentace [online]. [cit. 2023-10-01]. Dostupné z: https://flask-pymongo.readthedocs.io/en/latest/

[19] Flask SocketIO GitHub [online]. [cit. 2023-10-01]. Dostupné z: https://github.com/miguelgrinberg/Flask-SocketIO

[20] Eventlet oficiální stránka [online]. [cit. 2023-10-01]. Dostupné z: https://eventlet.net/

[21] PyTorch oficiální stránka [online]. [cit. 2023-10-01]. Dostupné z: https://pytorch.org/

[22] JavaScript MDN dokumentace [online]. [cit. 2023-10-01]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[23] Node.js oficiální stránka [online]. [cit. 2023-10-01]. Dostupné z: https://nodejs.org/en

[24] React.js oficiální stránka [online]. [cit. 2023-10-01]. Dostupné z: https://react.dev/

[25] SocketIO klientské API [online]. [cit. 2023-10-01]. Dostupné z: https://socket.io/docs/v4/client-api/

[26] MongoDB oficiální stránka [online]. [cit. 2023-10-01]. Dostupné z: https://www.mongodb.com/

[27] NVIDIA CUDA Toolkit 11.8 dokumentace [online]. [cit. 2023-10-01]. Dostupné z: https://docs.nvidia.com/cuda/archive/11.8.0/

10.2 Seznam použitých obrázků a tabulek

[28] **Obrázek 4 - Princip fungování klasifikátoru [28]:** Výpočetní inteligence: Přednášky [online]. In: SKRBEK, Miroslav. s. 111 [cit. 2023-09-11]. Dostupné z: https://elearning.jcu.cz/pluginfile.php/316893/mod_resource/content/8/vi_predn.pdf

[29] **Obrázek 5 - Kernel [29]**: Kernel. In: Sipl.eelabs.technion.ac.il [online]. [cit. 2023-09-11]. Dostupné z: https://sipl.eelabs.technion.ac.il/wp-content/uploads/sites/6/2016/10/project-image-1599-2-13.png

[30] **Obrázek 6 - Softmax funkce [30]**: Softmax funkce. In: Researchgate.net [online]. [cit. 2023-09-18]. Dostupné z: https://www.researchgate.net/figure/Softmax-function-image_fig1_325856086

[31] **Obrázek 7 - Softmax funkce aplikace [31]**: Softmax funkce aplikace. In: Towardsdatascience.com [online]. [cit. 2023-09-18]. Dostupné z: https://towardsdatascience.com/softmax-activation-function-explained-a7e1bc3ad60

[32] **Obrázek 8 – Sigmoida funkce [32]**: Sigmoida funkce. In: Towardsdatascience.com [online]. 2018 [cit. 2023-09-11]. Dostupné z: https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e

[33] **Obrázek 9 – ReLU funkce [33]**: ReLU funkce. In: Researchgate.net [online]. 2019 [cit. 2023-09-12]. Dostupné z: https://www.researchgate.net/figure/The-plot-of-the-ReLU-function_fig5_335540811

[34] **Obrázek 11 - Max Pooling příklad [34]**: Max Pooling příklad. In: Iq.opengenus.org [online]. [cit. 2023-09-12]. Dostupné z: https://iq.opengenus.org/pooling-layers/

[35] **Obrázek 12 – FC vrstva [35]**: FC vrstva. In: Makina-corpus.com [online]. [cit. 2023-09-12]. Dostupné z: https://makina-corpus.com/sig-webmapping/extraction-dobjets-pour-la-cartographie-par-deep-learning-choix-du-modele

[36] **Obrázek 13 – CNN architektura [36]**: CNN architektura. In: Guandi1995.github.io [online]. [cit. 2023-09-12]. Dostupné z: https://guandi1995.github.io/Classical-CNN-architecture/

10.3 Seznam obrázků, tabulek a zdrojových kódu

10.3.1 Seznam obrázků

10.3.2 Seznam zdrojových kódu

10.3.3 Seznam rovnic

10.3.4 Seznam tabulek

Seznam Příloh

Přílohy

A. COCO2YOLO Script

```
import json
import os
import numpy as np
from PIL import Image


def get_img_ann(image_id, data):
    img_ann = []
    isFound = False     for ann in data['annotations']:
        if ann['image_id'] == image_id:
            img_ann.append(ann)
            isFound = True     if isFound:
        return img_ann     else:
        return None


def get_img(
        filename,
        data,
        data_type,
        train_folder_path,
        val_folder_path
):
    train_folder = os.path.basename(train_folder_path)
    for img in data['images']:
        if data_type == train_folder:
            if img['file_name'] == os.path.join(train_folder, filename):
                img['img_data'] = Image.open(os.path.join(train_folder_path, filename))
                return img          else:
            if img['file_name'] == filename:
                img['img_data'] = Image.open(os.path.join(val_folder_path, filename))
                return img


def create_labels(
        input_images,
        input_json,
        output_labels,
        data_type,
        label_type,
        heigh_thresh,
        train_folder_path,
        val_folder_path
):
    train_folder = os.path.basename(train_folder_path)
    if data_type == train_folder:
        file_names = os.listdir(os.path.join(input_images, os.listdir(input_images)[0]))
    else:
        file_names = os.listdir(input_images)

    f = open(input_json)
    data = json.load(f)
    f.close()

    count = 0     for filename in file_names:

        img = get_img(filename, data, data_type, train_folder_path, val_folder_path)
        img_id = img['id']
        img_w = img['width']
```

```
        img_h = img['height']

        img_ann = get_img_ann(img_id, data)

        if img_ann:
            file_object = open(f"{output_labels}/{filename.split('.jpg')[0]}.txt", "a")

            for ann in img_ann:
                current_category = ann['category_id'] - 1

                if heigh_thresh is not None and ann['bbox'][3] < heigh_thresh:
                    continue

                if label_type == 'box':
                    current_bbox = ann['bbox']
                    x = current_bbox[0]
                    y = current_bbox[1]
                    w = current_bbox[2]
                    h = current_bbox[3]

                    x_centre = (x + (x + w)) / 2
                    y_centre = (y + (y + h)) / 2

                    x_centre = x_centre / img_w
                    y_centre = y_centre / img_h
                    w = w / img_w
                    h = h / img_h

                    x_centre = format(x_centre, '.6f')
                    y_centre = format(y_centre, '.6f')
                    w = format(w, '.6f')
                    h = format(h, '.6f')

                    file_object.write(f"{current_category} {x_centre} {y_centre} {w}
{h}\n")

                else:
                    s = [j for i in ann['segmentation'] for j in i]
                    s = (np.array(s).reshape(-1, 2) / np.array([img_w, img_h])).reshape(-
1).tolist()
                    file_object.write(f"{current_category} {' '.join(str(x) for x in
s)}\n")

            file_object.close()
        count += 1
```

B. Bee Slice Script

```
import json
import os
import numpy as np
from PIL import Image


def get_img_ann(image_id, data):
    img_ann = []
    isFound = False
    for ann in data['annotations']:
        if ann['image_id'] == image_id:
            img_ann.append(ann)
            isFound = True
    if isFound:
        return img_ann
    else:
        return None
```

```
def get_img(
        filename,
        data,
        data_type,
        train_folder_path,
        val_folder_path
):
    train_folder = os.path.basename(train_folder_path)
    for img in data['images']:
        if data_type == train_folder:
            if img['file_name'] == os.path.join(train_folder, filename):
                img['img_data'] = Image.open(os.path.join(train_folder_path, filename))
                return img
        else:
            if img['file_name'] == filename:
                img['img_data'] = Image.open(os.path.join(val_folder_path, filename))
                return img



def create_labels(input_images, input_json,
                  output_labels, output_images,
                  data_type, label_type,
                  heigh_thresh):
    if data_type == 'train':
        file_names = os.listdir(os.path.join(input_images, os.listdir(input_images)[0]))
    else:
        file_names = os.listdir(input_images)

    f = open(input_json)
    data = json.load(f)
    f.close()

    count = 0
    for filename in file_names:
        img = get_img(filename, data, data_type)
        img_id = img['id']
        img_ann = get_img_ann(img_id, data)

        if img_ann:
            for ann in img_ann:

                current_category = ann['category_id'] - 1

                if heigh_thresh is not None and ann['bbox'][3] < heigh_thresh:
                    continue

                if label_type == 'box':
                    current_bbox = ann['bbox']
                    x = current_bbox[0]
                    y = current_bbox[1]
                    w = current_bbox[2]
                    h = current_bbox[3]

                    x_centre = (x + (x + w)) / 2
                    y_centre = (y + (y + h)) / 2

                    new_x = x_centre - max(w / 2, h / 2)
                    new_y = y_centre - max(w / 2, h / 2)
                    w_crop, h_crop = max(w, h)
                    new_image = img['img_data'].crop(new_x, new_y, w_crop, h_crop)
                    new_image.save(f"{output_labels}/" + str(ann['id']) + '.jpg')


                else:
```

```
                    s = [j for i in ann['segmentation'] for j in i]

                    current_bbox = ann['bbox']
                    x = current_bbox[0]
                    y = current_bbox[1]
                    w = current_bbox[2]
                    h = current_bbox[3]

                    x_centre = (x + (x + w)) / 2
                    y_centre = (y + (y + h)) / 2

                    new_x = x_centre - max(w / 2, h / 2)
                    new_y = y_centre - max(w / 2, h / 2)
                    w_crop, h_crop = max(w, h), max(w, h)
                    new_image = img['img_data'].crop((new_x, new_y, new_x + w_crop, new_y
+ h_crop))
                    new_image.save(f"{output_images}/" + str(ann['image_id']) + '_' +
str(ann['id']) + '.jpg')

                    seg_ann_new = (np.array([np.array([coord[0] - new_x, coord[1] -
new_y]) for coord in
                                             np.array(s).reshape(-1, 2)]) /
np.array([w_crop, h_crop])).reshape(
                        -1).tolist()

                    new_seg_file_object =
open(f"{output_labels}/{ann['image_id']}_{ann['id']}.txt", "a")
                    new_seg_file_object.write(f"{current_category} {' '.join(str(x) for x
in seg_ann_new)}\n")

                    new_seg_file_object.close()
        count += 1
```