VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

DETEKCE A ANALÝZA NARUŠITELŮ VE SLEDOVANÉ OBLASTI

DETECTION AND ANALYSIS OF VIOLATORS IN THE MONITORED AREA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. JAKUB SADÍLEK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. TOMÁŠ GOLDMANN**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání diplomové práce

Student: **Sadílek Jakub, Bc.**
Program: Informační technologie a umělá inteligence
Specializace: Vývoj aplikací
Název: **Detekce a analýza narušitelů ve sledované oblasti**
Detection and Analysis of Violators in the Monitored Area
Kategorie: Umělá inteligence

Zadání:

1. Seznamte se s problematikou konvolučních neuronových sítí určených pro detekci objektů v obraze. Dále sumarizujte informace o dostupných řešeních pro detekci a rozpoznávání osob na základě obličeje. Primárně se zaměřte na řešení z posledních 5 let.
2. Navrhněte řešení, které dokáže detekovat narušitele ve sledované oblasti. V případě, že je jako narušitel detekovaná osob, systém provede identifikaci osoby na základě snímku obličeje. Řešení navrhněte jako klient-server aplikaci.
3. Navržené řešení implementujte v programovacím jazyce Python.
4. Otestujte funkčnost řešení a proveďte experimenty s reálnými záznamy zaměřené na určení výkonosti systému. Navrhněte další možná rozšíření.

Literatura:

- MICHELUCCI, Umberto. *Advanced applied deep learning: convolutional neural networks and object detection*. Apress, 2019.
- CAVAZOS, Jacqueline G., et al. Strategies of Face Recognition by Humans and Machines. In: *Deep Learning-Based Face Analytics*. Springer, Cham, 2021. p. 361-379.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 a 2.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Goldmann Tomáš, Ing.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 18. května 2022
Datum schválení: 3. listopadu 2021

Abstrakt

Cílem této práce je vytvořit internetovou aplikaci pro detekci a analýzu narušitelů ve sledované oblasti. Taková aplikace může následně posloužit k automatizovanému zpracování videozáznamů z bezpečnostních kamer či jinak pořízených videí ze střežené oblasti. První část práce je zaměřena na teorii neuronových sítí pro detekci a klasifikaci objektů v obraze a rozpoznávání osob podle obličeje. V další části jsou popsány použité technologie pro vývoj aplikací. Výsledkem pak je klient-server aplikace s možností konfigurace zpracování, která umožňuje detekci narušitelů, identifikaci osob, sledování a počítání objektů, vykreslování cest, vymezení sledované oblasti, uplatnění vlastního detektoru aj. Zpracované záznamy je nakonec možné přehrát, stáhnout nebo spolu s výpisem detekovaných narušitelů sdílet na internetu pomocí odkazu.

Abstract

The aim of this thesis is to create an internet application for detection and analysis of violators in the monitored area. Such an application then can be used for automated processing of records from security cameras or other captured videos from the guarded area. The first part of the work is focused on the theory of neural networks for object detection and classification in the image and recognition of people by their face. The next part describes used technologies for application development. The result is a client-server application with the possibility of configuration of processing, which allows detection of violators, identification of persons, object tracking and counting, path drawing, definition of the monitored area, usage of own detector, etc. Processed videos at the end can be played, downloaded or together with a list of detected violators shared on the internet via link.

Klíčová slova

konvoluční neuronové sítě, knn, detekce, klasifikace, rozpoznávání obličeje, sledování objektu, zpracování obrazu, YOLOv3, DeepFace, DeepSORT, internetová aplikace, React, Express.js, Socket.io

Keywords

convolutional neural networks, cnn, detection, classification, face recognition, object tracking, image processing, YOLOv3, DeepFace, DeepSORT, internet application, React, Express.js, Socket.io

Citace

SADÍLEK, Jakub. *Detekce a analýza narušitelů ve sledované oblasti.* Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Tomáš Goldmann

Detekce a analýza narušitelů ve sledované oblasti

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Tomáše Goldmanna. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jakub Sadílek
16. května 2022

Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Tomášovi Goldmannovi za poskytnutí cenných rad, které měly pozitivní dopad na kvalitu této práce.

Obsah

Kapitola 1

Úvod

Náplní této práce je vytvořit nástroj pro snadné zpracování videozáznamu za účelem detekce a analýzy narušitelů ve sledované oblasti s možností konfigurace. Praktickým příkladem využití může být záznam z bezpečnostní kamery, kde neoprávněna osoba vnikne na soukromý pozemek nebo záznam z veřejného prostranství s přímým výhledem na odpadkový koš, ve kterém byl nalezen předmět spojený s trestným činem. Zpětné dohledávání těchto osob pak v obvykle dlouhých záznamech může zabrat nemalé množství času.

Hlavní snahou je tedy tento proces více automatizovat a nabídnout uživateli nástroj, který bude lehce dostupný a snadno použitelný. Takovým nástrojem je klient-server aplikace, kde uživatel nahraje svá data, nakonfiguruje způsob zpracování a po zpracování záznamu mu budou zobrazeny všechny nálezy potencionálních narušitelů. V aplikaci je implementována řada funkcí, které může uživatel v tomto kontextu využít jako je např. vymezení sledované oblasti a zamezení tak detekce objektů, které se nachází mimo ni nebo identifikace osob podle jejich obličeje. Mezi další funkce pak patří sledování objektů, vykreslování stop, které znázorňují cestu odkud daný narušitel přišel, počítání všech nálezů aj. Pro pokročilejší zpracování je pak možné využít možnost pro použití vlastního detektoru.

V následujícím textu kromě teorie jsou taktéž nastíněny i praktické aspekty, se kterými bylo nutné se během vývoje vypořádat. Konkrétně v následující kapitole 2 je popsána problematika zpracování obrazu pomocí neuronových sítí. Kapitola je především zaměřena na detekci a klasifikaci objektů v obraze a rozpoznávání osob podle obličeje. Jsou zde uvedeny i některé modely, které se v této oblasti aktivně používají. Kapitola 3 je zaměřena na vybrané technologie, které byly použity při implementaci klientské a serverové části aplikace, včetně komunikace mezi nimi. V kapitole 4 je uveden návrh celého řešení a výsledná podoba aplikace, jenž je doplněna o některé zajímavé implementační detaily. Dále je v této části popsána implementace serveru a proces zpracování videa. Poslední kapitola 5 je věnována experimentům nad reálnými daty a následnému vyhodnocení celého systému z hlediska výkonnosti.

Kapitola 2

Neuronové sítě pro zpracování obrazu

Tato kapitola uvede čtenáře do oblasti neuronových sítí, kde na začátku jsou vysvětleny základní principy fungování sítě, její struktura a způsob trénování. Jelikož se tato práce zaměřuje na zpracování obrazu, tak následně je část textu věnována konvolučním neuronovým sítím, které se právě k tomuto účelu používají. Poté je nastíněna problematika detekce a klasifikace objektů a jsou blíže specifikovány páteřní sítě a *state-of-the-art* modely, které se v této oblasti uplatňují. Dále jsou zmíněny vybrané metody pro detekci obličejů a v závěru této kapitoly se pak nachází některá řešení pro rozpoznávání osob podle obličeje a sledování objektů.

2.1 Teorie neuronových sítí

Neuronové sítě [1] jsou známou technikou strojového učení inspirovanou fungováním lidského mozku. Základní jednotkou tohoto nervového systému jsou neurony, které jsou navzájem propojeny a jejich účelem je analyzovat a vyhodnocovat data okolo nás. V neuronových sítích jsou elementární výpočetní jednotkou *umělé* neurony, jenž simulují vlastnosti těch reálných. Přestože výpočetní technologie excelují nad člověkem při řešení matematických výpočtů, nelze je tak snadno využít na úlohy, které jsou naopak triviální pro lidský mozek jako může být například rozpoznávání. Tyto úlohy totiž není možné lehce algoritmizovat, proto se neuronové sítě používají zejména tam kde:

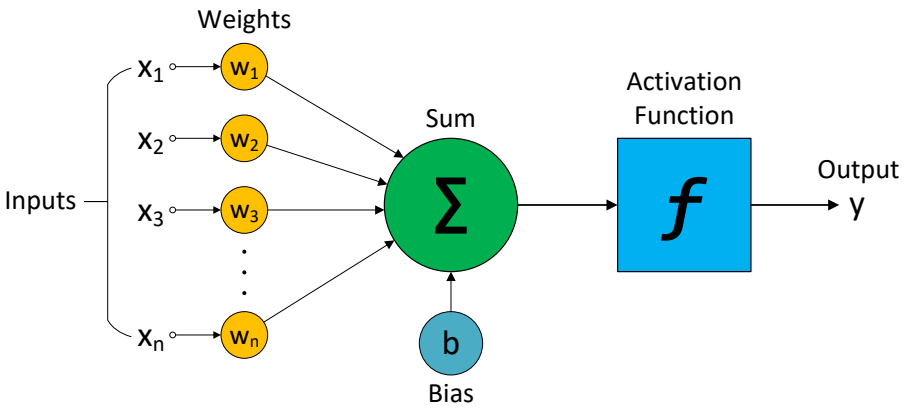
- model je velmi komplexní a náročný předem algoritmizovat,
- v závislosti na datech se chování programu postupem času mění,
- pro danou situaci nejsou předem známé všechny možnosti, které mohou nastat.

2.1.1 Neuron

Základní stavební jednotkou [1] neuronové sítě je neuron, jehož schéma je zobrazeno na obrázku 2.1. Neuron přijímá několik vstupů x, které mají vlastní váhy w. Váhy lze přirovnat ke *„zkušenostem“* a čím je hodnota váhy vyšší, tím je daný vstup pro výpočet důležitější. Proces vyhodnocení probíhá tak, že na každém vstupu se hodnota s váhou vynásobí a jejich následný součet poslouží jako vstup do aktivační (přenosové) funkce, která spočítá výslednou hodnotu. Funkci tohoto modelu je možné popsat následující rovnicí 2.1.

$$f(x) = \delta(\sum_{i=1}^{N}(x_i \cdot w_i) + b), \tag{2.1}$$

kde *f(x)* je výstupní hodnota neuronu, δ je aktivační funkce, N je počet vstupů, x je hodnota vstupu, w je váha vstupu a b je bias neboli posunutí.



Obrázek 2.1: Schéma neuronu s biasem. Převzato z knihy [1].

Účelem **aktivační funkce** je zavést do váženého součtu nelinearitu, která bude produkována na výstup neuronu. Tento krok [19] je důležitý zejména proto, že většina dat z reálného světa má nelineární průběh a cílem je neurony tuto nelineární reprezentaci naučit. V praxi [51] se vyskytuje více druhů těchto funkcí, mezi tradiční patří například sigmoida nebo tanh (viz obrázek 2.2). **Sigmoida** je příkladem logistické funkce, která se používá pro transformaci hodnot do intervalu <0, 1> a využívá se pro vyjádření míry pravděpodobnosti. Název funkce vychází z jejího tvaru do písmene S a je definována vztahem 2.2. Funkce **tanh** neboli hyperbolický tangens lze chápat spíše jako rozšíření sigmoidy, která má podobný tvar, ale produkuje výstup v rozmezí <-1, 1> (viz vzorec 2.3).

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{2.3}$$

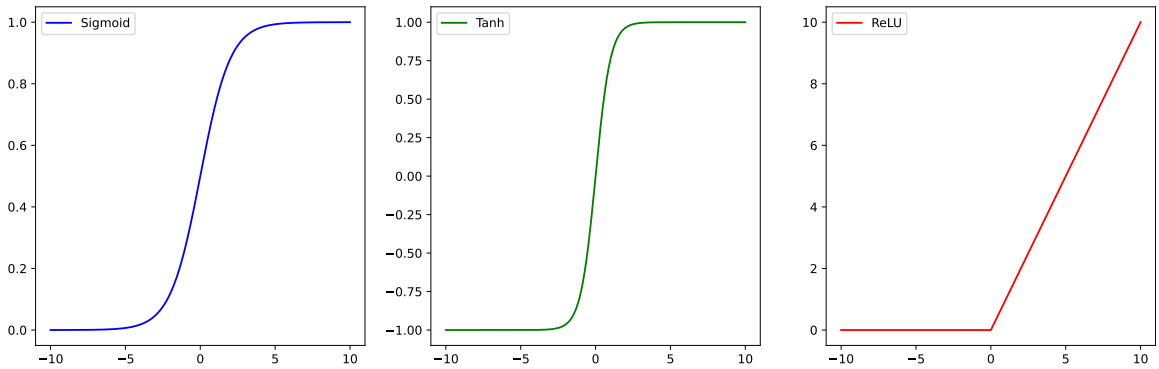
Dále se často používá funkce **softmax**, která je zobecněním logistické funkce do více dimenzí a je definována ve vzorci 2.4. Využívá se především u klasifikačních úloh v poslední vrstvě, jejímž cílem je přiřadit prvek do jedné z více předem definovaných tříd. Funkce přijímá vektor hodnot, který následně převede na vektor hodnot mezi 0 a 1, kde hodnota po součtu je rovná právě 1. Hodnoty tak reprezentují pravděpodobnost příslušnosti prvku k daným třídám.

$$softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{N} e^{x_j}} \tag{2.4}$$

Nejčastěji však používanou *moderní* funkcí je **ReLU** *(Rectified Linear Unit)* [51], která kladné hodnoty propaguje přímo na výstup, zatímco záporné hodnoty kompletně eliminuje na nulu – viz matematická definice 2.5 a její průběh na obrázku 2.2. Tato funkce se stala populární díky svému výkonu v podobě méně operací a menšímu problému s mizejícím gradientem v porovnání s výše zmíněnými funkcemi, čímž zapříčiňuje vysokou míru učenlivosti

modelu. Funkce však s sebou přináší problém, kdy při záporných hodnotách učení stagnuje kvůli nulovému gradientu (tento jev se nazývá *„dying ReLU problem“*). Problém řeší její upravená verze **Leaky ReLU** [51], která při záporné hodnotě propaguje pouze hodnotu velmi blízké nule a neuron tak musí neustále reagovat.

$$ReLU(x) = max(0, x) \tag{2.5}$$



Obrázek 2.2: Graf průběhů aktivačních funkcí, kde vlevo se nachází sigmoida, uprostřed tanh a vpravo ReLU.

2.1.2 Vrstvy a učení neuronové sítě

Stejně jako v lidském mozku, tak i v neuronových sítích jsou neurony mezi sebou propojeny. V neuronové síti jsou neurony koncipovány ve vrstvách, které jsou na sebe napojeny a mohou obsahovat libovolný počet neuronů. První vrstva se nazývá vstupní a definuje rozměry vstupního vektoru, zatímco výstup ze sítě je produkován poslední vrstvou, která se nazývá výstupní. Mezi vstupní a výstupní vrstvou se může nacházet několik skrytých vrstev, jejichž počet určuje hloubku neuronové sítě.

K tomu [28], aby byla síť schopna produkovat správné výsledky, musejí vrstvy implementovat funkci pro **dopřednou propagaci** *(forward propagation)* a k tomu, aby byla schopna se učit, tak implementovat funkci pro **zpětnou propagaci** *(backward propagation)*. Proces zpracování dat potom probíhá tak, že data ze vstupní vrstvy se přenáší přes skryté vrstvy, kde pomocí funkce dopředné propagace a následnému zavedení nelinearity se postupně dostanou až do výstupní vrstvy. Během této propagace se neděje žádné učení a účelem je pouze provést predikci nad vstupními daty a následně reprezentovat výsledek. Zpětná propagace probíhá opačným směrem a jejím účelem je zlepšit přesnost sítě (učit ji) upravením jejích vah. Míra úpravy je provedena v závislosti na hodnotě učícího koeficientu a podle toho, jak moc byl výsledek rozdílný vůči očekávanému výstupu *(ground truth)*. Typy učení se dělí následovně:

- **Učení s učitelem** *(supervised learning)* – v tomto typu učení jsou vstupní data poskytnuta zároveň s očekávaným výstupem pomocí tzv. *labels*.
- **Učení bez učitele** *(unsupervised learning)* – zde jsou poskytnuta pouze vstupní data a síť musí sama zjistit správný výstup bez jakéhokoliv vnějšího zásahu.
- **Posilované učení** *(reinforcement learning)* – tento přístup využívá k ohodnocení korektnosti výstupu tzv. odměny, jenž udávají pouze míru toho, jak byl výsledek

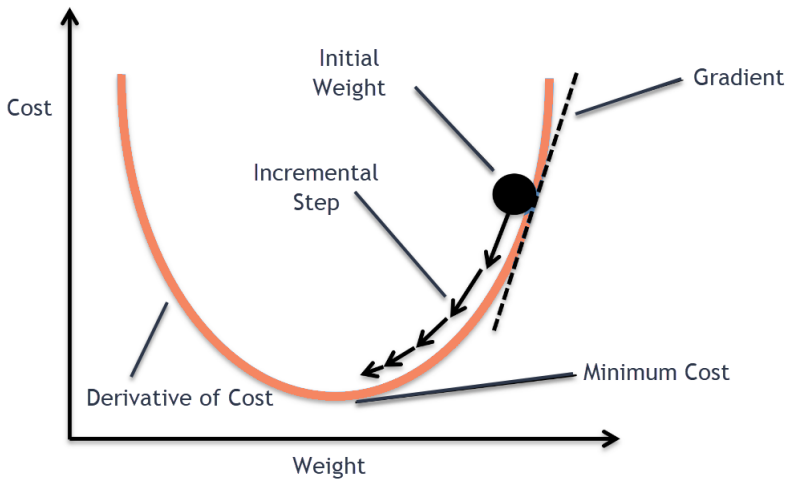
správný. Využívá se například u hraní her, kdy správný tah je ohodnocen kladným skóre a špatný tah záporným.

K zjištění míry [28] toho, jak daleko byl model od správného výsledku se používá tzv. **ztrátová funkce** *(loss function)*. Ačkoliv existuje mnoho druhů těchto funkcí, tak slouží ke stejnému účelu a její výběr se odvíjí podle účelu vyvíjeného modelu. Známou variantou je například funkce střední kvadratické chyby *(mean squared error)*. Úspěšnost modelu je pak definována **nákladovou funkcí** *(cost function)*, která udává chybu nad celým datasetem průměrováním jednotlivých hodnot loss funkcí. Během tréninku je tak cílem dosáhnout minimální hodnoty cost funkce, aby predikce modelu byly co nejpřesnější.

K minimalizaci chyby cost funkce [28] se používají **optimalizátory**. Ty pomocí vah upravují model do lepší formy v závislosti na hodnotách loss funkce. Díky tomu model neustále zlepšuje své predikce a postupně konverguje do ideální podoby. Opět zde existuje spousta algoritmu a mezi nejznámější patří např. gradientní sestup *(gradient descent)*, jehož reprezentaci lze vidět na obrázku 2.3.

Účelem algoritmu pro **gradientní sestup** [28] je dosáhnout lokálního minima pomocí jednotlivých kroků, jejichž velikost je dána hyperparametrem pro nastavení rychlosti učení *(learning rate)*. Tento parametr má významný dopad na průběh učení, kde příliš velký koeficient způsobí velké kroky, které mohou zapříčinit přeskočení minima. Na druhou stranu příliš malé ohodnocení bude vyžadovat velké množství času a výpočetního výkonu na dosažení uspokojivého výsledku.

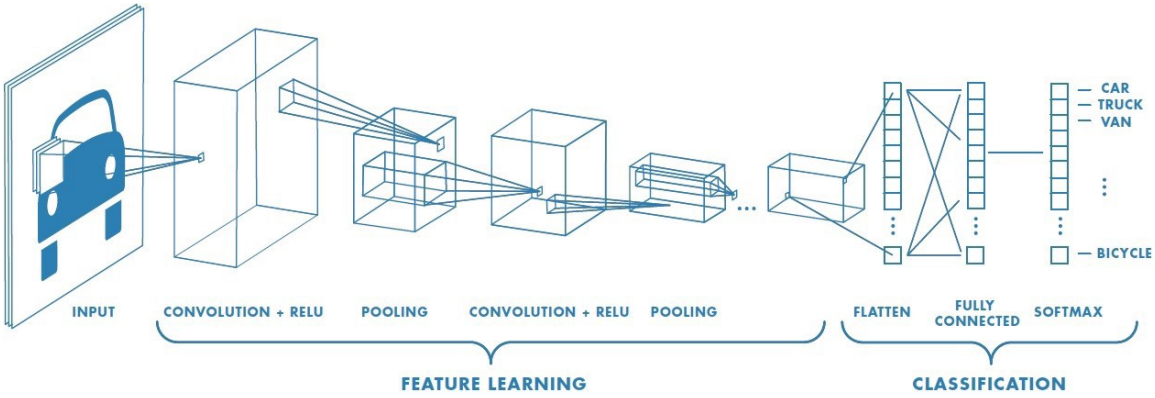
Potom co doběhne zpětná propagace, zahájí se další cyklus učení s novými vzorky přes dopřednou propagaci, kde by tentokrát měl model vykonat o trochu lepší predikci. Tento proces probíhá, dokud není trénování dokončeno a nalezeno minimum. Etapa, v niž jsou právě jednou zpracována všechna data z datasetu se nazývá „epocha“. Epochu pak lze pro trénování rozdělit na menší části datasetu, které se nazývají *„batche“*.



Obrázek 2.3: Znázornění algoritmu pro gradientní sestup, jenž v prvním kroku získá aktuální sklon funkce pomocí derivace prvního řádku v aktuálním bodě. Následně provede krok v opačném směru růstu gradientu pro nalezení lokálního minima. Převzato z článku [28].

2.2 Konvoluční neuronové sítě

Konvoluční neuronové sítě (KNN) [53] jsou typem neuronových sítí, které se používají pro zpracování obrazových dat. Tyto sítě jsou navrženy tak, aby nemusely v obraze počítat s každým pixelem jako s odlišnou informací, ale spíše se naučily rozeznávat známé vzory, stejně tak, jak je to přirozené pro člověka. Obvykle se skládá z konvolučních a pooling vrstev, které provádějí extrakci příznaků. Plně propojené vrstvy, pak provádí klasifikaci a převádí získané informace na výstupní vrstvu sítě – viz následující obrázek 2.4.



Obrázek 2.4: Architektura KNN, kde na vstupní vrstvu je přiveden obrázek, ze kterého jsou následně extrahované příznaky pomocí konvolučních a pooling vrstev. Nakonec jsou příznaky vektorizovány a pomocí sekvence plně propojených vrstev je obrázek klasifikován. Převzato z článku [41].

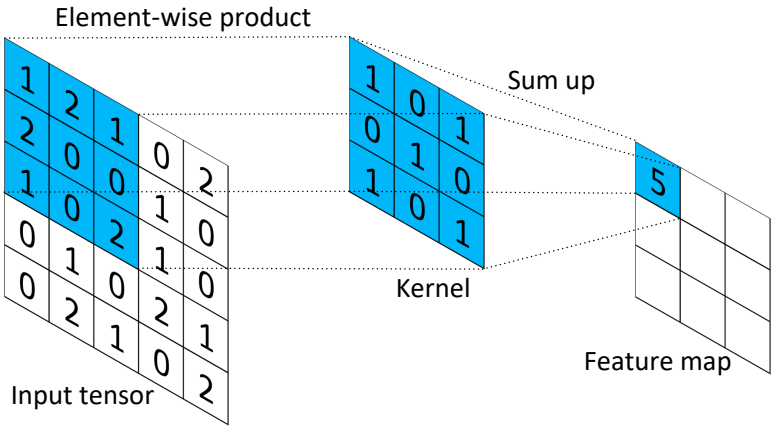
2.2.1 Konvoluční vrstva

Konvoluční vrstva [53] hraje v KNN klíčovou roli, která provádí extrakci příznaků pomocí konvoluce, kde na vstupní data (obraz), reprezentována ve dvourozměrné matici se aplikuje malý filtr *(kernel)*. Ve skutečnosti konvoluční vrstva může obsahovat více těchto filtrů, které extrahují různé vlastnosti a jejich počet pak udává hloubku dané vrstvy. Výsledkem těchto operací je matice nazývaná mapa příznaků *(feature map)*, jejíž hloubka se odvíjí od počtu použitých filtrů.

Každý filtr je složen z trénovatelných vah a také má vlastní hloubku, kterou přejímá od vstupních dat – např. filtr pro zpracování RGB obrázku bude mít hloubku 3. Konvoluce pak probíhá posuvem filtru přes vstupní matici, kde v každém kroku je spočítán jeden prvek výsledné mapy (viz obrázek 2.5). Velikost posuvu mezi kroky se nazývá *stride* a diskrétní 2D konvoluci lze popsat následujícím vztahem 2.6.

$$y(m,n) = \sum_{i=0}^{A-1} \sum_{j=0}^{B-1} f(i,j) \cdot x(m-i, n-j), \tag{2.6}$$

kde y je výstupní matice, f je použitý filtr, x je vstupní matice, A a B jsou rozměry filtru, m a n jsou souřadnice aktuálně počítaného prvku ve výstupní matici. Kvůli pohybu filtru [53] ve vstupní matici však není možné, aby střed filtru pokryl i hraniční prvky této matice, což má za následek zmenšení výstupní mapy. K řešení tohoto problému se používá tzv. nulová výplň *(zero-padding)*, která okraje vstupní matice vyplní nulami a tím umožní střed filtru



Obrázek 2.5: Proces získání mapy příznaků pomocí konvoluce, kde aplikovaný filtr postupnými posuvy zpracuje celou vstupní matici a produkuje mapu příznaků. Převzato z článku [53].

umístit i na její nejkrajnější prvky. Díky této technice pak zůstane velikost výstupní mapy vůči vstupu zachována, a je tak umožněno lépe aplikovat více filtrů. Nejvýznamnější dopad na proces trénování má však správná definice filtrů, které se nejvíce hodí na řešení daného úkolu. Filtry jsou pak jediné parametry v této vrstvě, které se během procesu tréninku učí a zlepšují predikci sítě. Na druhou stranu velikost filtrů, jejich počet, výplň a stride jsou hyperparametry, které se nastavují před samotným procesem.

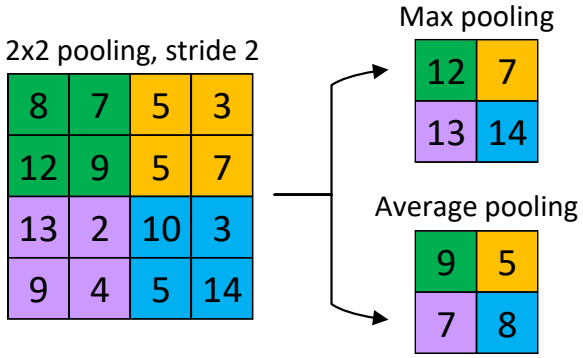
2.2.2 Pooling vrstva

Pooling vrstva [53] poskytuje typickou operaci převzorkování (podvzorkování), která snižuje velikost dimenzí mapy příznaků. Účelem je zavedení translační invariance v případě menších posunů a snížení tak počtu trénovatelných parametrů, které se nachází v konvolučních vrstvách. Hloubka ovšem zůstává nezměněna. Důležité však při tomto procesu je nějakým způsobem zachovat nejpodstatnější informace. Pooling vrstva sama o sobě neobsahuje žádné trénovatelné parametry, kdežto obsahuje hyperparametry pro nastavení stride, výplně a velikosti filtru obdobně jako u konvoluční vrstvy.

Nejčastěji používaná pooling vrstva bývá tzv. *max-pooling* nebo *average-pooling* vrstva s rozměry filtru 2×2 a nastaveném stride na 2 (viz obrázek 2.6). Max-pooling při filtraci příznaků propaguje prvek s nejvyšší hodnotou a ostatní eliminuje. Average-pooling vrstva zase produkuje průměrnou hodnotu ze všech hodnot z aktuálního překryvu. Filtr se následně posune o velikost stride dále.

2.2.3 Plně propojená vrstva

Plně propojená vrstva [53] se běžně vyskytuje u konce celého procesu, kdy mapa příznaků je již zploštělá z konečné konvoluční či pooling vrstvy do jednorozměrného pole (vektoru). V této vrstvě je každý vstup propojen s každým výstupem pomocí naučitelných vah a biasu, které jsou následně transformovány aktivační funkcí (viz předchozí sekce o umělém neuronu 2.1.1). Typicky jsou tak extrahované vzory předešlými konvolučními a pooling vrstvami namapovány množinou plně propojených vrstev na výstupní vrstvu sítě. Konečná plně propojená vrstva u klasifikačních úloh má pak počet výstupních uzlů rovný počtu klasifikačních tříd.



Obrázek 2.6: Ukázka podvzorkování mapy příznaků pomocí *max-pooling* a *average-pooling* filtrů s rozměry 2×2 a nastaveném stride na 2. Převzato z článku [53].

2.3 Detekce a klasifikace objektů

Detekce a klasifikace patří mezi nejčastější úlohy řešených pomocí neuronových sítí a spadá do základů počítačového vidění. Raná detekce [55] byla postavena na ručně vytvořených detektorech příznaků (viz sekce 2.4) jako je Viola-Jones, histogram orientovaných gradientů aj. Přestože tyto metody fungovaly, tak byly poměrně pomalé a nepřesné, což se změnilo při znovu-uvedení KNN a hlubokého učení. Detekce a klasifikace objektů pak nachází své uplatnění např. u řízení autonomních vozidel, rozpoznávání identit osob pro zvýšení bezpečnosti či k lékařským účelům.

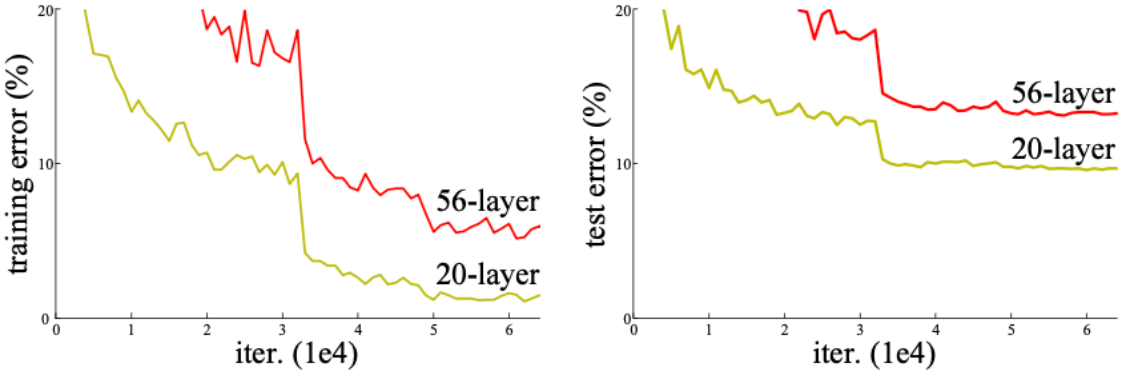
2.3.1 Páteřní architektury

Páteřní sítě *(backbone)* [55] jsou nejdůležitější komponentou pro detekci objektů. Tyto sítě mají za úkol ze vstupního obrazu extrahovat příznaky pro jejich další využití v modelu v podobě mapy příznaků. Níže jsou uvedeny některé milníkové a často používané architektury v moderních detektorech.

Reziduální sítě

Běžný postup [18, 45] v návrhu sítě pro řešení stále komplexnějších úloh vede k dalšímu přidávání vrstev a tím zvyšování hloubky sítě. Intuice spočívá v tom, že další vrstvy se postupně učí složitějším rysům, a tak hlubší síť poskytne lepší přesnost a výkon. Ukázalo se ovšem, že u tradičního modelu KNN existuje maximální práh hloubky, jak je znázorněno na obrázku 2.7.

Tento problém [18, 22] velmi hlubokých sítí se podařilo zmírnit zavedením reziduálních sítí (ResNets), které jsou tvořené speciálními vrstvami nazývané reziduální bloky (viz obrázek 2.8). Nejvýznamnější modifikací těchto bloků jsou tzv. zkratky, které umožňují přeskočit jednu či více vrstev. Výhoda spočívá v tom, že pokud nějaká vrstva zhoršuje výkon modelu, tak potom bude přeskočena (tj. v ideálním případě bude $F(x)$ na obr. 2.8 naučené na 0). Zkratka [31, 45] se skládá z tzv. „funkce mapování identity“, která se dělí na identitu a projekci. **Identita** se používá v případě, kdy data na vstupu a výstupu bloku mají stejnou dimenzi, a je tak možné data pouze předat bez jakékoliv modifikace. Vztah pro mapování identity je definován následujícím vzorcem 2.7.

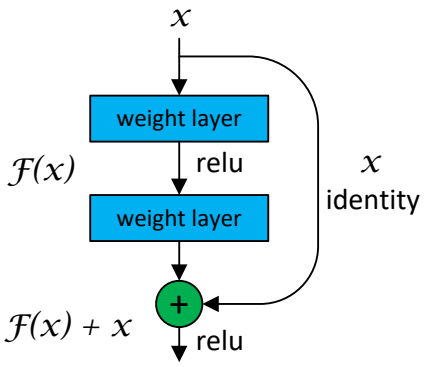


Obrázek 2.7: Znázornění vztahu mezi hloubkou sítě a jejím výkonem, kde hlubší síť vykazuje horší přesnost. Tento jev může být způsoben optimalizační funkcí, inicializací sítě nebo zejména problémem mizejícího (či explodujícího) gradientu. Převzato z článku [22].

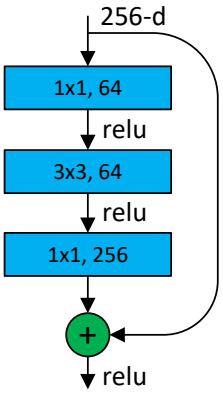
$$y = \mathcal{F}(x, \{W_i\}) + x, \tag{2.7}$$

kde y je výstup, x je vstup, $\mathcal{F}(x, \{W_i\})$ představuje reziduální mapování, jenž se má naučit, W_i označuje váhu i-té vrstvy v daném bloku. **Projekce** [31] se naopak používá v případě, kdy dimenze výstupu neodpovídá vstupu a je tak nutné data přizpůsobit pomocí konvoluce. Vzorec je pak modifikován o lineární projekci W_s ve vztahu 2.8.

$$y = \mathcal{F}(x, \{W_i\}) + W_s \cdot x \tag{2.8}$$



Obrázek 2.8: Architektura reziduálního bloku s dvěma vrstvami, kde $F(x)$ je trénovatelné reziduální mapování a x jsou vstupní data. Převzato z článku [45].



Obrázek 2.9: Architektura bottleneck bloku s třemi vrstvami použité v ResNet-50/101/152. Převzato z článku [22].

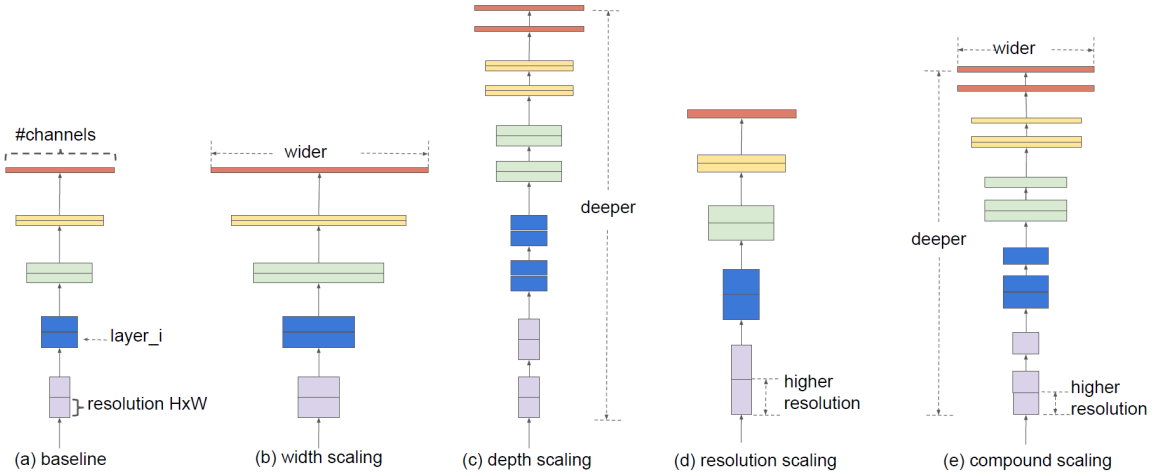
ResNet[1] je rodina reziduálních sítí jako jsou ResNet-18, ResNet-34, ResNet-101 atd., kde číslo na konci udává hloubku sítě. Významnou backbone architekturou [22] je **ResNet-50**, která obsahuje tzv. bottleneck bloky (viz obrázek 2.9). Bottleneck blok využívá 3 konvoluční vrstvy, kde vrstvy s rozměry 1×1 jsou pouze zodpovědné za zmenšení a následné

[1] ResNet: https://github.com/tensorflow/tpu/tree/master/models/official/resnet

zvětšení (obnovení) velikosti dat. V tomto bloku se používá jako zkratka mapování identity, což má za následek snížení času potřebného k natrénování, avšak přesnost sítě zůstává podobná. ResNet-50 je vylepšený model ResNet-34, který nahrazuje 2-vrstvé bloky za 3-vrstvé a zvyšuje tím tak svoji hloubku.

EfficientNet

EfficientNet[2] [34] je rodina sítí jenž se snaží zlepšit přesnost a efektivitu modelu pomocí škálování. Existují celkem tři škálovatelné parametry modelu KNN: hloubka, šířka a rozlišení (viz obrázek 2.10). Hloubka, jak již bylo dříve zmíněno, je dána počtem použitých vrstev, zatímco šířka sítě může být definována např. počtem kanálů v konvoluční vrstvě. Rozlišení pak jednoduše udává rozlišení vstupního obrazu, který je přiveden na vstup sítě.

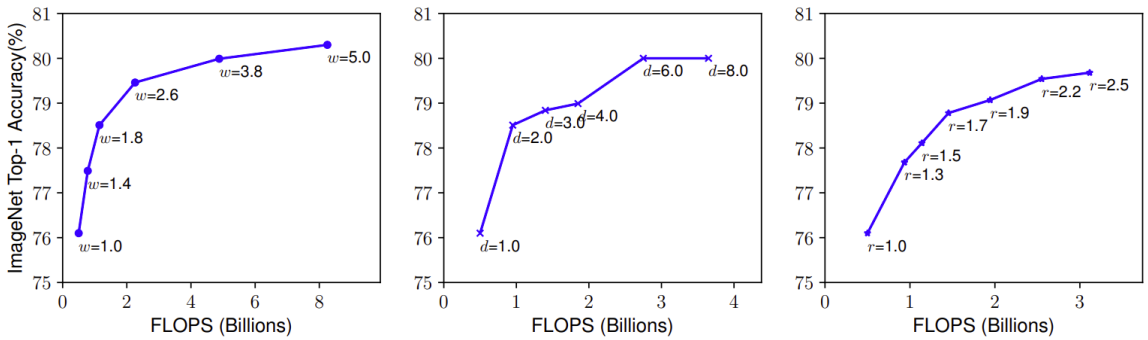


Obrázek 2.10: Škálování modelu KNN. a) je příklad základního modelu, b)-d) je konvenční škálování, které zvětšuje pouze jedenu dimenzi sítě, e) je model vytvořený pomocí složeného škálování, který jednotně škáluje všechny dimenze pevně daným poměrem. Převzato z článku [50].

Škálování hloubky [50] je nejběžnější způsob, jak dosáhnout lepší přesnosti. Například ResNet může být škálován od ResNet-18 přidáváním více vrstev až po ResNet-200. Ovšem kvůli běžnému problému s mizejícím gradientem aj. nelze zacházet neustále hlouběji. Pokud [34] je záměrem udržet model malý a vyhnout se tak problémům provázející hluboké sítě, tak se běžně využívá **škálování do šířky**, kde širší sítě jsou schopny zachytit i méně výrazné vzory a jsou také lépe trénovatelné. Bohužel i sebevětší šířka neposkytne uspokojující výsledky, jelikož takové modely i když jsou velmi dobré k naučení a následnému zapamatování, tak nejsou schopny generalizace *(overfitting)*, kdežto hlubší modely díky svým vrstvám tak poskytují abstrakci na více úrovních. Pro **škálování rozlišení** platí, že s vyšším rozlišením vstupního obrázku je teoreticky možné zachytit více jemných vzorů. Přirozeně tento přístup ale nefunguje lineárně a zisk přesnosti se po chvíli začne velmi rychle snižovat např. při zvednutí rozlišení z 500×500 na 560×560 nepřinese takový rozdíl jako z 220×220 na 280×280. Běžně používané rozlišení začíná na 224×224 a mezi vysoké rozlišení patří např. 600×600. Nejčastěji se však využívá rozlišení 300×300.

[2] EfficientNet: https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet

Zvyšování [34, 50] jakéhokoliv rozměru sice zlepšuje přesnost, ale u větších modelů pak dochází k saturaci (viz obrázek 2.11). V praxi je proto nutné škálovat ve všech dimenzích současně a najít tak mezi nimi rovnováhu. Existuje mnoho postupů, které poskytují obecný návod, jakým způsobem toto škálování provádět, ovšem každý model vyžaduje jiný přístup, a tak je zapotřebí tuto činnost provádět manuálně. Škálování však nemění operace vrstev, takže k zajištění vysoké přesnosti je především důležité mít dobrý základní model sítě *(baseline)*, na kterém následně toto škálování provádět. EfficientNet nabízí základní optimalizovaný model pro klasifikaci EfficientNet-B0 a metodu k jeho jednotnému škálování pomocí složeného koeficientu. Ostatní modely Efficient-B1 až B7 jsou pak získané jeho dalším škálováním.



Obrázek 2.11: Škálování baseline modelu různými koeficienty. Z naměřených hodnot ve všech parametrech je znatelný z počátku velký nárůst přesnosti sítě, který při vyšších hodnotách začíná saturovat dosáhnutím svých limitů, zatímco nároky na výpočetní výkon déle rostou. Převzato z článku [50].

2.3.2 Detektory

Úlohou detektorů [48] je predikovat objekt v obraze a zároveň ho klasifikovat. Existují dva hlavní typy *state-of-the-art* detektorů, prvním jsou dvoustupňové *(two-stage)*, které v první fázi používají metody ke zjištění oblastí zájmu a následně v druhé fázi tyto oblasti zasílají skrz *„pipeline“* ke klasifikaci a regresi bounding boxů. Tento typ modelů disponuje vyšší přesností, za cenu delší doby zpracování. Druhým typem jsou jednostupňové *(one-stage)* detektory, které naopak k detekci přistupují jako k jednoduchému problému regrese tím, že se ze vstupního obrazu naučí pravděpodobnosti všech tříd a souřadnic bounding boxů. Tyto modely tak dosahují nižší přesnosti, ale jsou mnohem rychlejší a umožňují provádět detekci v reálném čase, jako je např. SSD[3], RetinaNet[4], EfficientDet[5] aj.

Regionální konvoluční neuronové sítě

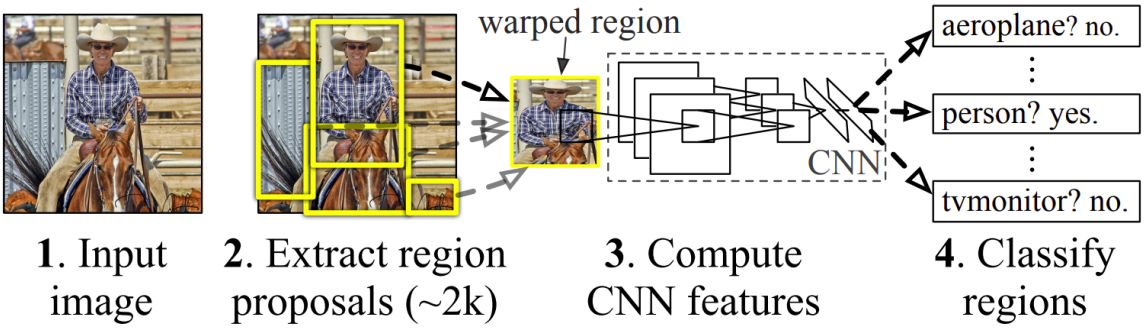
Regionální konvoluční sítě (RCNN) je soubor *two-stage* modelů pro detekci a klasifikaci v obraze, jejichž úkolem je dané objekty v obraze lokalizovat pomocí bounding boxů a přiřadit je do odpovídajících tříd. Jako první v této skupině vznikla stejnojmenná architektura **RCNN**[6] [15], která proces detekce dělí na tři hlavní navazující kroky (viz obrázek 2.12):

[3]SSD: https://github.com/amdegroot/ssd.pytorch
[4]RetinaNet: https://github.com/fizyr/keras-retinanet
[5]EfficientDet: https://github.com/xuannianz/EfficientDet
[6]RCNN: https://github.com/rbgirshick/rcnn

1. **Generování oblastí zájmu** – V prvním kroku jsou v obraze extrahovány kandidátní regiony, které mohou obsahovat hledané objekty. Běžně se jich může vyskytovat 2000 a více. Pro tento účel se používají algoritmy jako *Selective Search*, *Edge Boxes* aj.

2. **Extrakce příznaků** – V tomto kroku je z každého regionu extrahován vektor příznaků, který má následně velký podíl pro správnou predikci objektu. RCNN v tomto kroku používá KNN, i když dříve se k tomu účelu využívali algoritmy jako HOG aj.

3. **Klasifikace** – V poslední části je každý region klasifikován podle jeho vektoru příznaků. K tomu se zde využívají SVM *(Support Vector Machines)* klasifikátory, kde na každou třídu je naučen jeden klasifikátor. Cílem je tak určit predikci na přítomnost objektu v daném regionu.

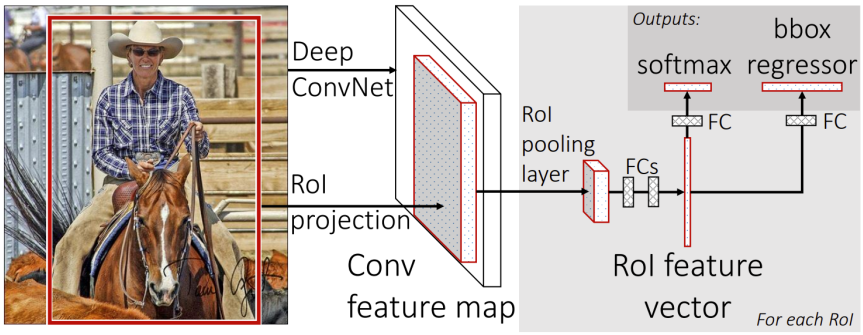


Obrázek 2.12: Schéma architektury RCNN, kde ze vstupního obrázku (1) je jsou extrahovány regiony (2), ze kterých jsou vypočteny vektory příznaků použitím KNN (3). Získané vektory jsou následně použity ke klasifikaci pomocí SVM klasifikátorů (4). Převzato z článku [15].

Model RCNN [13] však obsahuje několik problému, které mohou znemožňovat jeho nasazení v praxi. Jedním z problémů je klasifikace velkého počtu navrhovaných regionů, které znatelně prodlužují čas sítě potřebný pro zpracování obrazu a jejího natrénování. Navíc v mnoha případech se regiony překrývají a dochází tak k opakovaným výpočtům. Další nevýhodu přináší samotné pevné algoritmy pro extrakci regionů jako *Selective Search*, jelikož v této fázi neprobíhá žádné učení, a tato fáze může tedy vest ke generování nevhodných kandidátů. Řešení těchto nedostatků poskytla pak nová architektura Fast-RCNN.

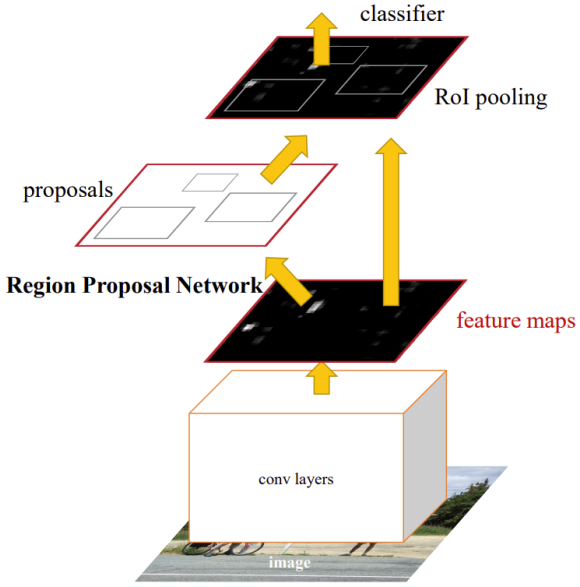
Architektura **Fast-RCNN**[7] [13, 14] poskytuje podobný přístup jako předešlá architektura RCNN, ale nabízí podstatně větší rychlost a snižuje její nároky. Tato architektura taktéž z obrázku generuje kandidátní regiony, avšak je již postupně nepředává do KNN, ale místo toho je ji předána celá množina regionů spolu se vstupním obrázkem, ze kterého je poté vygenerována mapa příznaků (viz obrázek 2.13). V tomto důsledku již nedochází k redundantním výpočtům a každá část obrázku je tak zpracována KNN pouze jednou. Následně jsou identifikovány regiony, pomocí region of interest (RoI) pooling vrstvy, která každý kandidátní region převede do mapy příznaků s pevnou velikostí. Následně jsou mapy regionů převedeny na vektory pomocí sekvence plně propojených vrstev. Výstup sítě je produkován dvěma oddělenými vrstvami, kde softmax vrstva predikuje vektor pravděpodobností tříd daného regionu a regresní vrstvou, která ve vektoru definuje koordináty příslušného bounding boxu.

[7] Fast-RCNN: https://github.com/rbgirshick/fast-rcnn



Obrázek 2.13: Schéma architektury Fast-RCNN, kde vstupní obrázek a regiony zájmu (RoI) jsou přivedeny na vstup KNN. Každý RoI je sdružen do mapy příznaků s pevnou velikostí a poté převeden na vektor pomocí plně propojených vrstev. Výstupní vrstvy sítě produkují vektor pravděpodobností (softmax) a vektor offsetů bounding boxu (regressor). Převzato z článku [14].

Poslední z rodiny RCNN je architektura **Faster-RCNN**[8], která opět vylepšuje svého předchůdce. Architektura Fast-RCNN disponovala nepříliš efektivní metodou generování kandidátních regionů pomocí pevných algoritmů, a tato část tak tvořila „úzké hrdlo“ celého systému, což ovlivňovalo její výkon. Tvůrci tedy přišli s novým způsobem, který nahrazuje ten starý a umožňuje síti se tyto kandidátní regiony naučit.

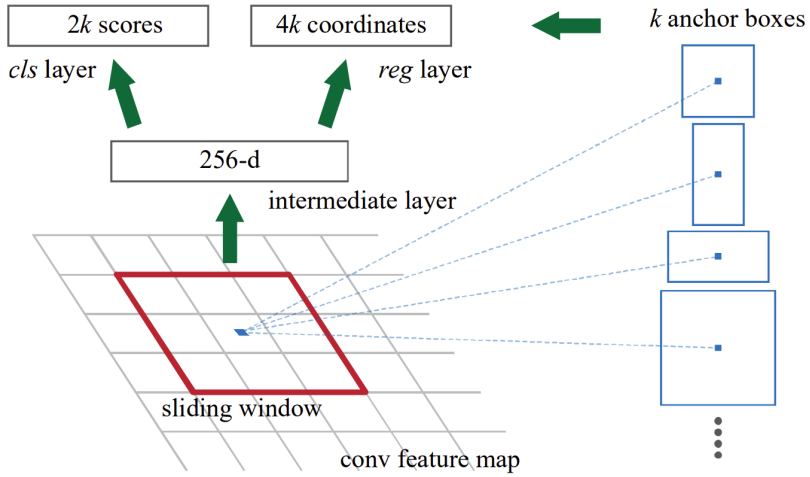


Obrázek 2.14: Architektura Faster-RCNN, kde pomocí KNN je ze vstupního obrázku získána mapa příznaků, která slouží jako vstup do RPN a RoI pooling vrstvy. RPN z mapy extrahuje kandidátní regiony a také je předá RoI vrstvě, která díky nim a disponující mapě příznaků data dále zpracuje, jak je to typické v architektuře Fast-RCNN. Převzato z článku [39].

[8] Faster-RCNN: https://github.com/rbgirshick/py-faster-rcnn

Faster-RCNN [2, 39] lze rozdělit na dvě části, kde první část je hluboká konvoluční síť tzv. Region Proposal Network (RPN), která je zodpovědná za generování kandidátních regionů a druhá část je klasický Fast-RCNN detektor, jenž tyto regiony využívá. Tento přístup nejen zkracuje dobu návrhu regionů z 2 sekund na 10 ms na snímek, ale také umožňuje během této fáze sdílet konvoluční vrstvy s fázemi detekce z Fast-RCNN, což celkově zlepšuje predikci příznaků.

Celý proces [2, 39] začíná vstupním obrázkem přivedeným do KNN *(backbone)*, který z něj získá mapu příznaků, jak je znázorněno na obrázku 2.14. Mapa slouží jako vstup do RPN (viz obrázek 2.15), která nad ní provádí konvoluci pomocí malého $n \times n$ okna (typicky 3×3), kde pro každou pozici okna je predikováno několik kandidátních regionů. Regiony jsou reprezentovány k kotvami *(anchors)*, přičemž tyto kotvy nejsou finální a budou dále filtrovány podle jejich *skóre*. Každá kotva má různou velikost a poměr stran, díky těmto kombinacím je možné detekovat objekty v různých velikostech a přesto zachovat okno i vstupní obrázek v jednom měřítku. Každá oblast okna je tak s kandidátními regiony namapována do vektoru příznaků, který je dále poslán do dvou oddělených plně propojených vrstev pro klasifikaci (cls) a regresi (reg). Klasifikační vrstva udává pouze pravděpodobnost s jakou se v daném okně vyskytuje hledaný objekt pomocí *objectness skóre* bez toho, aniž by prováděla jeho klasifikaci do tříd. Regresní vrstva zase určuje regresní koeficienty, které se používají pro určení bounding boxu daného regionu. Získané regiony jsou pak předány spolu se vstupním obrázkem do RoI pooling vrstvy a následně zpracovány stejným způsobem jako v architektuře Fast-RCNN.



Obrázek 2.15: Způsob zpracování mapy příznaků v RPN. Převzato z článku [39].

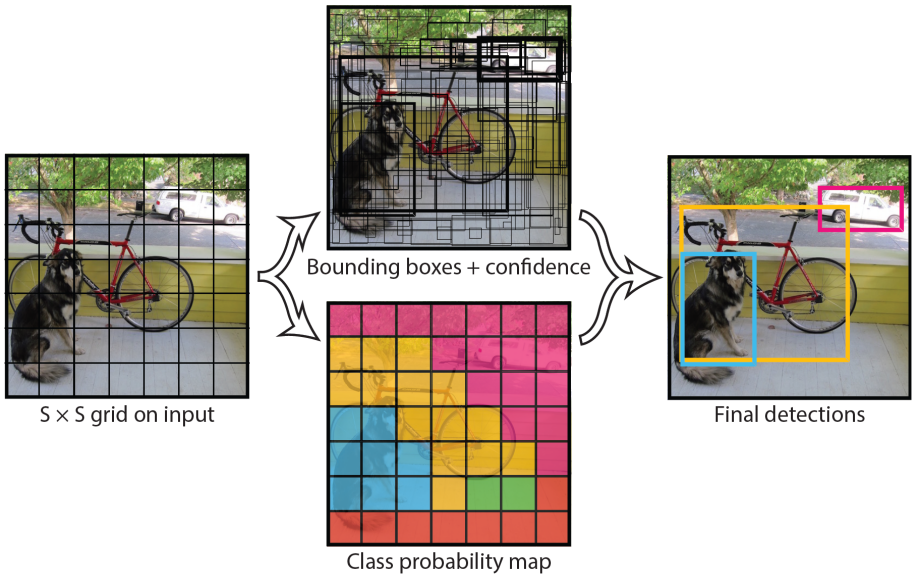
You Only Look Once

You Only Look Once[9] (YOLO) je one-stage algoritmus založený na regresi, který namísto hledání oblastí zájmu přímo predikuje třídy a jejich bounding boxy v jednom běhu a umožňuje tak provádět detekci v reálném čase. Existuje [27] více variant tohoto algoritmu, ovšem nejčastěji se lze setkat s verzí YOLOv3.

Algoritmus [27, 38] pracuje tak, že nejdříve vstupní obrázek rozdělí do pravidelné mřížky o velikosti $S \times S$, za účelem najít objekty, které se zde nachází (viz obrázek 2.16). Pokud se střed objektu bude nacházet uvnitř některé buňky v mřížce, bude tato buňka zodpovědná za

[9]YOLO: https://pjreddie.com/darknet/yolo

jeho detekci. Každá buňka pak predikuje B bounding boxů a jejich odpovídající *confidence skóre*. Každý box je tak reprezentován pěticí (x, y, w, h, cs), kde (x, y) jsou souřadnice středu boxu, šířka w a výška h jsou predikované rozměry a cs je přiřazené skóre. Confidence skóre reflektuje, jak moc si je model jistý, že daný box obsahuje objekt a také jak moc si myslí, že je přesné jeho ohraničení. Pokud se v dané oblasti žádný objekt nenachází, mělo by toto skóre být nulové. V opačném případě je skóre rovno hodnotě tzv. průniku nad sjednocením *(intersection over union)*, což je technika, která vyjadřuje jak moc se predikované a reálné boxy překrývají. Tedy v případě hodnoty 1 je predikovaný box stejný jako ten skutečný a díky tomu pak lze eliminovat nepřesné rámečky např. pomocí techniky Non-Maximum Suppression. Kromě toho každá buňka mřížky také predikuje množinu C pravděpodobností pro přiřazení do konkrétních tříd. Tyto pravděpodobnosti jsou predikované na danou buňku, bez ohledu na počet bounding boxů B. Výsledné predikce jsou nakonec zakódované v tenzoru o velikosti $S \times S \times (B \cdot 5 + C)$.



Obrázek 2.16: Model YOLO, kde v tomto případě je vstupní obrázek rozdělen do pravidelné mřížky 7×7. Každá buňka mřížky predikuje B bounding boxů, jejich odpovídající *confidence skóre* a C třídních pravděpodobností. Vpravo lze nakonec vidět výsledné predikce. Převzato z článku [38].

2.4 Detekce obličeje

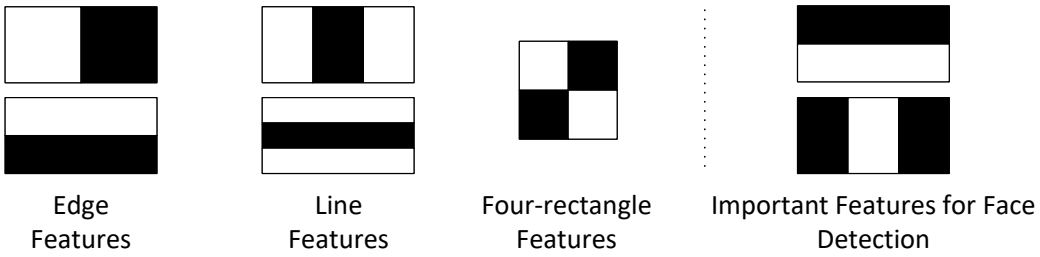
První krok pro zpracování obličeje je jeho lokalizace. V této situaci existuje více metod, které je zde možné využít. Jedním z přístupů je využití expertních znalostí, kde se využívají charakteristiky lidského obličeje jako je jeho kompozice či barvy. Dalším a velmi častým přístupem je použití neuronových sítí a přistupovat k tomuto problému jako k detekci objektů. Tyto přístupy také lze kombinovat. Stěžejní vliv na detekci má fakt, že obličeje jsou obecně nasnímány v různých pozicích, úhlech, výrazu či prostředí a světelných podmínkách. Aplikaci detekčních algoritmů proto běžně předchází fáze předzpracování, která zahrnuje transformaci barev, filtraci, převzorkování aj. za účelem zvýšení účinnosti detekce.

2.4.1 Viola-Jones algoritmus

Viola-Jones [17] je framework pro detekci objektů publikovaný v roce 2001 a ačkoliv se jedná spíše o zastaralou metodu, tak vykazuje pozoruhodné výsledky při detekci obličejů. Rámec umožňuje také zpracování v reálném čase, přestože je velmi pomalý na natrénování. Algoritmus pracuje tak, že nejdříve vstupní obraz konvertuje do šedotónového odstínu, protože pak obsahuje méně dat ke zpracování a lze tak s ním lépe pracovat. Obličej se na obrázku hledá v klouzavém okénku, které se postupně posunuje po obrázku, kde velikost okénka a posunu lze libovolně nastavit. Ovšem menšími posuny je možné dosáhnout větší přesnosti lokalizace. Algoritmus při zpracování obsahuje čtyři hlavní kroky:

1. Detekce Haarových příznaků
2. Vytvoření integrálního obrazu
3. Spuštění tréninku AdaBoost
4. Sestavení kaskády klasifikátorů

Pro detekci příznaků [17, 20] se v klouzavém okénku používají celkem tři typy **Haarových příznaků** (viz obrázek 2.17), které jsou převzaty z konceptu Haarových vlnek. Tyto příznaky představují funkce se světlou a tmavou oblastí, které se používají pro detekci rysů. Výsledek je prezentován jedinou hodnotou jako součet hodnot intenzit pixelů v tmavé oblasti mínus součet intenzit pixelů ve světlé oblasti. Hodnota bude nulová v případě, kdy všechny pixely mají stejnou hodnotu a oblast tak nenese žádnou informaci. Na obličeji se ve skutečnosti vyskytují velmi rozdílné oblasti, které poskytnou vysokou hodnotu. Takový přechod z tmavé oblasti do světlé může definovat okraj obočí, či lesklá linie mezi tmavými oblastmi může indikovat nos. Aplikací všech funkcí na různé pozice může nakonec určit zda obraz obsahuje lidskou tvář či nikoliv.

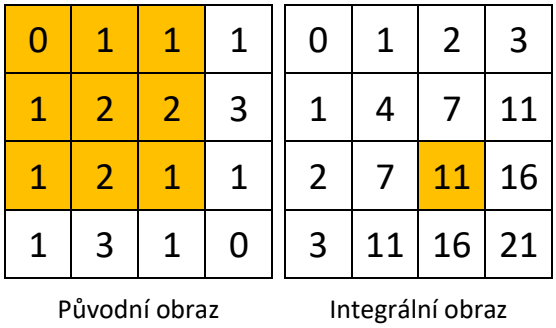


Obrázek 2.17: Haarovi příznaky pro detekci hran, linií a čtyřstranného příznaku. Převzato z článku [20].

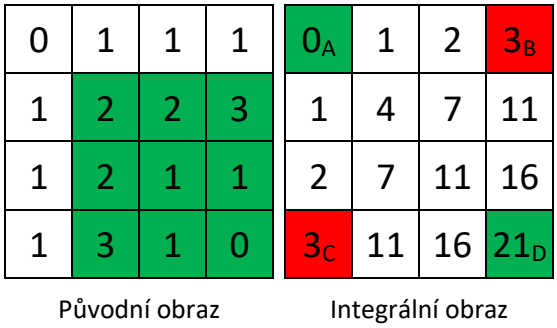
Ve skutečnosti [17, 20] výpočet hodnoty může být velmi náročný v případě rozlehlého příznaku, který zahrnuje velké množství pixelů. V tomto případě se používá technika **integrálního obrazu**, která dovoluje tyto výpočty provádět snadno a efektivně, čímž umožní rychle rozpoznat, zda příznak odpovídá jeho kritériím. Výpočet integrálního obrazu je realizován tak, že hodnota konkrétního pixelu je dána součtem všech pixelů v oblasti od jeho pozice k levému hornímu rohu, jak je znázorněno na obrázku 2.18. Výčet pak výsledné hodnoty z oblasti lze realizovat pouze pomocí čtyř hodnot z integrálního obrazu podle následujícího vzorce 2.9 s referencí na obrázek 2.19.

$$oblast = I(D) + I(A) - I(B) - I(C), \tag{2.9}$$

kde $I(D)$ je hodnota intenzity pixelu v integrálním obraze uvnitř oblasti v pravém dolním rohu, $I(A)$ je hodnota mimo oblast v levém horním rohu, $I(B)$ a $I(C)$ jsou hodnoty pixelů mimo oblast v pravém horním a levém dolním rohu.



Obrázek 2.18: Ukázka výpočtu pixelu v integrálním obraze, jehož hodnota je dána součtem pixelů z původního obrazu z oblasti od jeho aktuálního umístění směrem k levému hornímu rohu. Převzato z článku [17].



Obrázek 2.19: Výpočet hodnoty oblasti z integrálního obrazu podle vzorce 2.9, kde $21 + 0 - 3 - 3 = 15$, je to stejné jako $2 + 2 + 3 + 2 + 1 + 1 + 3 + 1 + 0 = 15$. Převzato z článku [17].

Jako další krok [17] v rámci Viola-Jones se používá algoritmus strojového učení pro zlepšení výkonnosti známý jako **AdaBoost** *(Adaptive Boosting)*. Pokud bude použita velikost detekčního okénka 24×24, tak počet přítomných příznaků potom může dosahovat až 160000, kdežto jen pár z nich je pro detekci obličeje opravdu důležitých. AdaBoost proto *rozhoduje* o tom, jaké typy a velikosti Haarových příznaků budou součástí výsledného klasifikátoru. Jinak řečeno každý Haarův příznak reprezentuje slabý klasifikátor a AdaBoost pomocí těchto slabých klasifikátorů má za úkol sestavit jeden silný klasifikátor. Vyhodnocení výkonnosti každého klasifikátoru se provádí na označených datech pro trénování. Po vyhodnocení se vyberou klasifikátory, které vykazovaly nejvyšší odezvu nad obrázky s tvářemi, zatímco obrázky bez tváří vyhodnocovaly negativně. Těmto klasifikátorům se přiřadí váhy podle jejich výkonnosti a budou nakonec součástí silného klasifikátoru.

Přestože AdaBoost [17, 20] výrazně napomáhá výběru relevantních příznaků, stále obsahuje mnoho výpočtů na to, aby byly všechny aplikovány v každé pozici okna. K tomuto účelu se využívá **kaskáda klasifikátorů**, která zvyšuje rychlost tím, že se snaží, co nejrychleji zahodit výpočet nad snímky bez tváře. Proces probíhá tak, že nad oknem je použit nejlepší disponující silný klasifikátor (např. identifikace hřbetu nosu), který rozhodne o přítomnosti příznaku. Pokud příznak v okně není dostupný, tak je okno zahozeno a již není dále zpracováno. V případě, kdy bude příznak detekován, tak bude zpracován druhým nejlepším klasifikátorem, který má opět možnost celé okno zahodit. Postupně je takto zpracována celá kaskáda vybraných klasifikátorů a pokud se dané okno dostane až na její konec, znamená to, že všechny klasifikátory potvrdily shodu a na snímku je detekován obličej.

2.4.2 Histogram orientovaných gradientů

Histogram orientovaných gradientů (HOG) je algoritmus pro detekci objektů, který se dá aplikovat i na lidský obličej. V prvním kroku [46] algoritmu jsou vypočteny hodnoty gradientů pro každý pixel ve vstupním obraze, jenž udávají změnu intenzity na osách x a y. Ve směru x se gradient vypočítá odečtením hodnoty intenzity levého souseda od hodnoty intenzity pravého souseda. Gradient ve směru y odečtením intenzity horního pixelu od

hodnoty spodního pixelu. Hodnota vyjde vysoká v případě, kdy dojde k velkému rozdílu intenzit (např. na okrajích objektů). Pro tento výpočet se používají 1D masky, které jsou znázorněny v definici 2.10.

$$g_x = \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \qquad g_y = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix} \tag{2.10}$$

Dalším krokem [46] je určení velikosti a směru výsledného gradientu pro každý pixel. Velikost gradientu se spočítá pomocí Pythagorovy věty a jeho úhel pomocí funkce arctan, viz následující vzorce 2.11.

$$g = \sqrt{g_x^2 + g_y^2} \qquad \phi = arctan\left(\frac{g_y}{g_x}\right) \tag{2.11}$$

Potom [46] co je pro každý pixel spočítán jeho gradient, je obrázek rozdělen do pravidelné mřížky. Běžně se využívá velikost buňky 8×8 pixelů, ale obecně lze zvolit libovolnou velikost – záleží na úrovni detailu. V každé buňce mřížky se spočítá histogram gradientů, který definují odpovídající pixely. Histogram představuje vektor o velikosti devíti tříd, jenž definuje rozsahy orientací gradientů s rozmezím 20° (od 0° do 180° při použití znaménka). Každý pixel tak může přispět do dvou tříd, kde celková velikost příspěvku je dána velikostí jeho gradientu. Třídy do kterých přispěje, ohraničují interval jeho spočtené orientace gradientu a konkrétní přidaná velikost do dané třídy je dána poměrově. Tedy vyšší příspěvek bude ve třídě, která má hodnotu bližší jeho orientaci.

Nyní je obrázek rozdělen na menší úseky v mřížce a pro každý úsek je spočítán histogram orientací gradientů. Protože jsou gradienty citlivé na osvětlení a některé části obrázku jsou světlejší než jiné, tak se provádí normalizace osvětlení, která zvyšuje odolnost příznaku na světelných podmínkách. Normalizace se provádí na čtyřech buňkách (v tomto případě 16×16 pixelů), kde každá buňka obsahuje vektor (histogram) gradientů o velikosti 9, které se spojí do jednoho vektoru s velikostí 36. Tento spojený vektor [11] se následně normalizuje pomocí metody L2-norm (lze i jinak) – viz následující vzorec 2.12.

$$f = \frac{v}{\sqrt{||v||_2^2 + e^2}}, \tag{2.12}$$

kde v je nenormalizovaný vektor obsahující všechny histogramy v daném bloku, $||v||_2$ je jeho 2-norma a e je malá konstanta. Tato normalizace [46] je provedena pomocí klouzavého okénka přes celý obrázek. Při velikosti obrázku 64×128 pixelů a velikosti normalizačního bloku 16×16 vznikne celkem $7 \cdot 15 = 105$ posunů bloku, kde každý posun generuje vektor příznaků o velikosti 36. Výsledný vektor příznaků celého obrázku bude pak mít velikost $105 \cdot 36 = 3780$ prvků, který je nakonec klasifikován pomocí binárního klasifikátoru SVM *(Support Vector Machines)*.

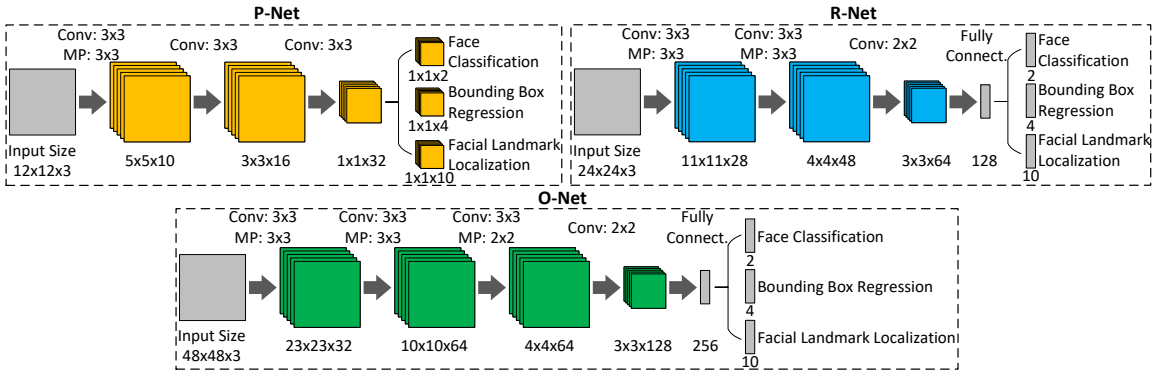
2.4.3 Detekce obličeje pomocí neuronových sítí

Moderní přístupy pro detekci obličeje používají neuronové sítě a přistupují k tomuto problému jako k detekci objektů. Jedním z důvodů je to, že i přesto, že přístupy jako Viola-Jones dosahují dobrého výkonu v reálném čase, tak se tyto detektory mohou výrazně zhoršit při aplikaci na lidské tváře s většími vizuálními variacemi, které se ve skutečném světě nacházejí. V následující části jsou proto lehce přiblížený některé architektury, na které lze aktuálně v této oblasti narazit.

Multi-task Cascaded Convolutional Networks

Multi-task Cascaded Convolutional Networks[10] (MTCNN) [16] je framework pro detekci a následné zarovnání obličeje s lokalizací orientačních bodů. Celý proces funguje v *„pipeline“*, která je rozdělena do tří fází, kde každá fáze obsahuje konvoluční neuronovou síť. Cílem je ohraničit obličej a umístit orientační body na jeho oči, nos a ústa. V první části se používá mělká síť (P-Net) pro rychlý návrh kandidátních regionů, v další části se pak tyto regiony upřesní složitější sítí (R-Net) a v konečné fázi dojde k dalšímu zpřesnění a určení pozic bodů pomocí komplexní KNN (O-Net). Architektury těchto sítí lze vidět na obrázku 2.20.

První krok [16, 56] obsahuje plnou konvoluční síť (FCN, tj. nepoužívá žádnou plně propojenou vrstvu a je složena pouze z konvolučních vrstev), která se používá k získání kandidátních regionů a regresi jejich bounding boxů. Získané bounding boxy, reprezentovány vektory, jsou následně vyfiltrovány technikou Non-Maximum Suppresion (NMS), která eliminuje překrývající se regiony. V druhé fázi jsou všechny tyto regiony přivedeny na vstup běžné konvoluční sítě, která dále provede redukci nesprávných oblastí, kalibraci s regresí bounding boxů a NMS. Výstupem této vrstvy jsou tři vektory, kde první určuje zda vstup obsahuje obličej, další vektor o velikosti 4 definuje rozměry bounding boxu a vektor s 10-ti prvky pro souřadnice orientačních bodů obličeje. Poslední fáze je velmi podobná druhé fázi, ale v tomto případě jde o detailnější analýzu obličeje, kde síť vygeneruje pět výsledných orientačních bodů na obličeji.



Obrázek 2.20: Architektura sítí P-Net *(Proposal Network)*, R-Net *(Refine Network)* a O-Net *(Output Network)*. Velikost kroku v konvolučních vrstvách je 1 a v pooling vrstvách 2. Převzato z článku [56].

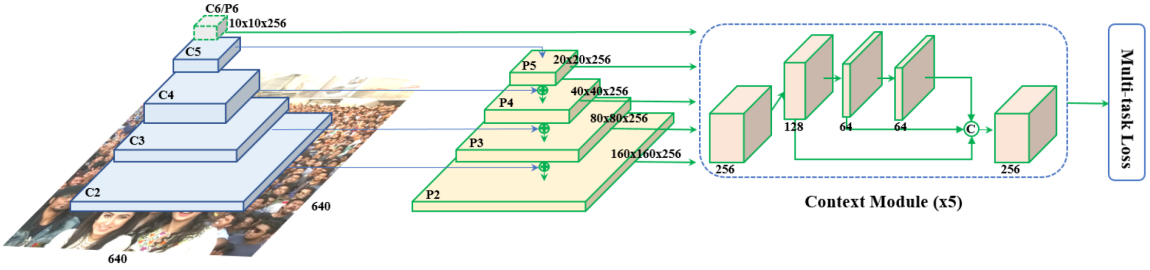
RetinaFace

RetinaFace[11] [12] je jednostupňový detektor obličeje, jenž provádí lokalizaci po pixelech na různých velikostech tváří s podporou strojového učení. Využívá *multi-task* strategii k současné predikci skóre obličeje, jeho bounding boxu, 3D rekonstrukci a odhadu pěti orientačních bodů na obličeji stejně jako v případě MTCNN.

Architektura řešení [12, 35] se skládá z pyramidy příznaků, z kontextového modulu a z kaskády *multi-task loss* funkcí – viz obrázek 2.21. Do pyramidy příznaků je přiveden vstupní obrázek, nad kterým je vygenerováno pět různých velikostí. První čtyři mapy

[10] MTCCN: https://github.com/ipazc/mtcnn
[11] RetinaFace: https://github.com/serengil/retinaface

příznaků jsou vypočítány pomocí sítě ResNet-152, která byla předtrénována na datasetu ImageNet-11k. Poslední mapa je spočtena pomocí konvoluce 3×3 s krokem 2, kde hodnoty jsou náhodně inicializovány metodou *„Xavier“*. Inspirací SSH[12] a PyramidBox[13] byly i zde aplikovány nezávislé kontextové moduly na úrovně pyramidy za účelem vylepšení modelovaného kontextu. Ovšem všechny 3×3 konvoluční vrstvy byly nahrazeny deformovatelnou konvoluční sítí (DCN).



Obrázek 2.21: RetinaFace je navržena na pyramidě příznaků s nezávislými kontextovými moduly. V návaznosti na kontextové moduly jsou spočítány *multi-task loss* funkce pro každou kotvu. Převzato z článku [12].

Ke zlepšení lokalizace obličeje [35] je použita kaskádová regrese spolu s *multi-task loss* funkcí. Tento mechanismus funguje tak, že první kontextový modul predikuje bounding box pomocí běžných kotev a poté následující moduly předpovídají přesnější boxy pomocí regresních kotev. Ztrátová funkce má následující tvar 2.13.

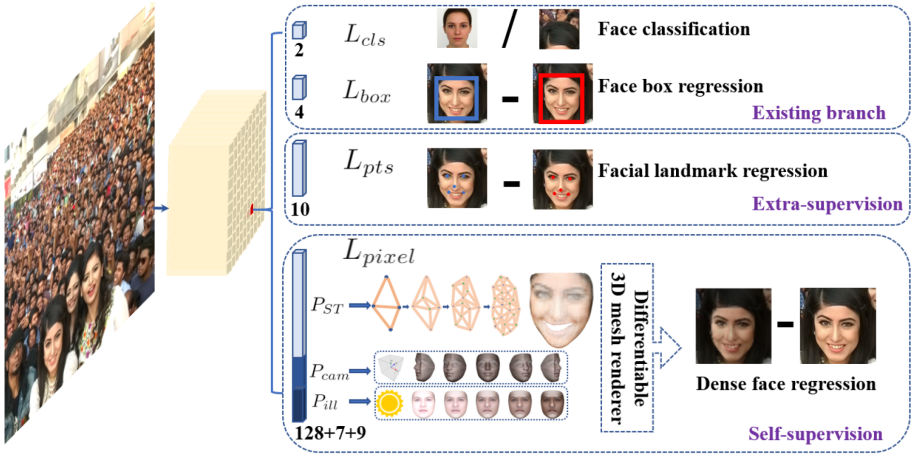
$$L = L_{cls}(p_i, p_i^*) + \lambda_1 \cdot p_i^* \cdot L_{box}(t_i, t_i^*) + \lambda_2 \cdot p_i^* \cdot L_{pts}(l_i, l_i^*) + \lambda_3 \cdot p_i^* \cdot L_{pixel}, \tag{2.13}$$

kde $L_{cls}(p_i, p_i^*)$ [12] reprezentuje klasifikaci obličeje, přičemž p_i je predikovaná pravděpodobnost, že kotva i obsahuje obličej a p_i^* je 1 pro pozitivní kotvu a 0 pro negativní. L_{cls} je softmax ztrátová funkce pro binární klasifikaci. Regrese bounding boxu je dána $L_{box}(t_i, t_i^*)$, kde t_i a t_i^* jsou souřadnice predikovaného a *ground truth* boxu. Souřadnice jsou normalizovány ve středovém formátu a $L_{box}(t_i, t_i^*) = R(t_i - t_i^*)$, kde R je ztrátová funkce $smooth - L_1$ definovaná v [14]. Regrese pěti bodů na obličeji je $L_{pts}(l_i, l_i^*)$, kde l_i a l_i^* reprezentují predikované a anotované body, L_{pts} je stejná ztrátová funkce jako R při regresi ohraničení. L_{pixel} je hustá regresní funkce popsaná v [12]. Parametry pro vyvážení funkce $\lambda_1 - \lambda_3$ jsou nastaveny na $0,25$, $0,1$ a $0,01$, čímž zvyšují význam ohraničení a poté bodům na obličeji. Výstup z této metody lze vidět na obrázku 2.22.

2.5 Rozpoznávání osob podle obličeje

Rozpoznání obličeje je proces, během kterého je tvář ze vstupního obrázku rozpoznávána vůči databázi již známých tváří. Moderní systémy dělí tento proces na čtyři kroky: detekce, zarovnání, reprezentace a klasifikace. Detekci obličeje byla věnována předchozí sekce 2.4 a pro tento účel je možné použít např. algoritmus Viola-Jones. Opět zde existuje více přístupů, které lze pro identifikaci uplatnit a dále jsou uvedeny pouze některé představitelé *state-of-the-art*, které v této oblasti utvořili významné milníky.

[12] Single Stage Headless Face Detector: https://github.com/mahyarnajibi/SSH
[13] PyramidBox: https://github.com/yxlijun/Pyramidbox.pytorch



Obrázek 2.22: Každá pozitivní kotva poskytuje skóre obličeje, jeho ohraničení, pět orientačních bodů a vrcholy 3D obličeje promítnuté na rovinu obrazu. Převzato z článku [12]

2.5.1 DeepFace

DeepFace[14] je systém pro rozpoznávání obličejů vytvořený společností Facebook, který ho využívá k označování osob na obrázku. K rozpoznávání používá devítivrstvou neuronovou síť, která obsahuje více než 120 mil. parametrů a byla natrénována na datasetu se 4 mil. označených obrázků s obličeji od uživatelů Facebooku. Autoři uvádějí přesnost modelu na 97,35 %, čímž může dosahovat vyšší přesnosti než člověk. Tento model vyniká především vlastním způsobem zarovnání tváře a reprezentace rysů, které využívá namísto konvenčních metod strojového učení. Tato skutečnost pak umožňuje provést analýzu nad větším množství dat.

Cílem [42] zarovnání tváře je ze vstupního obrázku, který může obsahovat libovolně pootočený obličej, vygenerovat pohled na jeho čelní stranu. K tomuto účelu byla navržena metoda 3D frontalizace, během niž dochází k 2D zarovnání, poté 3D zarovnání a ke konečné frontalizaci (viz obrázek 2.23). Konkrétně [49] celý proces obsahuje následující kroky:

1. **Detekce referenčních bodů** – Ve vstupním obrázku je identifikován obličej pomocí šesti referenčních bodů, kde dva body jsou na očích, jeden na špičce nosu a tři body na ústech.

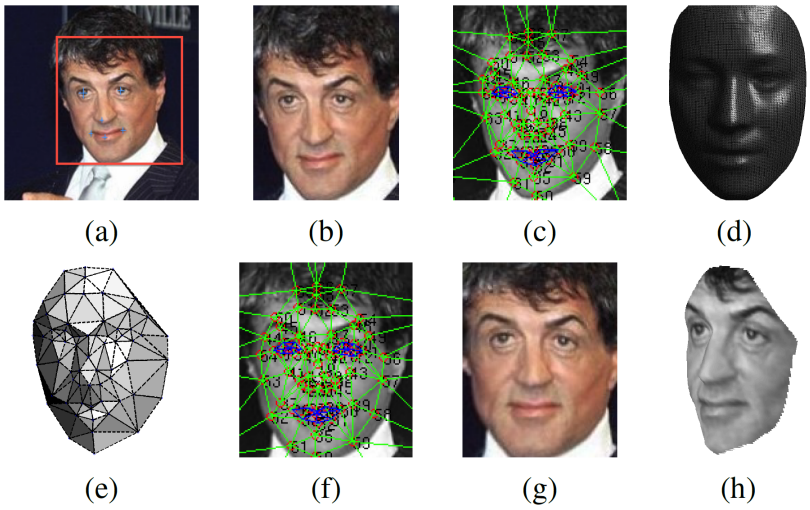
2. **Zarovnání a oříznutí** – Následně pomocí detekovaných bodů je tvář oříznuta a vzniká tak 2D obrázek obličeje.

3. **Detekce 67 bodů** – Na obrázek obličeje se aplikuje mapa 67 referenčních bodů za účelem vygenerovat odpovídající 3D model. Tento krok je velmi důležitý pro kompenzaci rotace obličeje.

4. **Odhad orientace** – Poté je proveden odhad orientace obličeje pomocí vztahu mezi jeho 2D a 3D reprezentací.

5. **Zarovnání obličeje** – V poslední fázi je 3D reprezentace transformována do frontálního pohledu. Výsledkem je tvář v čelním pohledu, která je vizuálně podobná obličeji skutečného jednotlivce.

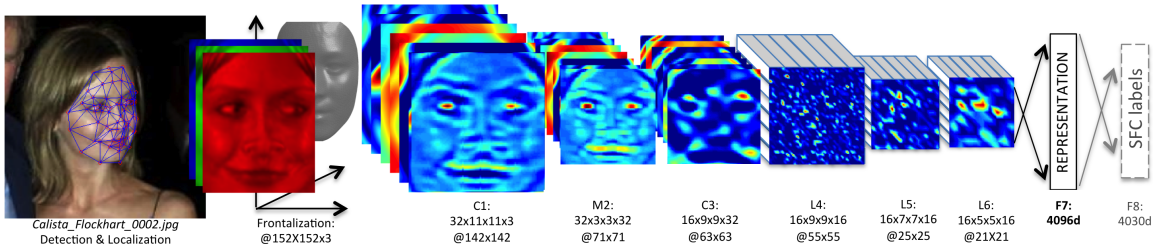
[14] DeepFace: https://github.com/serengil/deepface



Obrázek 2.23: Proces zarovnání obličeje. (a) Detekovaný obličej pomocí 6-ti referenčních bodů. (b) Zarovnaný a oříznutý 2D obličej. (c) Na 2D oříznuté tváři je detekováno 67 bodů. (d) Referenční 3D reprezentace obličeje. (e) Viditelnost trojúhelníku s ohledem na pozici kamery za účelem odhadu orientace obličeje. (f) Všech 67 bodů reprezentovaných 3D modelem je použito k natočení 2D modelu. (g) Konečný frontalizovaný výřez obličeje. (h) Nový pohled generovaný 3D modelem (pro zajímavost). Převzato z článku [49].

Potom [42, 49] co je obličej zarovnaný a reprezentovaný RGB obrázkem o velikosti 152×152, následuje proces klasifikace pomocí hluboké neuronové sítě (viz obrázek 2.24). Síť zpočátku obsahuje konvoluční vrstvu s 32 filtry o velikosti 11×11 následovanou max-pooling vrstvou 3×3 s krokem 2 a konvoluční vrstvou 16 filtrů 9×9. Tyto vrstvy mají za úkol získat nízkoúrovňové příznaky jako jsou hrany a textury.

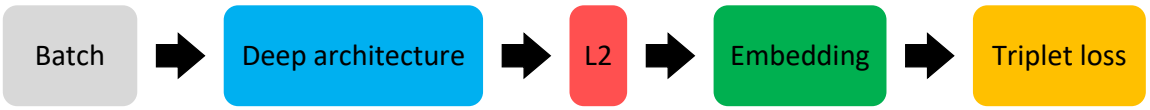
Následují [42, 49] další tři lokálně propojené vrstvy, které stejně jako konvoluční vrstvy obsahují sadu filtrů, jenom je každý filtr aplikovaný na jinou oblast v mapě příznaků. Tím architektura sítě využívá skutečnosti, že různé oblasti v zarovnaném obrázku mají různou entropii, např. oblast mezi očima a obočím vykazuje vysokou míru rozlišitelnosti oproti oblasti mezi ústy a nosem. Poslední dvě vrstvy jsou plně propojené a jsou schopny zachytit korelaci mezi prvky obličeje. Výstup předposlední vrstvy reprezentuje obličej jako vektor příznaků a výstup poslední plně propojené vrstvy je zpracován softmax funkcí, jenž vytváří distribuci mezi třídami. Poslední lokálně a plně propojené vrstvy obsahují velké množství parametrů (zhruba 95 % parametrů sítě).



Obrázek 2.24: Architektura sítě DeepFace. Předzpracovaný obrázek je předán konvoluční vrstvě, která je následovaná pooling a poté další konvoluční vrstvou. Dále jsou aplikovány tři lokálně propojené vrstvy a na konci se nachází dvě plně propojené. Převzato z článku [49].

2.5.2 FaceNet

FaceNet[15] [44] je model vytvořený společností Google, který dosahuje velmi vysoké přesnosti na rozsáhlých datasetech jako je 95, 12 % na databázi obličejů YouTube nebo 99, 63 % na LFW *(Labeled Faces in the Wild)*. Pracuje tak [29], že každý obrázek obličeje mapuje do Euklidovského prostoru, kde vzdálenosti mezi jednotlivými tvářemi odpovídají jejím podobnostem. Jako základní architekturu [43] využívá ZF-Net nebo Inception, na kterých dále staví. Přidává několik konvolučních vrstev 1×1 za účelem snížení počtu parametrů a využívá *end-to-end* učení. Výstupem použité neuronové sítě je vektor příznaků *(embedding)* o velikosti 128, na kterém je provedena L_2 normalizace. Tyto embeddingy jsou pak předány loss funkci, která definuje podobnost obličeje (viz obrázek 2.25). Cílem aplikované loss funkce je zajistit, aby kvadratická vzdálenost mezi dvěma obrázky tváře byla malá v případě stejné identity osoby a byla nezávislá na stavu snímku či výrazu jedince. Naopak musí poskytnout velkou odezvu v případě snímků s rozdílnými osobami. K tomuto účelu byla představena nová ztrátová funkce nazývaná Triplet loss.

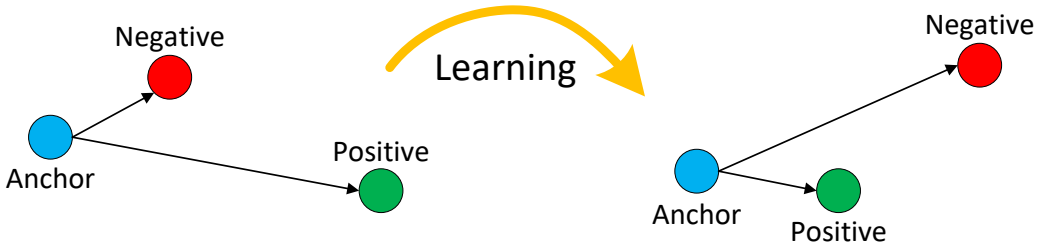


Obrázek 2.25: Struktura modelu FaceNet, kde na vstupní vrstvu je přiveden batch a hluboká KNN je následována L_2 normalizací, jejíž výsledkem je embedding obličeje, který slouží jako vstup Triplet loss funkce. Převzato z článku [44].

Triplet loss funkce [29] využívá trojici obrázků: referenční *(anchor)*, pozitivní a negativní. Cílem je, aby vstupní obrázek obličeje x_i^a *(anchor)* byl blíže pozitivním obrázkům x_i^p, které patří stejné osobě, než těm ostatním x_i^n, jenž obsahují odlišné osoby (viz obrázek 2.26). Funkci lze formálně definovat vztahem 2.14.

$$L = \sum_{i}^{N} \left[||f(x_i^a) - f(x_i^p)||_2^2 - ||f(x_i^a) - f(x_i^n)||_2^2 + \alpha \right]_+ , \tag{2.14}$$

kde x_i reprezentuje obrázek, $f(x_i)$ jeho odpovídající embedding a α je hyperparametr, který definuje hranici rozlišitelnosti neboli prahovou hodnotu, která určuje rozdíl mezi páry obrázků.



Obrázek 2.26: Triplet loss funkce se snaží minimalizovat vzdálenost mezi referenčním obrázkem a jeho pozitivním vzorkem, který spadá do stejné třídy a přitom maximalizovat vzdálenost od negativních prvků, které obsahují jinou totožnost. Převzato z článku [44].

[15] FaceNet: https://github.com/davidsandberg/facenet

Výběr správných trojic [29, 43] obrázků je velmi důležitý, protože existuje mnoho párů, které podmínku vzdálenosti splní a model tak bude při učení velmi pomalu konvergovat. K rychle konvergenci je proto důležité vybrat takové obrázky, které porušují výše uvedenou rovnici 2.14. Tedy takové, aby vzdálenost mezi referenčním a pozitivním obrázkem byla co největší a zároveň vzdálenost mezi referenčním a negativním případem byla co nejmenší. Potom bude model zpracovávat pouze užitečné informace a jeho učení se urychlí. Najít takové páry je ovšem výpočetně náročné, a tak se hledají pouze v rámci mini-batche (tj. 1000-2000 vzorků).

2.6 Sledování objektů

Sledování objektů *(tracking)* [36] je poměrně častá úloha při zpracování videozáznamu, která umožňuje vytvořit asociaci mezi objekty napříč snímky. Tato oblast je rozdělena na kategorii *Single-Object tracking* (SOT), která umožňuje sledovat pouze jeden cíl a v těchto přístupech je znám jeho vzhled. Další kategorií je *Multiple-Object tracking* (MOT), jenž umožňuje sledovat více objektů současně bez jakékoliv předchozí znalosti o jejich vzhledu nebo umístění, avšak tyto metody vyžadují jako první krok provést detekci objektů. Následující text je zaměřen na sledování více objektů (MOT), pro kterou existuje řada metod jako je MHT[16], MDNet[17], GOTURN[18], ROLO[19] atd. Mezi současné nejmodernější řešení však patří algoritmy SORT a DeepSORT, které umožňují i sledování v reálném čase s vysokou snímkovou frekvencí.

2.6.1 SORT

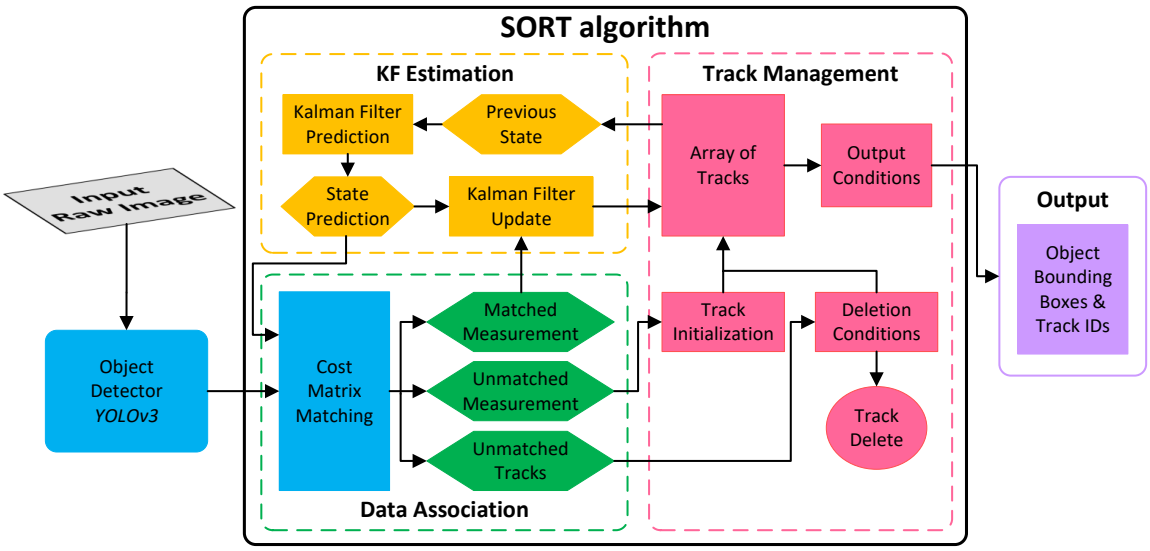
SORT[20] *(Simple Online and Realtime Tracking)* [26] algoritmus lze rozdělit do čtyř hlavních částí: detekce, odhad, asociace a správa stop (viz obrázek 2.27). Detekci objektů byla věnována předchozí část tohoto textu a pro tento účel lze použít některý model zmíněný v sekci 2.3.2. Cílem odhadu je predikovat příští regiony zájmů pro aktuální detekce a šířit je do dalšího snímku. K tomuto účelu se používá Kalmanův filtr (KF), který využívá konstantní rychlosti a gaussového rozdělení pro odhad oblastí, v závislosti na aktuálním pohybu objektu. Pokud je detekovaný objekt [36] přidružen ke sledovanému objektu (stopě), tak je použit jeho bounding box k aktualizaci dané stopy. Avšak pokud k dané stopě není zrovna přidružená žádná detekce, tak je její stav pouze předpovídán pomocí lineárního modelu rychlosti.

Při asociaci [4, 26, 36] dochází k přiřazení nových detekcí k již existujícím stopám. To se provádí na základě překryvu každého detekovaného bounding boxu se všemi predikovanými boxy z předchozího snímku, jenž je definován velikostí průniku nad sjednocením (IoU). Z hodnot je následně sestavena matice nákladů přiřazení *(assignment cost matrix)*, která je poté použita k výslednému přiřazení pomocí Maďarského algoritmu *(Hungarian algorithm)*. Samotná přiřazení jsou ještě filtrována s ohledem na minimální IoU, přičemž dochází k eliminaci sdružení, jejichž hodnota je menší než daný práh.

Správa stop [4, 26] je zodpovědná za vytváření a mazání existujících stop. K vytváření nových stop dochází pokud do scény přijde nový objekt nebo pokud překrytí existující sto-

[16] OpenMHT: https://github.com/jonperdomo/openmht
[17] MDNet: https://github.com/hyeonseobnam/MDNet
[18] GOTURN: https://github.com/davheld/GOTURN
[19] ROLO: https://github.com/Guanghan/ROLO
[20] SORT: https://github.com/abewley/sort

pou je menší než minimální IoU. Při tomto procesu je stopě přiřazen unikátní identifikátor, stav KF je inicializován jejím bounding boxem a rychlost je pak nastavena na 0, jelikož v tuhle dobu není známa. Stopa se pak nějaký čas nachází v nepotvrzeném stavu, dokud nejsou k její trajektorii přiřazeny další detekce z následujících snímků videa. Díky tomu je tak efektivně zabráněno vzniku falešně pozitivních jevů. Naopak pokud stopa není detekována po určitý počet snímků za sebou, tak dochází k její odstranění. Počet snímků v této sekvenci lze nastavit a pokud se daný objekt poté znovu objeví, bude implicitně sledován pod novou identitou. Mazání stop také pomáhá regulovat počet aktuálně udržovaných stop, což opět zabraňuje vzniku falešně pozitivních jevů a snižuje prostorovou náročnost řešení.



Obrázek 2.27: Přehled architektury algoritmu SORT pro sledování objektů. Převzato z článku [36].

2.6.2 DeepSORT

DeepSORT[21] [26] je vylepšení algoritmu SORT (viz předchozí sekce 2.6.1) o metrikou hloubkové asociace. Přestože SORT dosahuje celkově pozoruhodných výsledků v přesnosti sledování, tak i přes účinnost Kalmanova filtru vrací poměrně mnoho kandidátních stop. To zapříčiňuje časté záměny identit, kterou dále znatelně ovlivňuje úhel pohledu aj. Jako kompenzace tohoto nedostatku je v tomto algoritmu zavedena další metrika, která je založená na „vzhledu“ objektu, a tím tak vylepšuje proces asociace. Architektura tohoto algoritmu je zobrazena na obrázku 2.28.

Asociace detekovaných objektů [36] ke stopám je i zde provedena pomocí Maďarského algoritmu stejně jako v předchozím algoritmu SORT. DeepSORT ovšem nahrazuje asociační metriku za dvoudílnou porovnávací kaskádu. V první části kaskády jsou použity metriky pohybu a vzhledu pro přiřazení k platným stopám, v druhé části se pak používá stejná strategie jako v algoritmu SORT pro asociaci nepřiřazených či nepotvrzených stop s nepřiřazenými detekcemi.

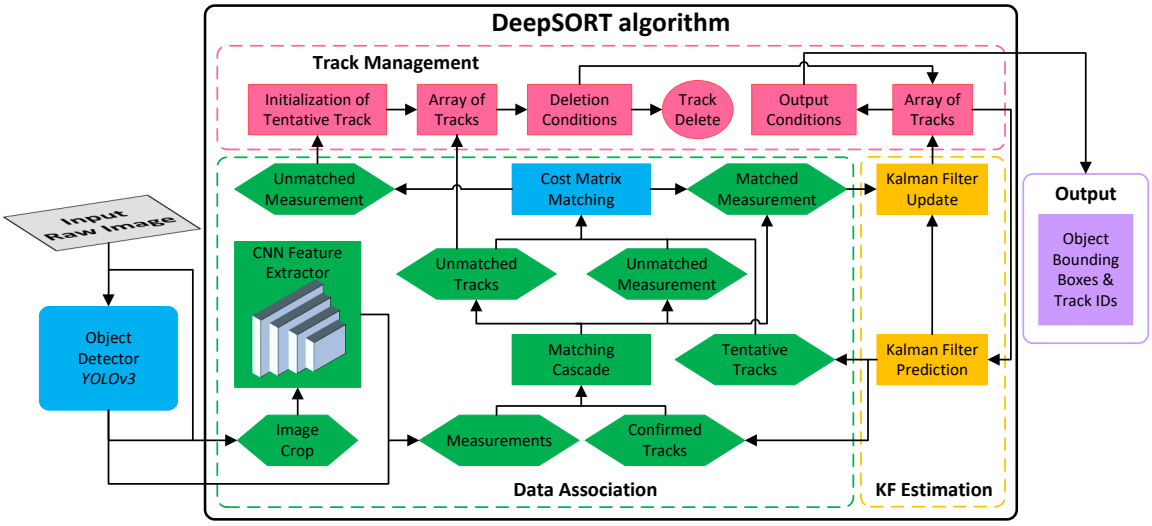
Informace o pohybu [36] jsou začleněny v kvadratické Mahalanobisové vzdálenosti mezi predikovanými oblastmi Kalmanova filtru a současnými detekcemi. Tyto informace jsou

[21] DeepSORT: https://github.com/nwojke/deep_sort

užitečné zejména v krátkodobé predikci. Pomocí této metriky [52] je také možné vyloučit nepravděpodobné asociace prahováním vzdálenosti na 95 % míry jistoty. Metrika vzhledu je založená na nejmenší kosinusové vzdálenosti mezi stopami a tzv. „deskriptory vzhledu“, které jsou zvláště užitečné pro obnovení stopy po dlouhodobých výpadcích, kdy je pohyb méně relevantní. Deskriptor vzhledu je spočítaný nad detekcí pomocí předtrénované KNN, která byla natrénovaná na rozsáhlém datasetu osob. Asociační metoda je potom definována kombinací těchto dvou metrik a lze ji zapsat vztahem 2.15.

$$D = \lambda \cdot D_m + (1 - \lambda) \cdot D_c, \tag{2.15}$$

kde D_m je Mahalanobisová vzdálenost, D_c je kosinusová vzdálenost a λ je váhový parametr.



Obrázek 2.28: Přehled architektury algoritmu DeepSORT. Převzato z článku [36].

Kapitola 3

Technologie pro vývoj internetových aplikací

V následující kapitole budou detailněji popsány použité technologie při implementaci. Jako první je uvedeno prostředí Node.js, jelikož na něm staví další použité nástroje. Následující část je pak věnována frameworku Express.js a jeho hlavním inovacím, které s sebou do tohoto prostředí přináší, včetně způsobu, jakým zpracovává příchozí požadavky. Dále je text zaměřen na knihovnu React a její principy pro vytváření uživatelského rozhraní. Tato sekce také podrobněji popisuje životní cyklus komponent a jak se s nimi zachází. Na konec jsou uvedeny komunikační technologie, které byly použity pro přenos dat jako jsou Socket.io a REST API.

3.1 Node.js

Node.js[1] [23, 37] je multi-platformní prostředí pro vývoj vysoce škálovatelných internetových aplikací, které je postaveno na JavaScriptovém open-source enginu V8[2], jenž byl původně vytvořený pro Google Chrome. Tento engine byl vytvořen za účelem zvýšení rychlosti JavaScriptu začleněním *„just-in-time“* (JIT) kompilátoru, který kompiluje JavaScript do strojového kódu. Tím je umožněno používat JavaScript i mimo prohlížeč a vývojářům je tak dovoleno psát v tomto jazyce nejen klientskou stranu aplikace, ale taktéž serverovou část. V důsledku lze takto použít pouze jeden programovací jazyk na celou aplikaci a prosadit tak paradigma *„JavaScript everywhere“*. Nejčastěji se ovšem Node.js používá pro vývoj síťových programů jako jsou webové servery, jelikož se drží neblokujícího, událostmi řízeného modelu.

3.1.1 Architektura

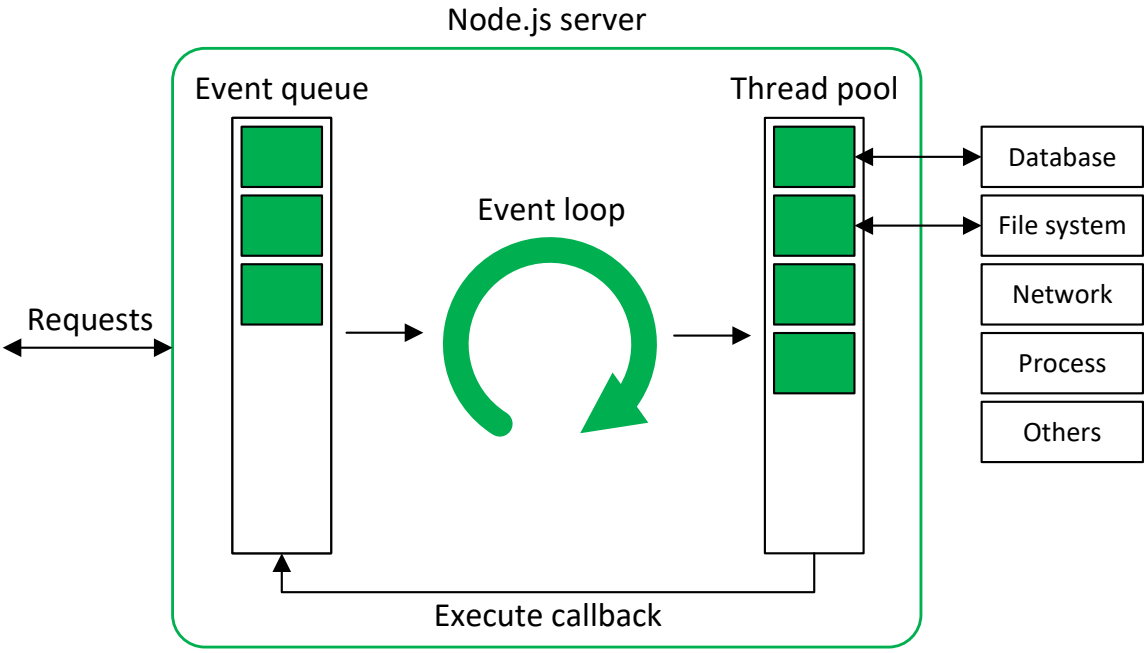
Běžné serverové systémy využívají *„one-thread-per-client“* přístup, kde s každým novým požadavkem je vytvořeno nové vlákno, které daný požadavek obslouží (např. Apache). Přestože tento přístup má své výhody, tak zatěžuje systémovou operační paměť a je nutné také vzít v úvahu režii spojenou s vytvářením samotných vláken, obzvlášť v případě, kdy požadavky obnášejí jednoduché, výpočetně nenáročné operace.

[1] Node.js: https://nodejs.org
[2] V8 engine: https://v8.dev

Na rozdíl tomu Node.js využívá velmi malé množství vláken k obsluze všech příchozích požadavků. Tím jsou ušetřeny systémové prostředky spojené s režií vláken a lze tak strávit více času nad samotnými požadavky. Na druhou stranu, protože Node.js disponuje pouze několika vlákny, je nutné s nimi zacházet moudře. Dělí se na dvě skupiny – Event loop (smyčku událostí) a Thread pool.

Smyčka událostí [10] je podstatnou částí Node.js, která umožňuje provádět asynchronní, neblokující operace. Obsahuje pouze jedno vlákno, které se stará o všechny příchozí požadavky v nekonečné smyčce (viz obrázek 3.1). Příchozí požadavky se řadí do fronty událostí a hlavní vlákno ho buď vykoná, jedná-li se o blokující operaci nebo ho předá ke zpracování volnému pracovnímu vláknu, které po dokončení asynchronní operace invokuje *callback* funkci, která je opět zaslána zpět do vstupní fronty. Lze si povšimnout, že pro rychlý chod programu je nutné tuto smyčku událostí nezatěžovat náročnými blokujícími operacemi, protože by mohlo dojít k jejímu zablokování a tím k zamrznutí programu.

Thread pool [10] poskytuje dodatečná pracovní vlákna (v základu 4, ale lze konfigurovat až na 1024), na které může hlavní smyčka delegovat asynchronní či náročné požadavky včetně blokujících I/O operací. Typicky mezi takové požadavky patří přístup k souborovému systému, databázím či síťové transakce aj. O celou tuto funkcionalitu včetně smyčky událostí se stará knihovna Libuv[3], která tvoří podstatnou část Node.js a kde jsou tyto náročné operace většinou již asynchronně implementovány.

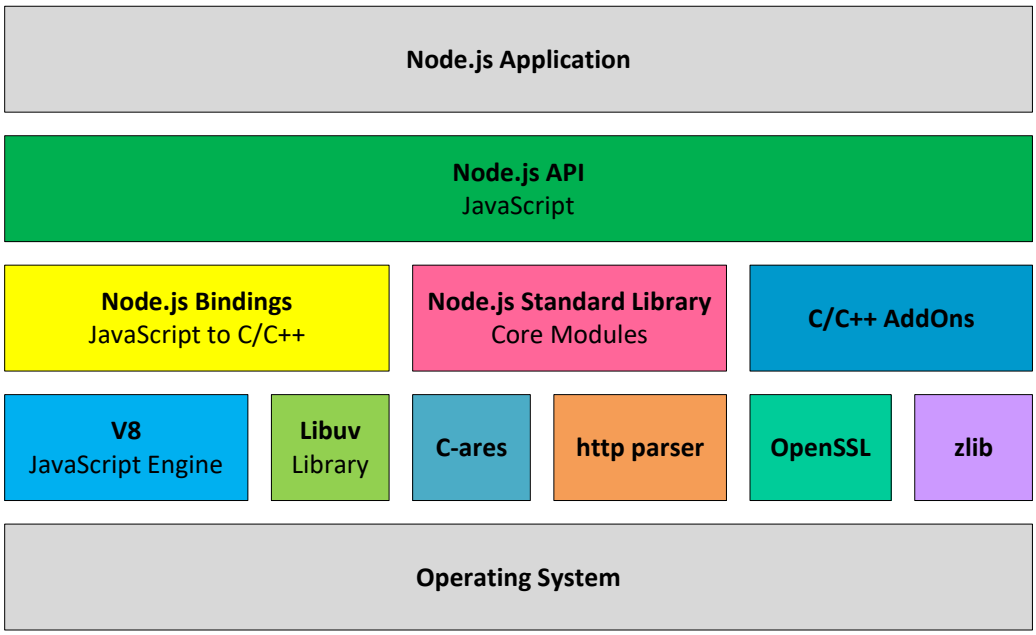


Obrázek 3.1: Architektura knihovny Libuv pro obsluhu požadavků pomocí smyčky událostí. Převzato z článku [10].

Doposud byly zmíněny pouze závislosti Node.js na enginu V8 a knihovně Libuv. Tyto dvě komponenty patří sice mezi nejdůležitější, ovšem nejsou jediné a celou architekturu Node.js lze vidět na obrázku 3.2. Mezi další závislosti [5] patří:

[3]Libuv: https://libuv.org

- **llhttp**[4] – Knihovna pro parsování HTTP. Navržena tak, aby neprováděla žádná systémová volání či alokace.
- **c-ares**[5] – Knihovna pro asynchronní DNS požadavky použité v DNS modulu.
- **OpenSSL**[6] – Pro kryptografické funkce použité v TLS (SSL) a crypto modulech. Poskytuje mnoho funkcí, na které moderní weby spoléhají z hlediska bezpečnosti.
- **zlib**[7] – Knihovna pro synchronní, asynchronní a stream kompresi a dekompresi.



Obrázek 3.2: Architektura Node.js se všemi jejími závislostmi. Převzato z článku [5].

3.1.2 Moduly

Node.js [8] umožňuje vývojářům definovat vlastní zapouzdřenou funkcionalitu a šířit ji v ekosystému modulů *(packages)*, jenž se nachází mimo jádro Node.js. Tento princip má velký dopad na strukturu kódu programu a kulturu tohoto prostředí, protože dává svobodu vývojářům experimentovat a opakovat určitá existující řešení na problémy, se kterými se během vývoje potýkají. Spolu s NPM (viz následující sekce 3.1.3) pomáhá Node.js vyřešit konflikty v závislostech tak, že každý nainstalovaný balíček má svoji vlastní oddělenou množinu závislostí. Tímto způsobem je možné zajistit velkou míru znovupoužitelnosti a vytvářet nové aplikace podstatně rychleji. Moduly [9] se dělí do tří kategorií:

- **Základní moduly** – Node.js má spoustu vestavěných modulu, které jsou již součástí prostředí a jsou dodány během instalace.

[4] llhttp: https://github.com/nodejs/llhttp
[5] c-ares: https://c-ares.org
[6] OpenSSL: https://www.openssl.org
[7] zlib: https://zlib.net

- **Lokální moduly** – Na rozdíl od vestavěných a externích modulů jsou lokální moduly vytvářeny přímo ve vyvíjené aplikaci a zpřístupněny jiným souborům pomocí exportů.
- **Moduly třetích stran** – Jsou moduly, které jsou dostupné na internetu přes NPM. Tyto moduly mohou být nainstalovány přímo do projektu nebo také globálně.

3.1.3 Node Package Manager

Velké množství funkcionality [37] spojené s Node.js přichází prostřednictví modulů třetích stran. Tyto moduly lze nainstalovat manuálně do prostředí vyvíjené aplikace nebo pomocí specializovaného nástroje. Jedním z těchto nástrojů je Node Package Manager[8] (dále jen NPM), což je výchozí správce balíčků dodaný při instalaci Node.js, ačkoliv nemusí být vždy poskytnutý v nejnovější verzi. NPM se ovšem neomezuje pouze na správu balíčku, ale taktéž nabízí kompletní správu projektu jako je definování vlastních příkazů, sledování závislostí balíčků či jejich zranitelností. Nově nainstalované balíčky přidává NPM do adresáře `node_modules`.

NPM [40] využívá ke správě projektu soubor `package.json`, ve kterém jsou obsažena metadata projetu. Zde je definován kromě jména projektu, verze, licence aj. také seznam přímých závislostí projektu na jiných balíčcích, včetně jejich verze. Tento soubor je velmi důležitý a musí být přenášen spolu s aplikací, neboť při její sestavení v novém prostředí se znovu podle něj stáhnou zde definované balíčky v odpovídajících verzích a sestaví se nový strom závislostí.

Dalším podstatným souborem je `package-lock.json` [40], který obsahuje přesný strom závislostí, jenž zaručuje kompletně shodné závislosti všech balíčků v totožných verzích při opětovném sestavení aplikace. Rozdíl oproti klasickému sestavení je v tom, že nebudou zaručeny pouze stejné verze přímých závislostí, ale taktéž závislosti těchto závislostí. To umožňuje pokaždé sestavit identický celý strom. Tento soubor je automaticky generovaný NPM při modifikaci souboru `package.json` nebo adresáře `node_modules`, ovšem již není nutný k sestavení a používá se spíše při vývoji průběžných integrací do aplikace od více vývojářů.

3.2 Express.js

Express.js[9] (dále jen Express) [6, 33] je minimalistický a flexibilní webový framework postavený na základu Node.js. Je určený především k vývoji internetových aplikací či API, ale postupem času se stal de facto standardním serverovým aplikačním rámcem pro Node.js. Totiž běžný proces vývoje serveru postaveného pouze na Node.js zahrnuje stále opakující se a zdlouhavé úkony jako je analýza těl HTTP požadavků, správa sezení, cookies, extrahování parametrů z URL aj. Express se snaží tyto rutinní činnosti více abstrahovat a poskytnout tak způsoby k jejich elegantnímu řešení.

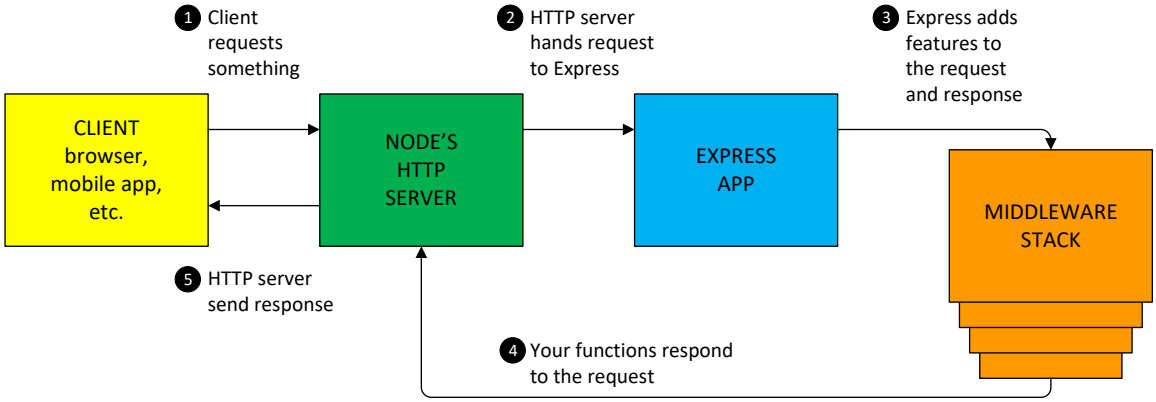
Přestože Express [6] v základu poskytuje jen minimální rámec klíčových funkcí, je také velmi snadno rozšiřitelný. Jelikož se stále jedná o framework postavený na Node.js a o JavaScript, je možné snadno zpřístupnit balíčky třetích stran pomocí NPM a přidat tak pouze to, co je nutné. Některé balíčky jsou navíc vytvořeny výhradně pro funkce Expressu, ale obecně lze využít každý balíček pro Node.js.

[8] NPM: https://www.npmjs.com
[9] Express.js: https://expressjs.com

3.2.1 Middleware

Middleware [6, 21] jsou funkce, které se provádějí během životního cyklu požadavku na serveru v tzv. *„pipeline“*, kde záleží na pořadí. Toto zřetězení umožňuje rozdělit logiku kódu na menší celky a vytvořit tak znovupoužitelné middleware funkce. Ve skutečnosti samotný Express je takto sestavený výhradně z middlewaru (viz obrázek 3.3). Funkce mají parametry `request` a `response` objekty pro každou cestu, ke které je připojená a parametr `next`, což je funkce. Každá middleware funkce může provádět následující:

- Spustit libovolný kód pro obsluhu požadavku.
- Provést změny v objektech požadavku a odpovědi.
- Ukončit životní cyklus požadavku.
- Zavolat další middleware v zásobníku pomocí funkce `next`.



Obrázek 3.3: Diagram znázorňující životní cyklus požadavku, který je na straně Expressu zpracován pomocí sekvence middlewaru. Převzato z knihy [21].

3.2.2 Směrování

Směrování *(routing)* [21, 33] umožňuje definovat koncové body *(endpoints)*, kam mohou uživatelé zasílat své požadavky pomocí URI a specifické HTTP metody. Každý koncový bod má poté jednu či více obslužných funkcí (jako middleware), které jsou postupně za sebou provedeny, pokud dojde k jejich kompletní shodě. Express mimo jiné dovoluje organizovat tyto cesty do menších sekcí zvaných *„routers“*. Definice koncového bodu má potom formát tak, jak je uvedeno v následujícím výpisu 3.1.

```
1       app.method(path, handler)
```

Výpis 3.1: Struktura definice koncového bodu v Express.js. Kde `app` je instance třídy express, `method` je použitá HTTP metoda požadavku, `path` je cesta na serveru ke koncovému bodu a `handler` je obslužná funkce.

3.3 React

React[10] je open-source JavaScriptová knihovna pro tvorbu uživatelského rozhraní, která je velmi dobře optimalizována pro práci s často měnícími se daty. Zejména se používá při tvorbě jednostránkových aplikací tzv. SPAs *(Single Page Applications)*, mobilních aplikací pomocí React Native[11] nebo desktopových aplikací využívajících Electron[12].

Aplikace implementované v Reactu [3, 24] umožňují zápis v JavaScriptovém rozšíření JSX (viz výpis 3.2), jenž se používá k definici toho, jak má uživatelské rozhraní vypadat. Syntax je velmi podobný HTML či XML, ale doopravdy se jedná pouze o syntaktický cukr, jehož cílem je zajistit čitelnější definice React elementů. Webové prohlížeče ovšem této syntaxi nerozumí a je tedy nutné ji před nasazením na web přeložit.

O tuto část se stará Babel[13] [3], což je JavaScriptový transpilátor, který se hlavně používá pro převod ECMAScriptu 2015+ (ES6+) do zpětně kompatibilních verzí JavaScriptu, které mohou být spuštěné na starších enginech (např. v prohlížečích). Kromě toho Babel umožňuje také převádět JSX do prostého JavaScriptu (viz výpis 3.3). Samotný React však JSX syntax nevyžaduje a je možné psát přímo v čistém JavaScriptu, ale v praxi se tento přístup příliš nevyužívá – viz následující příklad:

```
1  <MyButton color="blue" shadowSize={2}>
2      Click Me
3  </MyButton>
```

Výpis 3.2: Definice elementu v syntaxi JSX.

```
1  React.createElement(
2      MyButton,
3      {color: 'blue', shadowSize: 2},
4      'Click Me'
5  )
```

Výpis 3.3: Ekvivalentní zápis elementu z výpisu 3.2 v JavaScriptu.

3.3.1 Objektový model dokumentu

Objektový model dokumentu (dále jen DOM z ang. názvu *„Document Object Model“*) [40] je standardní a objektově orientovaná reprezentace HTML/XML dokumentu ve stromové struktuře, kde každý element v dokumentu je v této hierarchii reprezentován odpovídajícím uzlem. Tím, že tento formát je takto logicky strukturovaný, je umožněno snadno měnit obsah či pracovat s konkrétní částí dokumentu přímo z JavaScriptu.

Manipulace s DOM [30] je ovšem nákladná operace, jelikož pokaždé, když se změní stav programu, musí se strom znovu upravit a překreslit uživatelské rozhraní, aby byly uživateli reflektovány nové změny. Nejde ani tak o samotnou aktualizaci stromové struktury, jelikož na to již existuje spousta rychlých algoritmů. Co se ukazuje jako nákladné je fakt, že se musí znovu překreslit aktualizovaný element a celý jeho podstrom, zejména pokud daný podstrom obsahuje velké množství dalších elementů.

[10] React: https://reactjs.org
[11] React Native: https://reactnative.dev
[12] Electron: https://www.electronjs.org
[13] Babel: https://babeljs.io

React se snaží tomuto problému předcházet a využívá koncept zvaný **virtuální DOM**. Virtuální DOM [30] je vlastně odlehčená kopie reálné DOM reprezentace a tedy každý uzel v reálném stromě má zastoupení i v tom virtuálním, s tím rozdílem, že provedené změny ve virtuální reprezentaci nejsou vykresleny na obrazovku uživatele. Díky této vlastnosti je práce s virtuálním DOM časově velmi efektivní. Ve skutečnosti to probíhá tak, že před každou novou změnou stavu v programu, React vytvoří kopii aktuálního virtuálního DOM pomocí *„snapshotu“* a poté tuto kopii modifikuje. Následně porovná aktuální a upravenou verzi a detekuje rozdíly. Nakonec jsou tyto rozdíly aktualizovány v reálném DOM tím nejlepším možným způsobem. Tento proces se také nazývá *„diffing“* a může připomínat aplikaci opravného patche.

React s optimalizací však zachází ještě dále a provádí aktualizace stavu po dávkách *(batch update)* za účelem zvýšení výkonu aplikace. Stav programu se totiž provádí asynchronně a React tak při každé aktualizaci stavu ještě chvíli vyčkává na případné další nové změny a poté tyto změny překreslí zároveň.

3.3.2 Komponenty

Komponenty jsou základní, nezávislé a opakovaně použitelné části kódu, které slouží k definici uživatelského rozhraní a umožňují ho dekomponovat na menší části. Výsledná kompozice komponent má potom stromovou strukturu, kde kořen stromu je komponenta na nejvyšší úrovni, která je potom při překladu napojena na výchozí prvek (`div` element s identifikátorem `root`) ve vstupním bodě aplikace. Každá komponenta může mít vstupy (nazývané *„properties“* zkr. *„props“*) či vlastní stav a musí vracet React elementy.

Vstupy slouží k distribuci dat napříč aplikací a jsou určeny pouze ke čtení. React používá tzv. jednosměrnou datovou vazbu a vzhledem k tomu, že komponenty jsou organizovány do stromové struktury, tak může pouze nadřazená komponenta zpřístupnit hodnoty své přímo podřízené komponentě. Opačný směr předávání dat je možný jen přes referenci, ale většinou tento přístup nepatří mezi dobré zásady.

Stav [40] naopak umožňuje komponentě spravovat vlastní data, která nelze číst a nastavit mimo danou komponentu. Data uložená ve stavu jsou typicky zobrazována uživateli a postupem času se mění při interakci s aplikací pomocí událostí. Kdykoliv dojde k aktualizaci stavu či vstupu, bude komponenta znovu překreslena. Z hlediska stavu se pak komponenty dělí na:

- **Bezstavové**
  - Nemají žádný stav
  - Můžou mít vstupy
  - Překreslení je závislé pouze na vstupech
- **Stavové**
  - Mají vlastní stav
  - Můžou mít vstupy
  - Překreslení je závislé na vstupech a stavu

Samotné komponenty [54] lze implementovat ve funkcionálním nebo třídním zápisu. Na první pohled je nejvýznamnějším rozdílem syntax, kde funkcionální komponenty jsou jednodušší na porozumění, jelikož se jedná o obyčejné JavaScriptové funkce, které vrací

React elementy (viz výpis 3.4). Třídní komponenty jsou zase JavaScriptové třídy, které rozšiřují třídu `React.Component` o implementaci metody `render()`, která opět musí vracet elementy Reactu (viz výpis 3.5).

```
const MyComponent = (props) => {
    return <h1>Hello, {props.name}</h1>;
};
```

Výpis 3.4: Funkcionální komponenta definovaná pomocí šipkové notace.

```
class MyComponent extends React.Component {
    render() {
        return <h1>Hello, {this.props.name}</h1>;
    }
}
```

Výpis 3.5: Ekvivalentní definice výpisu 3.4 v třídní komponentě.

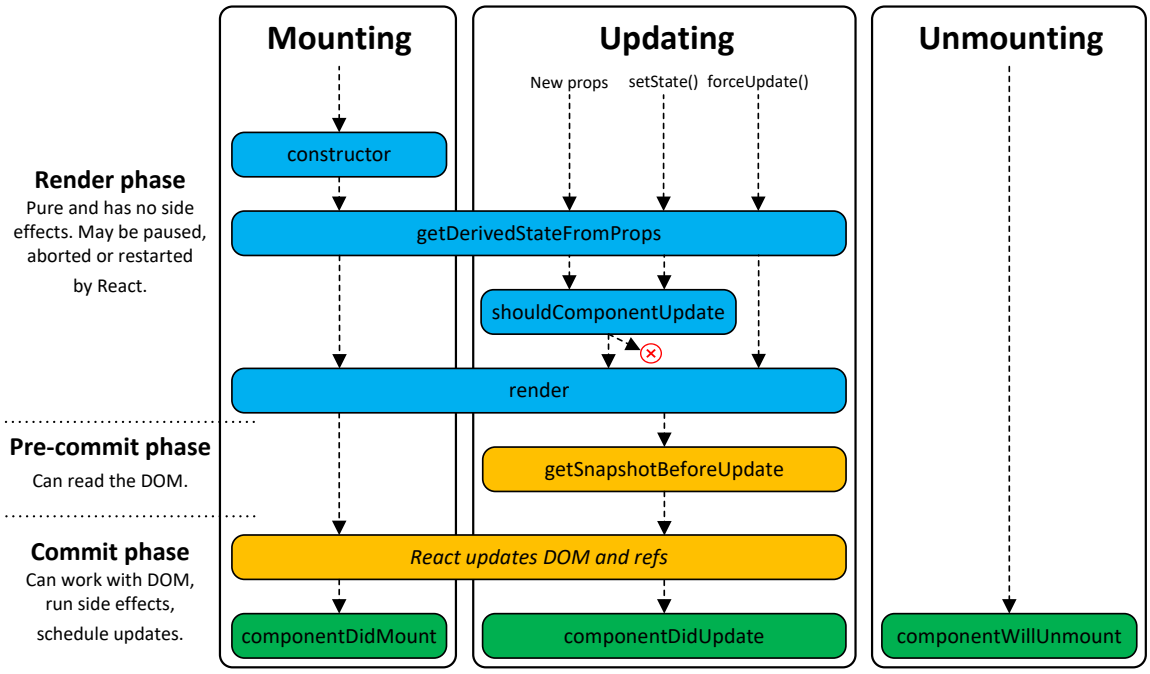
Dříve se funkcionální zápis používal pouze k definici bezstavových komponent a tyto komponenty poskytovaly vyšší výkon. Ovšem od verze 16.8 disponují funkcionální komponenty i React Hooky, což jim umožňuje využívat mechaniky jako jsou vstupy, stavy, reference, přístup k událostem z životního cyklu komponenty aj. Nyní jsou funkcionální komponenty stejně silné jako třídní komponenty (i z hlediska výkonu).

3.3.3 Životní cyklus komponent

Každá komponenta má vlastní životní cyklus, který lze definovat jako sérii událostí, které jsou vyvolávány v různých fázích její existence. Díky tomu je možné v tomto procesu pomocí metod definovat její chování. Tyto metody jsou také známé jako metody životního cyklu a využívají se v třídních komponentách (viz obrázek 3.4). Funkcionální zápis umožňuje přístup k těmto událostem pomocí React Hooků a nabízejí tak podobnou funkcionalitu. Každá komponenta [7, 25, 40] prochází následujícími třemi fázemi:

1. **Montáž** *(Mounting)* – je první fáze v životním cyklu, kde komponenta je přidána do DOM a poprvé vykreslena. Zde je zavolán její konstruktor, inicializován její stav a vstupy. Během této fáze jsou dostupné tyto metody:
   (a) `constructor()` – první zavolaná metoda předtím, než je komponenta namontována. Typicky se využívá pro inicializaci stavu a navázání metod pro obsluhu událostí na instanci.
   (b) `static getDerivedStateFromProps()` – tato metoda je zavolána vždy těsně předtím, než je komponenta vykreslena. Využívá se u vzácných případů, kdy stav komponenty závisí na změnách na vstupech.
   (c) `render()` – tato metoda vrací obsah, jenž bude vykreslen do prohlížeče (React elementy). Uvnitř této metody by nemělo docházet ke změně stavu komponenty ani k přímé interakci s prohlížečem.
   (d) `componentDidMount()` – metoda je zavolána ihned potom, co je komponenta namontována do DOM. Zde je možné dodatečně změnit stav komponenty, ještě předtím než bude aktualizováno uživatelské rozhraní, což sice způsobí dodatečné překreslení, ale uživatel neuvidí mezistav.

2. **Aktualizace** *(Updating)* – v této fázi se komponenta vyskytuje při překreslení způsobeného změnou stavu nebo vstupu.
   (a) `static getDerivedStateFromProps()` – metoda je zavolána vždy před překreslením komponenty (viz výše).
   (b) `shouldComponentUpdate()` – je pravdivostní metoda, která těsně před překreslením komponenty rozhoduje o tom, zda k němu dojde či nikoliv. Používá se k optimalizačním účelům a ve výchozím nastavení je návratová hodnota nastavena vždy na `true`.
   (c) `render()` – opět vykreslující metoda, jenž definuje podobu komponenty (viz výše).
   (d) `getSnapshotBeforeUpdate()` – tato metoda je zavolána předtím, než budou provedeny změny v reálném DOM. Umožňuje komponentě zachytit některé informace z virtuálního DOM o výsledném rozložení a slouží tak jako poslední kontrola předtím, než se samotná data opravdu vykreslí.
   (e) `componentDidUpdate()` – metoda je zavolána ihned po aktualizaci všech změn v reálném DOM.
3. **Odpojení** *(Unmounting)* – během této fáze dochází k odstranění komponenty ze struktury DOM.
   (a) `componentWillUnmount()` – metoda je invokována těsně před odpojením a následným zničením komponenty. Umožňuje tak dodatečně uvolnit alokované zdroje.



Obrázek 3.4: Životní cyklus komponent ve třech fázích. V každé fázi je zobrazena sekvence výskytu metod životního cyklu tak, jak na sebe chronologicky navazují. Převzato z článku [7].

3.4 Socket.io

Socket.io[14] [47] je JavaScriptová knihovna implementující vrstvu řízenou událostmi pro obousměrnou, stavovou komunikaci mezi klientem a serverem. Je postavena na knihovně nižší úrovně Engine.io[15] a abstrahuje přenos dat pomocí HTTP long-polling a WebSocketů do jediného API, jenž má širokou podporu napříč prohlížeči. Kromě běžné komunikace knihovna také poskytuje další užitečné vlastnosti jako je automatická obnova spojení při výpadku a ukládání paketů do vyrovnávací paměti, broadcastování všem klientům nebo jejich podmnožině, rozdělení logiky pomocí jmenných prostorů, podpora streamování binárních dat aj.

K vytvoření spojení [47] knihovna nejdříve používá dotazování pomocí xhr-pollingu, jakmile je toto spojení navázáno, tak použije nejlepší dostupnou metodu komunikace – ve většině případů právě WebSocket. Ve výchozím nastavení Socket.io před samotným vylepšením metody však ještě nějaký čas používá transport přes HTTP polling. Ukázalo se totiž, že komunikaci přes WebSockety není možné vždy úspěšně navázat navzdory své široké podpoře kvůli firemním proxy serverům, firewallům, antivirovým programům aj. Toto neúspěšné navázání se pak může promítnout až do 10-ti sekundového čekání, než započne výměna dat. Pro vylepšení metody klient musí provést následující:

1. Ujistit se, že jeho odchozí buffer je prázdný.
2. Nastavit aktuální přenos do režimu pouze pro čtení.
3. Pokusit se navázat přenos pomocí nové metody.
4. V případě úspěchu uzavřít původní spojení.

3.5 REST API

Representational State Transfer (REST) [32] je architektonický styl pro komunikaci mezi klientem a serverem. Obsahuje soubor pravidel, které definuje rozhraní API a způsob jakým klient přistupuje k datům. Data se označují jako zdroje a přístup k nim je umožněn přes endpointy, ke kterým uživatel přistupuje pomocí konkrétní URL. Poté co uživatel odešle na tento koncový bod svůj požadavek, je na serveru provedena příslušná operace a vrácena konkrétní odpověď. REST ke komunikaci nejčastěji používá následujících pět HTTP metod:

- **GET** – Poskytne přístup ke čtení zdroje.
- **POST** – Vytvoří nový zdroj.
- **PUT** – Aktualizuje existující zdroj jeho nahrazením.
- **PATCH** – Aktualizuje existující zdroj jeho částečnou modifikací.
- **DELETE** – Odstraní zdroj.

Požadavek [32] kromě adresy koncového bodu a metody, také obsahuje hlavičky a tělo. Hlavičky se používají k poskytnutí dodatečných informací s HTTP požadavkem jak klientovi, tak serveru například o formátu nesených dat. Existuje více druhů hlaviček a jsou

[14] Socket.io: https://socket.io
[15] Engine.io: https://github.com/socketio/engine.io

vždy v páru klíč-hodnota. Tělo neboli data pak obsahují informace, které daný požadavek s sebou nese. Tato část se však nepoužívá u metody `GET`. Samotný REST je bezstavový, což znamená, že na straně serveru není uložen žádný kontext či historie předchozí komunikace s klientem. Každý požadavek od klienta proto musí obsahovat všechny náležité informace, aby server byl schopný na něj správně reagovat.

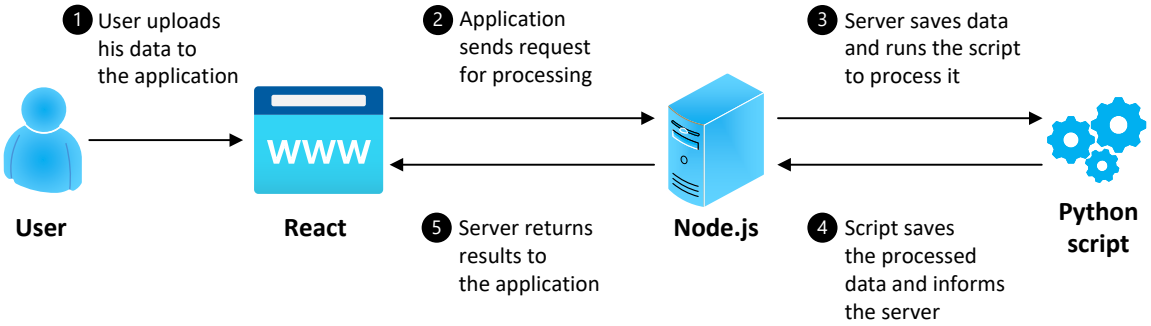
Kapitola 4

Návrh a implementace

Tato kapitola je zaměřena na návrh a implementaci aplikace pro detekci narušitelů. Na začátku je popsána obecná architektura řešení, z jakých částí se skládá a jak se mezi nimi data zasílají. V další části je uvedena výsledná podoba klientské části, která je doplněna o některé implementační detaily. Následující sekce je pak věnována implementaci serveru, která popisuje komunikaci s klientem, způsob, jakým se data ukládají a jak se s nimi následně pracuje. Na závěr této kapitoly je podrobněji sepsán proces pro zpracování videozáznamu, včetně detailů všech hlavních komponent, kterými disponuje.

4.1 Návrh architektury

Jak již bylo zmíněno v úvodní kapitole, tak aplikace je navržena typu klient-server ve webovém rozhraní. Uživatel prostřednictvím prohlížeče načte veškerá potřebná data a nakonfiguruje způsob zpracování videozáznamu. Data jsou následně odeslána na server, kde se celý záznam zpracuje. Zpracovaný záznam spolu s výpisem detekovaných objektů bude na konci pak vrácen zpět ke klientovi, který si ho bude moci přehrát či stáhnout. Celý návrh architektury je možné vidět na obrázku 4.1.



Obrázek 4.1: Návrh architektury internetové aplikace.

Klientská strana je implementována pomocí JavaScriptové knihovny React (viz sekce 3.3), jelikož je dobře optimalizována pro práci s často měnícími se daty, umožňuje snadnou dekompozici a také usnadňuje její implementaci. Uživatel v aplikaci po načtení svého záznamu pak nastaví podrobnosti ohledně jeho dalšího zpracování jako např. vymezí na záznamu sledovanou oblast, nahraje vlastní detektor nebo snímky obličejů pro identifikaci osob aj. Na konci této fáze budou data s konfigurací odeslána na server, kde proběhne zpracování.

Na straně serveru, postaveného na Node.js (viz sekce 3.1), jsou přijatá data dočasně uložena tak, aby k nim byl umožněn přístup skriptu, který celé video zpracuje. Důležité v tomto procesu je rozlišovat data od klientů a ukládat je odděleně tak, aby nedocházelo ke konfliktům a žádná data nijak nezasahovala mezi data jiného klienta. To by při současném zpracování více záznamů mohlo v důsledku zapříčinit, že by na konkrétním záznamu mohly být rozpoznávány osoby i podle snímků obličeje, které nahrál jiný uživatel.

Skript pro zpracování videa je implementován v pythonu a zpracování probíhá po jednotlivých snímcích. Na každém snímku jsou jako první detekovány a následně klasifikovány objekty, jež mohou být považovány za narušitele pomocí detektoru YOLO (viz sekce 2.3.2). Pokud narušitelem bude osoba, tak dojde k její identifikaci pomocí systému DeepFace (viz sekce 2.5.1) vůči databázi se snímky obličejů nahraných od uživatele. Každý snímek obličeje obsahuje jméno osoby, které v případě shody bude zobrazeno na videu. Ostatní detekované objekty jsou pak označeny pouze jejich třídou, confidence skóre, případně unikátním identifikátorem při nastaveném sledování objektů.

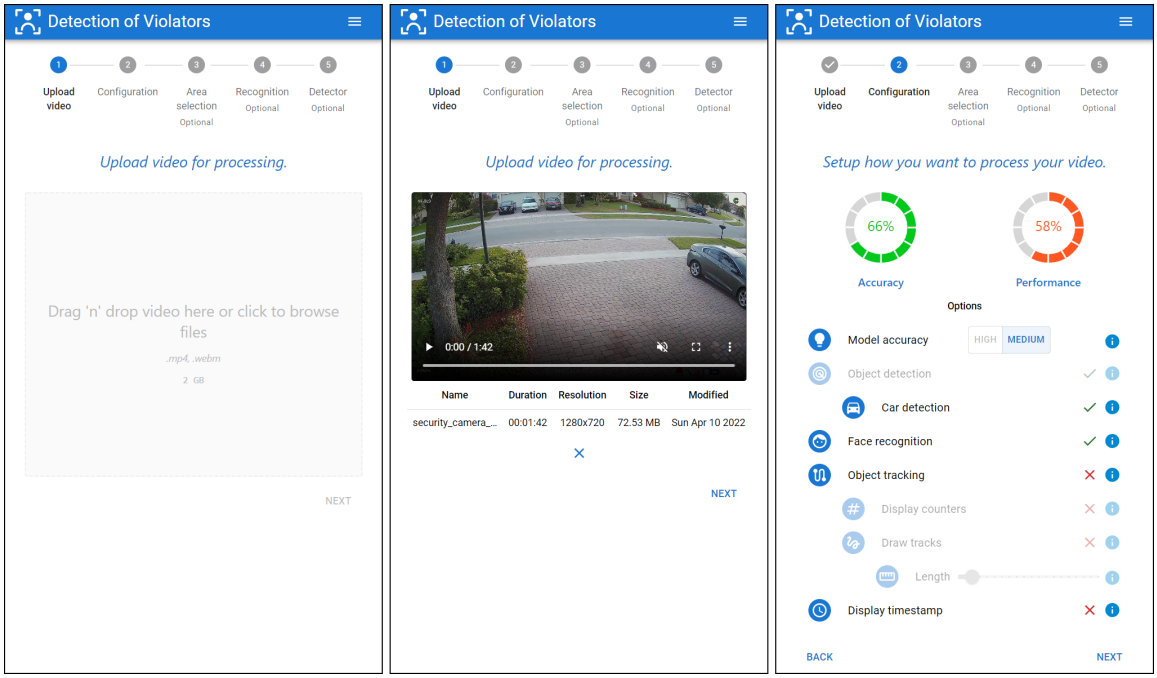
Během procesu zpracování jsou uživateli pravidelně zasílány informace o aktuálním postupu pro predikci zbývajícího času. Nakonec jsou data zaslána zpět do klientské části, kde je k dispozici zpracované video spolu se souhrnem výsledků s detekovanými objekty a zaznamenanými časy.

4.2 Návrh a implementace aplikace

Aplikace je rozdělena na řadu jednotlivých kroků, kterými musí uživatel postupně projít, než bude moci svůj záznam úspěšně odeslat ke zpracování. Tento způsob zejména usnadňuje orientaci v aplikaci tím, že každý krok přímo popisuje, jaká akce se od uživatele očekává a celý proces nepůsobí tak chaoticky. Zároveň je tímto způsobem ošetřené pořadí některých úkonů, které nelze zaměnit, např. výběr sledované oblasti není možný bez toho, aniž by byl předtím nahraný záznam. Celý proces konfigurace je zároveň u jednotlivých operací doplněný poznámkami a dodatečnými popisky.

V prvním kroku musí uživatel nahrát svůj záznam (viz obrázek 4.2), tato část je povinná a není možné bez ní pokračovat. Poté co uživatel vloží video, zobrazí se v přehrávači včetně jeho metadat a povolí se postup do dalšího kroku. V druhém kroku uživatel navolí konfiguraci, jak má být jeho záznam zpracován. Podstatou celého zpracování je detekce potencionálních narušitelů, proto detekci objektů nelze vypnout a slouží tak spíše jako symbol indikující, že detekce proběhne vždy, i když uživatel zakáže ostatní možnosti. Lze však dodatečně deaktivovat detekci vozidel pro případ, kdy by byl záznam pořízen z oblasti, ve které se vozidla běžně vyskytují a nelze je tak považovat za „podezřelé“ jevy (např. kamerový záznam z domovní příjezdové cesty, kde obyvatelé obvykle parkují nebo záznam z firemního areálu, které zároveň slouží jako parkoviště). Dále u samotné detekce je možné nastavit její přesnost. Tato možnost definuje, v jakém rozlišení budou snímky vkládány do neuronové sítě, která provádí detekci. Vyšší rozlišení tak zajistí přesnější ohraničení objektů, ovšem znatelně prodlouží dobu na zpracování celého záznamu.

Další dostupnou volbou, která má vliv na dobu zpracování je rozpoznávání osob podle obličejů. V tuto chvíli dojde pouze k aktivaci tohoto modulu, ale na nahrání snímků dojde až v pozdější části celého procesu. Poslední volbou ovlivňující výkon je sledování objektů (alias *tracking*). Tento modul ke každé detekci přiřadí unikátní identifikátor a umožní k nim ukládat dodatečné informace napříč snímky. Tím se pak povoluje možnost i jednotlivé objekty na snímku počítat nebo vykreslovat za nimi cesty a zobrazit tak trasu odkud přišly. U cest je zde také možnost specifikovat její délku v sekundách, po které začne opět



Obrázek 4.2: Ukázka aplikace, kde vlevo a uprostřed je zobrazen první krok pro nahrání videa a napravo se nachází nastavení konfigurace pro zpracování.

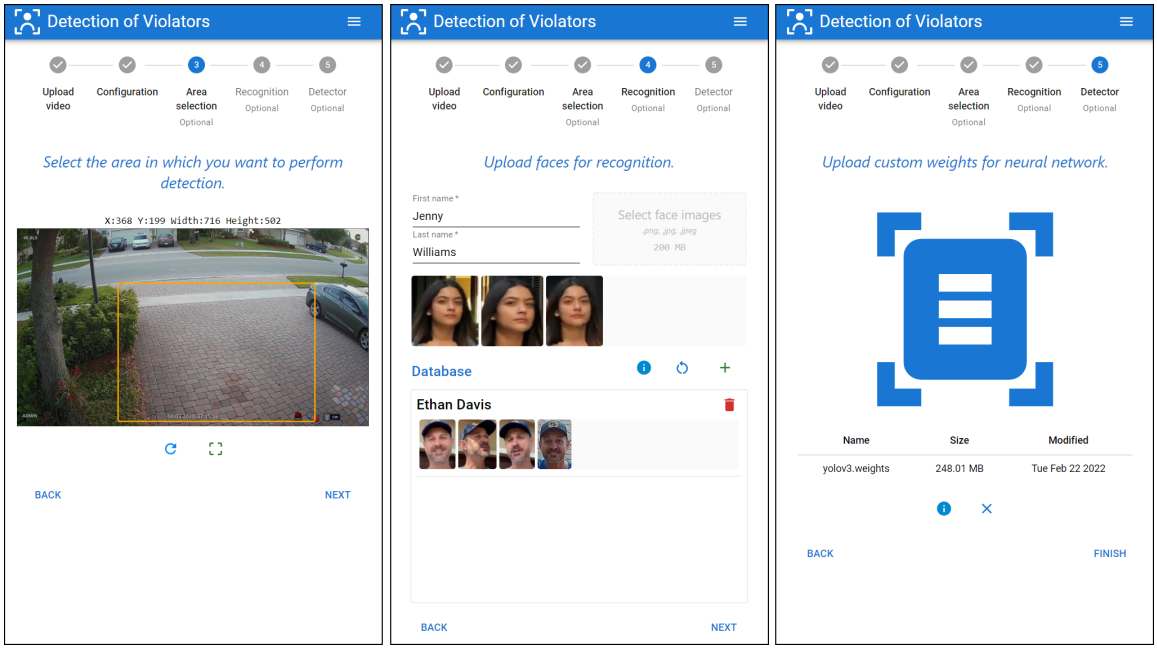
mizet. Sledování objektů v kombinaci s rozpoznáváním obličejů umožní po identifikaci osoby udržet její jméno zobrazené i po tom, co osoba zahalí svůj obličej. Poslední parametr pak umožňuje na každý snímek vypsat aktuální čas na záznamu.

V následujícím kroku může uživatel na záznamu vymezit sledovanou oblast a tím zamezit detekci objektů, které se nachází mimo ni (viz obrázek 4.3). Zobrazený snímek pro vymezení oblasti je vygenerován náhodně při vložení záznamu a lze ho případně kdykoliv změnit, pokud by z nějakého důvodu nevyhovoval. Oblast je pak reprezentována souřadnicemi levého horního rohu, svojí šířkou a výškou (minimálně však 50×50 pixelů). Tato oblast je definovaná pro všechny snímky a je možné i nastavit, aby byla či nebyla znázorněna na zpracovaném záznamu.

V předposledním kroku je uživateli umožněno nahrát snímky obličejů, které budou vloženy do databáze a použity pro rozpoznávání detekovaných osob. Počet snímku není nijak omezený, ale je doporučeno nahrát ke každé osobě aspoň pět snímků tváří včetně čelního pohledu, kde každý snímek je pootočený o 45 stupňů. U každé osoby je kromě snímků nutné zadat její jméno a příjmení, které bude v případě identifikace vykresleno uvnitř odpovídajícího bounding boxu. Duplicity jmen jsou povoleny, protože interně je každá osoba reprezentována unikátním identifikátorem.

Poslední krok umožňuje nahrát vlastní váhy pro neuronovou síť YOLOv3, které se použijí na zpracování daného záznamu. Důležité je, aby váhy byly dotrénovány na již předtrénovaných vahách na datasetu COCO[1] a umožnily tak detekci všech základních 80-ti tříd. Pokud budou nahrány nekompatibilní váhy, aplikace informuje uživatele o chybě. Tato možnost je tu pro případ, kdyby uživatel disponoval lepším detektorem pro danou scénu na záznamu, mohl by tak zlepšit přesnost detekcí. Původně bylo v aplikaci plánováno nahrát pouze trénovací, anotovaná data a dotrénovat detektor na straně serveru. Ovšem tento pří-

[1]COCO: https://cocodataset.org

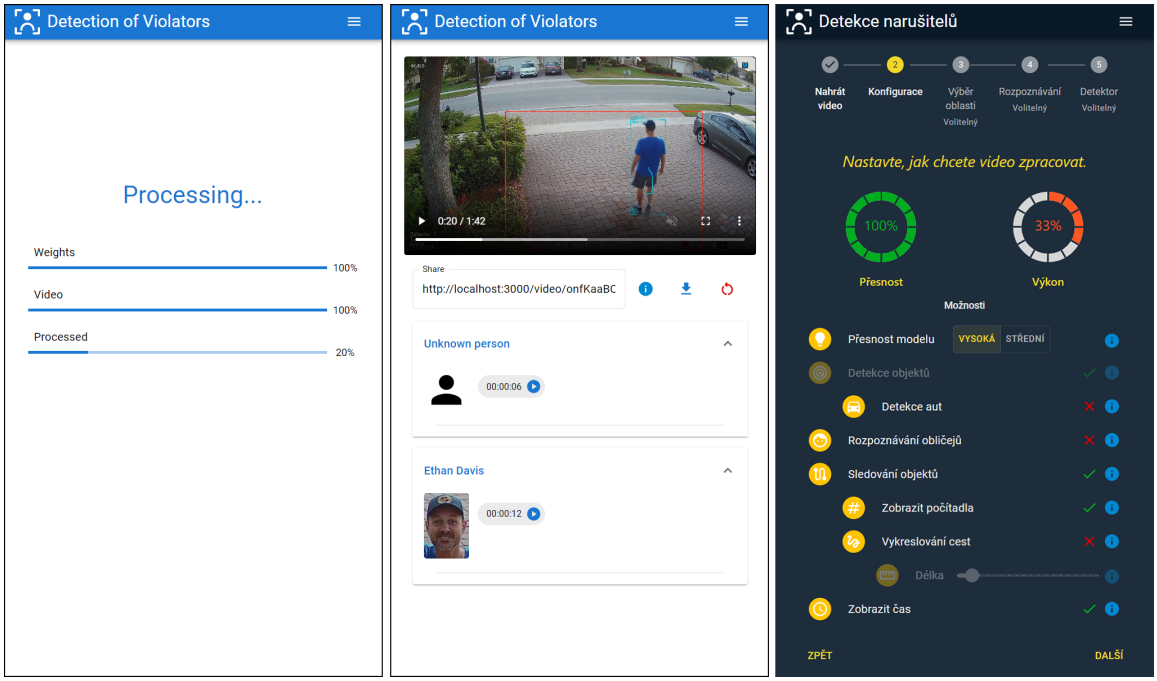


Obrázek 4.3: Vlevo se nachází krok v aplikaci pro zvolení detekční oblasti, uprostřed je krok pro nahrání snímků obličeje do databáze a vpravo je zobrazen poslední krok s nahranými váhami neuronové sítě.

stup by znatelně zatěžoval server na dlouhou dobu a tím by se pak omezil výkon i ostatním uživatelům. Uživatel po tomto kroku může odeslat data na server a zahájit zpracování.

Během zpracování je uživatel informován o aktuálním postupu pomocí ukazatelů průběhu (viz obrázek 4.4). Poté co je video úspěšně zpracováno, je uživatel automaticky přesměrován na další obrazovku s výsledky. Výsledná obrazovka obsahuje zpracované video, které je možné ihned přehrát ve webovém přehrávači nebo ho stáhnout. Níže se nachází seznam nalezených objektů, kde každá záložka představuje konkrétní třídu objektu a po jejím rozbalení jsou zobrazeny časy, ve kterých byl objekt na záznamu spatřen. Po kliknutí na daný čas pak bude video automaticky přetočeno na tuto detekci. Osoby, pro které byly nahrány snímky obličejů a byly na záznamu identifikovány je také vytvořena speciální záložka, která kromě časů obsahuje zároveň i seznam snímků obličejů, vůči kterým byla detekce rozpoznána. Výsledky je možné sdílet pomocí přiloženého odkazu po dobu 24 hodin, poté budou data ze serveru trvale smazána.

Aplikace je implementována v Reactu (viz sekce 3.3) v řadě komponent, které zejména kvůli jejich množství nemá význam popisovat. Nicméně veškerá data, která uživatel nahraje či nakonfiguruje jsou uložena v kontextu dat v souboru `app/src/utils/DataProvider.js`, která jsou po posledním kroku odeslána na server. Odeslání dat je provedeno pomocí socketu, který je implementovaný v souboru `app/src/utils/WsProvider.js`. Napojení těchto API (ale i dalších) je provedeno v souboru `app/src/Root.js`. Pro sdílení výsledků se v aplikaci používá cesta `/video/ID`, kde `ID` je unikátní identifikátor zpracovaného videa, který vygeneroval server. Aplikace je dostupná v češtině i angličtině a implementuje i tmavý motiv. Uživatelské preference se pak ukládají do cookies po dobu 7 dní.

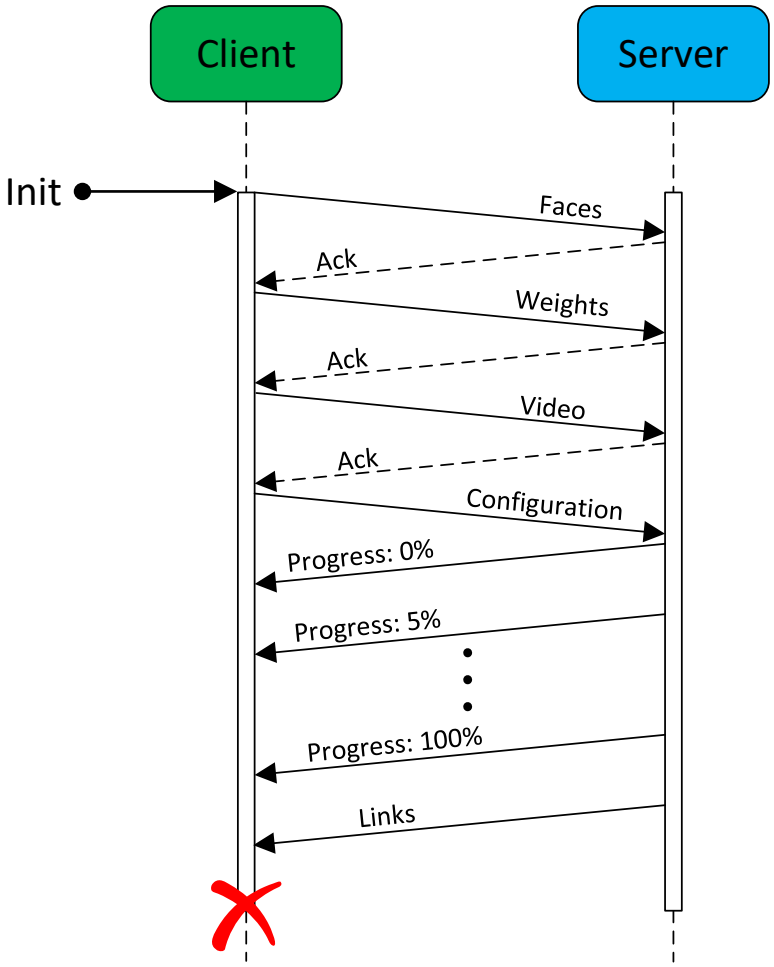


Obrázek 4.4: Vlevo se nachází obrazovka aplikace, která informuje uživatele o aktuálním průběhu během zpracování. Uprostřed se nachází výsledná obrazovka se zpracovaným záznamem a souhrnem o detekovaných objektech. Vpravo je pak ukázka české lokalizace a tmavého motivu aplikace.

4.3 Implementace serveru

Server je implementovaný v prostředí Node.js (viz sekce 3.1) a kombinuje technologie Socket.io (viz sekce 3.4) a Express.js (viz sekce 3.2). Po spuštění server vytvoří adresáře `tmp` a `videos`, které následně poslouží ke zpracování videozáznamů. Každý klient, jenž se k serveru připojí pak komunikuje nejdříve prostřednictvím Socket.io. Každému spojení je přiřazen unikátní identifikátor, který se mimo jiné používá i pro rozlišení dat napříč klienty. Ihned po navázání spojení se klientovi vytvoří dočasný adresář pojmenovaný podle jeho identifikátoru v adresáři `tmp`, do kterého se dále budou ukládat jeho soubory. Uvnitř tohoto adresáře se potom vytvoří adresář `database`, který případně poslouží modulu pro identifikaci osob k vyhledání snímků obličejů, pokud ho uživatel aktivuje ve své konfiguraci.

Výměna dat v rámci socketu probíhá ve třech tématech *(topics)*, kde ze začátku dochází k nahrání snímků obličejů, poté vah neuronové sítě pro detekci objektů a na konec videozáznamu – vždy v tomhle pořadí. Nicméně snímky a váhy jsou volitelné a lze tedy tyto kroky přeskočit. Celý proces komunikace je pak znázorněn na obrázku 4.5. Nahraní snímků se provádí postupně za sebou a jsou ukládány přímo do konkrétního adresáře `database` pod názvem `jméno_přijmení_idOsoby_idSnímku`. Tento formát umožňuje transparentní přenos informací o snímku až do skriptu, který provede zpracování videa, a proto není nutné vytvářet separátní strukturu s informacemi o nahraných snímcích. Formát je dostupný i na straně klienta a slouží tak i ke zpětnému dohledání rozpoznaných obličejů. Díky tomu, lze vrátit pouze určitý soupis identifikátorů, podle kterých pak klient vyhledá odpovídající obličeje a není tak nutné je znovu posílat nazpět.



Obrázek 4.5: Komunikace mezi klientem a serverem prostřednictvím Socket.io.

Poté, co jsou nahrány všechny snímky je klientská strana o této skutečnosti informována a začne posílat soubor s váhami. Váhy se uloží přímo do dočasného adresáře vedle databáze. Během přenosu je klient informován o množství přenesených dat, která jsou pak znázorněna pomocí ukazatele. Jakmile jsou váhy kompletně nahrány, tak si k nim server uloží cestu a informuje klienta o dokončeném přenosu. Ten následně začne posílat videozáznam. Celý proces přenosu probíhá stejně jako v případě vah a klient je na konci opět informován o úspěšném dokončení akce.

Následně co klientská strana obdrží zprávu o dokončeném přenosu videozáznamu, tak odešle data s konfigurací, kterou uživatel nastavil v předchozích krocích. Potom co tuto konfiguraci obdrží server, tak ji převede na argumenty python skriptu. Program obsahuje celkem 15 argumentů a většina z nich je převedena pomocí funkce `parseArgsCLI()` v souboru `utils.js`. Kromě uživatelské konfigurace je totiž nutné taky zadat cesty k nahraným souborům, které si server postupně ukládal. Jakmile je konfigurace převedena na parametry, tak se vytvoří nový proces, který s těmito parametry spustí program. Vytvoření speciálního procesu pro tento úkon má zásadní vliv na chod serveru, protože celý běh serveru je řízen pouze jedním vláknem a v opačném případě by došlo k jeho zablokování a znemožnění konkurentního zpracování (viz sekce 3.1 o Node.js).

Během toho, co nový proces zpracovává video, tak vypisuje na standardní výstup aktuální postup zpracování. Tento výstup je zachytáván hlavním procesem a přeposílá ho do klientské části, kde je následně zobrazen. Uživatel je tak neustále informován o aktuálním průběhu. Poté co skript skončí, tak se ověří jeho návratová hodnota. Návratový kód 0 indikuje, že video bylo úspěšně zpracováno a skript uložil zpracovaný záznam do adresáře `videos` spolu se souborem JSON, ve kterém jsou zaznamenány informace o detekovaných objektech. Uživatel je informován o dokončeném zpracování a jsou mu odeslány odkazy, pod kterými tato data najde. Pokud skript skončil s nenulovou hodnotou, tak je uživatel informován o chybě (může způsobit např. špatný formát vah pro neuronovou síť). Nakonec je celý dočasný adresář klienta odstraněn pomocí funkce `rmDirRecursive()` implementované v souboru `utils.js`, protože nadále již není potřeba a klient je od serveru odpojen.

Výsledné dva soubory jsou tedy uloženy v adresáři `videos` pod identifikátorem klienta. V této části přichází na řadu Express.js, který je zpřístupní klientovi pomocí běžných bezstavových HTTP metod. Express implementuje v adresáři `routes` následující tři cesty:

- **`video`** – Cesta, která pomocí parametru `ID` vrátí klientovi zpracovaný videozáznam, jenž je použit v aplikaci pro přehrání.
- **`download`** – Cesta, která pomocí parametru `ID` umožní klientovi stáhnout zpracovaný záznam.
- **`data`** – Cesta, která pomocí parametru `ID` vrátí klientovi data s detekovanými objekty ve formátu JSON, která jsou uživateli zobrazena jako souhrn.

Na výpisu 4.1 je znázorněna reálná ukázka obsahu JSON souboru s informacemi o detekovaných objektech na videu. Soubor obsahuje objekt, kde každý detekovatelný objekt je reprezentován odpovídajícím klíčem. Každý klíč má hodnotu pole, do kterého se ukládají časy v sekundách, ve kterých došlo k detekci daného objektu. Detekce osob má však jiný formát, protože je nutné odlišit rozpoznané osoby od těch ostatních.

U osob se do pole místo čísel ukládají další objekty, které reprezentují konkrétní osobu. Každá osoba má unikátní identifikátor, jméno a pole s detekcemi, které obsahuje objekty reprezentující konkrétní detekci na záznamu. Objekt detekce pak obsahuje identifikátor, který odpovídá identifikátoru snímku obličeje, vůči kterému byla osoba rozpoznána a položku s časy, při kterých k této události došlo. Pokud osoba bude rozpoznána vůči dvěma snímkům, potom pole s detekcemi bude obsahovat dva objekty. V případě, kdy dojde k opětovnému rozpoznání vůči snímku, který je zde již zaznamenán, tak se přidá pouze další čas do odpovídajícího objektu podle `ID`. Osoba, jejíž `ID` je prázdný řetězec pak reprezentuje ostatní osoby, které nebyly rozpoznány a obsahuje právě jednu detekci pro zaznamenání časů. Důležité je poznamenat, že čas bude uložen pouze pokud k dané detekci došlo naposledy nejméně před 5-ti sekundami, jinak by docházelo k nadbytečnému ukládání dat či redundancím.

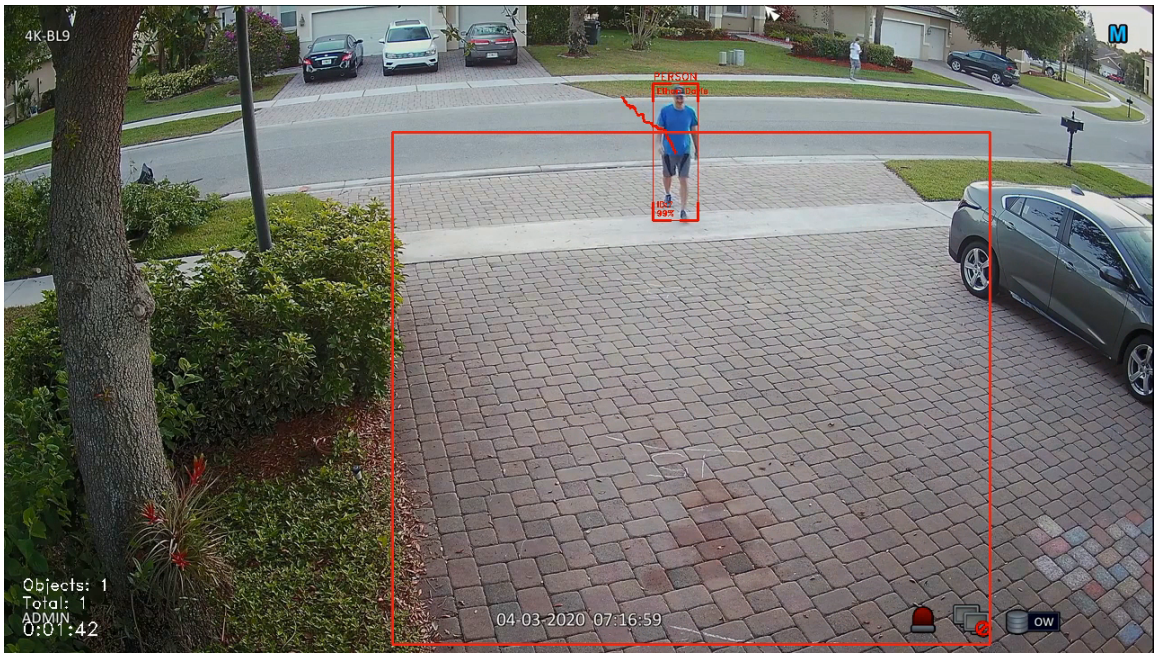
Zpracovaná data jsou na serveru dostupná po dobu 24 hodin, než dojde k jejich odstranění. O tuto činnost se stará funkce `deleteFiles()` implementovaná v souboru `utils.js`, která je volána každou hodinu a prochází adresář `videos`. V adresáři postupně kontroluje metadata každého souboru a pokud je soubor starší více než 24 hodin, tak ho smaže.

```
{
  "person": [
    {
      "id": "",
      "name": "Unknown",
      "detections": [
        {
          "id": "",
          "last": 1,
          "timestamp": [
            0
          ]
        }
      ]
    },
    {
      "id": "8",
      "name": "Jenny Williams",
      "detections": [
        {
          "id": "6",
          "last": 1,
          "timestamp": [
            0
          ]
        }
      ]
    }
  ],
  "car": [
    0
  ],
  "cat": [],
  "dog": [],
  "horse": [],
  "sheep": [],
  "cow": [],
  "elephant": [],
  "bear": [],
  "zebra": [],
  "giraffe": []
}
```


Výpis 4.1: Ukázka obsahu JSON souboru s informacemi o detekovaných objektech ve videu.

4.4 Zpracování videozáznamu

Po spuštění programu pro zpracování videa dojde nejdříve k inicializaci objektů `Detector`, `Recognizer`, `Tracker` a `Recorder`, které budou následně zodpovědní za detekci objektů, rozpoznávání osob, sledování a zaznamenávání detekcí na záznamu. Celý proces zpracování je abstraktně znázorněn na konci této sekce v algoritmu 1 a ukázku zpracovaného snímku lze vidět na obrázku 4.6. Objektu třídy `Detector` je při inicializaci nutné předat seznam s třídami detekovatelných objektů a specifikovat cestu ke konfiguračnímu souboru a předtrénovaným vahám YOLOv3 (viz sekce 2.3.2). V adresáři `yolov3` jsou uloženy dva konfigurační soubory, které se liší v rozlišení snímků, které budou vkládány do neuronové sítě. Lze vybrat mezi dvěma rozlišeními – 320×320 a 608×608, jaká konfigurace se však použije rozhoduje uživatel argumentem programu (výchozí je 320×320).



Obrázek 4.6: Ukázka snímku ze zpracovaného videa, na kterém se nachází detekována osoba, jenž vchází do sledované oblasti na příjezdové cestě. Za osobou je vykreslená stopa, která zobrazuje trasu odkud přišla a zároveň je zobrazeno její jméno, které bylo zjištěno při rozpoznání obličeje v předchozí části videa.

Při inicializaci objektu třídy `Recognizer` je nutné definovat cestu k adresáři se snímky obličejů, které mají být použity pro identifikaci. Rozpoznávání je prováděno pomocí systému DeepFace (viz sekce 2.5.1) na modelu Facenet. `Tracker` je implementovaný na algoritmu DeepSORT (viz sekce 2.6.2), který je uložený v adresáři `deep_sort` a při jeho inicializaci se pouze nastavují parametry jako maximální stáří sledovaných objektů, práh shody, doba mezi aktualizacemi měření a další. Hodnoty těchto parametrů byly testovány na různých záznamech a napevno nastaveny tak, aby došlo k rovnováze mezi častou záměnou identifikátorů a jejich chybnému přiřazení. Objekt třídy `Recorder` se používá pouze k zaznamenávání detekovaných objektů a při inicializaci je nutné mu předat seznam detekovatelných tříd, podle kterých vytvoří strukturu, do které bude zaznamenávat časy.

Poté co jsou inicializovány nejdůležitější komponenty pro zpracování videa, tak se otevře zdrojové video a získají se z něj metadata, která se použijí pro zápis zpracovaného videa, výpočtu aktuálního času aj. Cílové video bude uloženo do adresáře specifikovaného v argumentu programu ve formátu `.mp4` s kodekem H.264[2]. Tento kodek byl vybrán kvůli jeho široké podpoře napříč prohlížeči a při zobrazení výsledného videa ve webovém přehrávači by tedy neměl nastat žádný problém. Rozlišení videa a jeho snímková frekvence je pak stejná jako u zdrojového videa, ačkoliv by mohlo dojít ke značné redukci za účelem snížení časové náročnosti. Zachování těchto vlastností je ze strany uživatele na první pohled očekávané a z toho důvodu nedochází k žádné modifikaci.

Celý proces zpracování běží ve smyčce přes všechny snímky ve videu – viz algoritmus 1. Nad každým snímkem se nejdříve zavolá funkce `predict()` objektu `detector`, která vrátí třídy, bounding boxy a confidence skóre všech nalezených objektů. Funkce jako první převede snímek s přehozeným červeným a modrým kanálem barev na *blob*, který následně vloží na vstup neuronové sítě. Z výstupních vrstev YOLOv3 se poté posbírají predikce, které se vyfiltrují pomocí techniky Non-Maximum Suppression za účelem eliminace překrývajících se bounding boxů, které ohraničují ten samý objekt. Z vyfiltrovaných detekcí se pak vytvoří seznam objektů třídy `Detection`, jenž reprezentuje konkrétní detekci na snímku a uchovává její data.

Pokud je aktivován modul pro rozpoznávání osob, tak u každé detekce, která byla klasifikována jako osoba dojde k vyříznutí odpovídajícího bounding boxu ze snímku a předá se funkci `find()` objektu `recognizer`. Funkce provede identifikaci a vrátí jméno osoby, naměřenou vzdálenost, identifikátor osoby a obličeje. Jméno a identifikátory jsou získány z názvu snímku obličeje, podle kterého byla osoba rozpoznána (viz formát uložených snímků v sekci 4.3). Pokud nedojde k žádné shodě, tak je jméno osoby nastaveno na *Unknown* a naměřená vzdálenost je rovna 1. Tato data se pak přiřadí k detekci pomocí funkce `setIdentity()`.

Následuje modul pro sledování objektů, který se aktivuje pouze v případě, pokud byl nastavený odpovídající argument programu. Celý proces je proveden ve funkci `track` objektu `tracker`, která bere jako parametry aktuální snímek a seznam detekcí. Zde je ze začátku nutné převést objekty ze třídy `Detection` na objekty reprezentující detekci, které používá DeepSORT. Ten kromě základních údajů také vyžaduje reprezentaci detekce pomocí mapy příznaků, kterou používá ke sledování (porovnání) objektů napříč snímky. Zároveň je žádoucí také uchovat informace o identitě osoby, barvy bounding boxu aj. Z těchto důvodů bylo nutné upravit následující soubory DeepSORTu:

- **`detection.py`** – Třída reprezentuje detekci. Při inicializaci byly přidány parametry a atributy pro uchování dalších dat.
- **`track.py`** – Implementuje třídu, která reprezentuje danou stopu. Opět je zde potřebné přidat při inicializaci parametry a atributy pro uchování dat. Ve funkci `update()`, která přijímá jako parametr novou detekci byla doplněna logika pro aktualizaci atributů během existence daného objektu. Např. confidence skóre a třída se pokaždé aktualizují, ale identita osoby se přepíše pouze pokud nová identita obsahuje lepší naměřenou vzdálenost a zároveň pokud je splněný minimální práh pro aktualizaci.
- **`tracker.py`** – Ve třídě `Tracker` je na konci pouze doplněné předávání dat z detekce do stop při její inicializaci.

[2]OpenH264: https://github.com/cisco/openh264

Potom co DeepSORT spočítá z detekcí nový stav stop, tak se všechny potvrzené stopy převedou opět na původní objekty třídy `Detection` a budou navíc doplněné o identifikátor a seznam středových bodů předchozích bounding boxů, jenž slouží pro vykreslování cest. To se provádí jednoduchým propojením bodů úsečkami od aktuální pozice postupně ke starším bodům. Počet propojených bodů je specifikován argumentem programu, který definuje po jaké době se cesty už nemají vykreslovat. Je tedy nutné počet bodů spočítat pomocí tohoto argumentu v závislosti na snímkové frekvenci videa.

V tomhle okamžiku jsou získána všechna data o objektech na aktuálním snímku. Před samotným vykreslením je ale potřeba ošetřit, jestli detekované objekty patří mezi objekty, které mají být na snímku vykresleny, protože předtrénovaná síť YOLO na datasetu COCO detekuje 80 různých tříd. Kromě toho takové automobily se detekují potom, co je nastavený odpovídající argument programu. Dále také program umožňuje specifikovat detekční oblast a do této doby se detekce prováděla na celém snímku. Je tedy nezbytné zároveň ošetřit, jestli detekce, která má být vykreslena, se nachází uvnitř této oblasti porovnáním hraničních hodnot oblasti a souřadnic středového bodu bounding boxu.

Detekce se po vykreslení zaznamenají do objektu třídy `Recorder`, který na konci procesu vygeneruje souhrn všech nálezů ve formátu JSON a jeho formát je popsán v předchozí sekci 4.3. V poslední části, pokud bylo nastavené počítání objektů, se na snímek vykreslí počítadlo s přítomnými objekty, jehož hodnota je dána počtem aktuálních detekcí, jejichž třída patří mezi detekovatelné objekty a nachází se uvnitř detekční oblasti. Hodnota počítadla s celkovým počtem objektů je pak dána mohutností množiny, do které se postupně ukládají identifikátory těchto detekcí. Pokud je nastaveno i vykreslení detekční oblasti nebo času na snímek, tak dojde k jejich vykreslení. Tento proces se následně opakuje na všech snímcích ve videu.

Algorithm 1 Proces zpracování videa

```
1:  procedure ProcessVideo(args, videoPath, destPath, detectableObjects)
2:      detector ← YOLOv3                                          ▷ Inicializace
3:      recognizer ← DeepFace
4:      tracker ← DeepSORT
5:      recorder ← {object: list() for each object ∈ detectableObjects}
6:      video ← videoReader(videoPath)                             ▷ Načtení videa
7:      output ← videoWriter(destPath)
8:
9:      for each frame ∈ video do                                  ▷ Zpracování každého snímku
10:         classes, confs, bboxes ← detector.predict(frame)       ▷ Detekce
11:         detections ← list()
12:         for each class, conf, bbox ∈ zip(classes, confs, bboxes) do
13:             detections.append(Detection(class, conf, bbox))
14:         if args.recognition == True then                       ▷ Identifikace
15:             for each detection ∈ detections do
16:                 if detection.class == "person" then
17:                     crop ← imageCrop(frame, detection.bbox)
18:                     identity, faceDist, personId, faceId ← recognizer.find(crop)
19:                     detection.setIdentity(identity, faceDist, personId, faceId)
20:         if args.tracking == True then                          ▷ Sledování objektů
21:             detections ← tracker.track(frame, detections)
22:         for each detection ∈ detections do                     ▷ Vykreslení detekcí
23:             if detection.class == "car" ∧ args.cars == "False" then
24:                 continue
25:             if detection.class ∈ detectableObjects then
26:                 if args.area == None ∨ detection.bbox ∈ args.area then
27:                     drawDetection(frame, detection)
28:                     recorder.add(detection)
29:         if args.area ≠ None ∧ args.frame == True then          ▷ Vykreslení oblasti
30:             drawRect(frame, args.area)
31:         if args.tracking == True ∧ args.counters == True then  ▷ Počítadla
32:             drawCounters(frame, detections, args.area)
33:         if args.timestamp == True then                         ▷ Vykreslení aktuálního času
34:             drawTime(frame, video.position, video.fps)
35:     recorder.saveJSON(destPath)
```

Kapitola 5

Experimenty a vyhodnocení systému

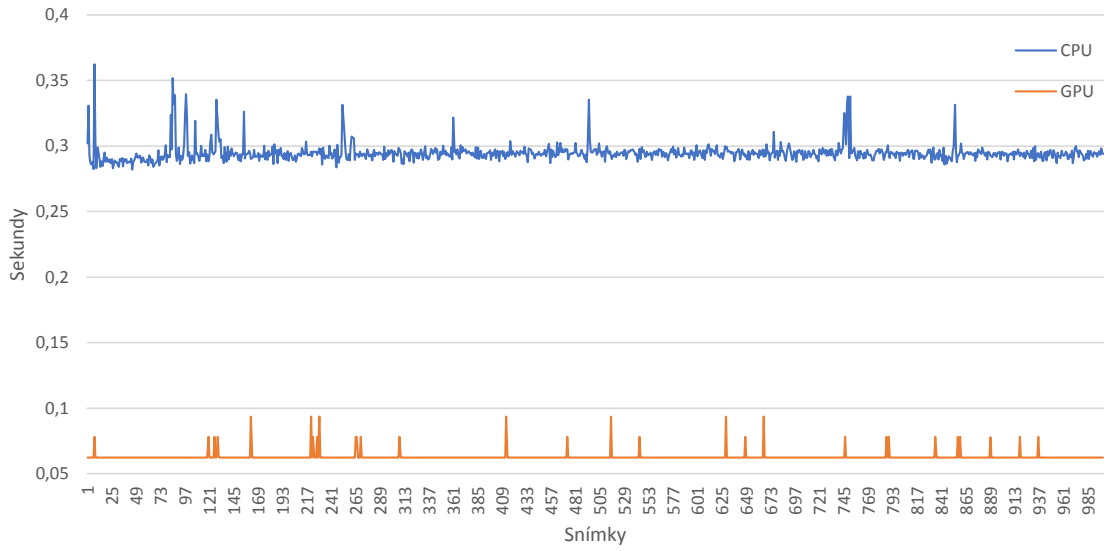
V poslední části bylo provedeno měření za účelem vyhodnocení kvality a použitelnosti celého systému. Veškerá dále uvedená data byla naměřena na reálných záznamech z kamer na sestavě, která disponovala procesorem značky Intel Core i7-7700HQ s frekvencí 2, 80 GHz a operační pamětí 8 GB. Kromě toho tato kapitola obsahuje popis, podle kterého byl při implementaci zohledněn výběr detektoru z pohledu jeho přesnosti.

5.1 Vyhodnocení výkonnosti

Výkonností řešení se v tomto kontextu myslí potřebný čas ke zpracování jednoho záznamu, jelikož právě to nejvíce ovlivňuje uživatele a použitelnost celého systému v praxi. Existuje mnoho faktorů, které tuto dobu ovlivňují, jako je např. množství aktivovaných operací v konfiguraci, snímková frekvence videa nebo hardwarové vybavení serveru. V předchozí kapitole 4 bylo řečeno, že mezi hlavní komponenty, které majoritně ovlivňují dobu zpracování patří detekce, identifikace osob a sledování objektů, proto v rámci měření byly tyto časy zaznamenávány odděleně.

Doba k provedení detekce je dána výkonností neuronové sítě YOLOv3 a na neštěstí je značně ovlivněna počtem detekovatelných objektů, přičemž značná část nemá v řešení využití. Na druhou stranu potřebný čas k provedení této operace při rozlišení 320×320 je poměrně malý, tj. průměrně ~294 ms na CPU. Čas je dále možné značně zredukovat akcelerací pomocí GPU, kde průměrný čas na snímek dosahuje ~63 ms na grafické kartě GTX 1050Ti s pamětí 4 GB a v tomto případě bylo tak dosaženo ~$4, 7$násobného zrychlení (viz graf 5.1).

Identifikace osob je provedena pouze v případě, pokud byla osoba v první řadě na snímku detekována. Nelze tedy říci, že tento modul bude zatěžovat celý průběh zpracování záznamu, protože v případě, kdy se na snímku nenachází žádná osoba, tak dochází k nulovému zpoždění. Přičemž se dá očekávat, že na skutečném záznamu, ve kterém se vyskytují narušitelé jen výjimečně, tak nebude docházet k aktivaci příliš často. Nicméně průměrný čas pro identifikaci osoby pomocí systému DeepFace z naměřených hodnot na CPU zabere ~60 ms (viz graf 5.2). Jelikož jsou osoby rozpoznávány jednotlivě, tak výsledný čas také záleží na množství osob, které se na daném snímku nachází, tedy v případě čtyř osob bude pak celkové zpoždění $4 \cdot 60 = 240$ ms.



Obrázek 5.1: Časy pro detekci objektů na 1000 snímcích pomocí procesoru Intel Core i7-7700HQ 2, 80 GHz a grafické karty GTX 1050Ti 4GB, kde grafická karta vykazuje znatelně lepší výsledky.

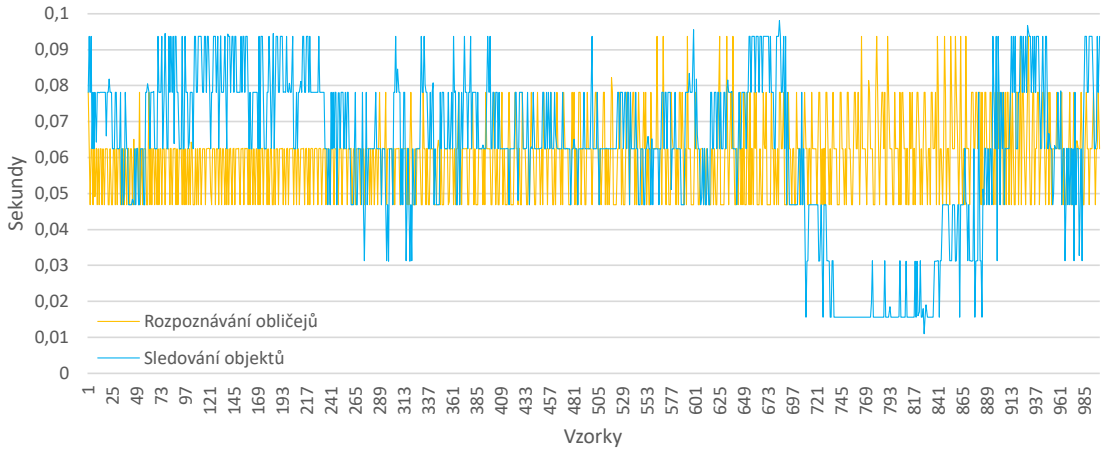
Pokud je nastavené sledování objektů, tak je tato operace stejně jako detekce provedena na každém snímku a značně se tak podílí i na celkové době zpracování. Ovšem DeepSORT, umožňuje zpracování v reálném čase a z naměřených hodnot pak vychází průměrná hodnota ~62 ms na jeden snímek (viz graf 5.2), což se stále dá považovat za přijatelný výsledek.

Pro ilustraci lze uvážit zpracování 5minutového videa, kde při výpočtu je zanedbána identifikace osob, protože je definovaná obsahem na záznamu a značně by tak zkreslovala výsledek. Pokud detekce pomocí GPU zabere 63 ms a sledování objektů 62 ms, potom jeden snímek potrvá zpracovat $63 + 62 = 125$ ms. Při snímové frekvenci 20 pak zpracovat 1 sekundu záznamu zabere $125 \cdot 20 \div 1000 = 2,5$ sekund a celý záznam poté $12,5$ minut. Výsledné zpomalení je $12,5 \div 5 = 2,5$, pokud by tento 5minutový záznam ovšem měl poloviční snímkovou frekvenci, pak by jeho zpracování zabralo pouhých $6,25$ minut.

5.2 Vyhodnocení přesnosti

Dosažení vyššího výkonu zajištěním co nejkratšího času na provedení detekce objektů přímo ovlivňuje její přesnost. V rámci řešení bylo vyzkoušeno více detektorů jako je např. SSD či Faster-RCNN s ResNet-50 aj. za účelem najít rovnováhu mezi přesností a dobou zpracování. Detekce jako první operace tak značně ovlivňuje další zpracování a její přesnost rozhoduje např. o tom, zda osoby pak budou vůbec rozpoznány či správně sledovány. Osobu totiž nelze nijak rozpoznat pokud daný bounding box neobsahuje její obličej. Časté výpadky při detekci objektu pak zapříčiňují špatnou asociaci s jeho předchozí polohou.

Při experimentech byla použita např. síť YOLOv3-tiny, která nabízí vysokou rychlost zpracování. Na CPU dosáhla průměrné rychlosti detekce 52 ms na snímek při nastaveném rozlišení 416×416 a tím umožnila $5,65$krát rychlejší detekci než běžné YOLOv3 při rozlišení 320×320. Ovšem s detekovanými objekty z této sítě již není možné dále v řešení pracovat (viz obrázek 5.3). V tomto případě jsou bounding boxy objektů velmi nepřesné, osoby jsou



Obrázek 5.2: Časy pro identifikaci jedné osoby podle obličeje a sledování detekovaných objektů na jeden snímek. Naměřeno na 1000 vzorcích.

detekovány bez obličejů, dochází k velkému množství výpadků a chybných detekcí (nízké confidence skóre). Z oficiální dokumentace tato síť poskytuje mAP *(mean average precision)* $33,1$. Z toho důvodu byl použit detektor YOLOv3 při rozlišení 320×320 jako nejrychlejší možnost, protože s těmito detekcemi lze s jistotou dále pracovat a jeho mAP dosahuje $51,5$, při rozlišení 608×608 bylo pak mAP naměřeno na $57,9$.

Obrázek 5.3: Porovnání detekce osob pomocí YOLOv3-tiny s rozlišením 416×416 (vlevo) a YOLOv3 při rozlišení 320×320 (vpravo), kde YOLOv3-tiny nabízí mnohonásobně větší rychlost zpracování, ovšem detekce jsou velmi nepřesné a některé zcela chybí.

Kapitola 6

Závěr

Zadáním této práce bylo nastudovat problematiku detekce a klasifikace objektů v obraze a následně se zaměřit na aktuální řešení pro rozpoznávání osob podle obličeje. Dále pak navrhnout a implementovat řešení pro detekci a analýzu narušitelů ve sledované oblasti v podobě klient-server aplikace. Celý systém otestovat na reálných záznamech a nakonec zhodnotit jeho praktické nasazení.

Nastudovaná teorie byla v tomto textu sepsána obecným úvodem do oblasti neuronových sítí, jenž má za úkol uvést čtenáře do kontextu práce. O ten se pak opírá následující sekce o konvolučních neuronových sítích, které se používají pro zpracování obrazu. Následně byly uvedeny některé páteřní architektury a detektory pro detekci a klasifikaci objektů. Další část textu pak byla věnována detekci obličejů, vybraným přístupům pro rozpoznávání osob a metodám pro sledování objektů. V rámci teoretické části byly také popsány technologie, které byly použity při implementaci výsledné aplikace.

Po návrhu řešení v rámci praktické části byl nejdříve implementován skript pro zpracování videozáznamu, po kterém byla následně vytvořena klientská a serverová část celého systému. Po integraci všech těchto částí vznikla použitelná aplikace pro detekci narušitelů s možností konfigurace, která je schopna spolehlivě zpracovávat záznamy od více uživatelů současně. Aplikace nabízí kromě identifikace osob řadu dalších užitečných funkcí jako je vymezení sledované oblasti, sledování a počítání objektů, vykreslování cest, uplatnění vlastního detektoru, modifikaci přesnosti a mnoho dalšího. Výsledky je pak umožněné sdílet mezi uživateli pomocí odkazu. Aplikace byla nakonec otestována na reálných záznamech a naměřené hodnoty byly vyhodnoceny.

Jako navazující rozšíření této práce by bylo možné přidat do konfigurace zpracování další funkce, které by uživatel mohl v rámci detekce narušitelů využít. Mezi takové funkce by mohl patřit zejména odhad výšky osob nebo jejich pozice v prostoru. Získané pozice by se pak odesílaly zpět ke klientovi obdobně jako současný souhrn narušitelů, kde by byly vyneseny např. v interaktivní 3D scéně. Pokud by však aplikace měla mít širší využití, tak při současné koronavirové situaci by mohla najít využití funkce pro měření vzdálenosti mezi osobami, kde by při blízkém kontaktu došlo k zaznamenání zvláštní události.

Literatura

[1] Aggarwal, C. *Neural Networks and Deep Learning*. 1. vyd. Gewerbestrasse 11, 6330 Cham, CHE: Springer, září 2018. 512 s. ISBN 9783319944623.

[2] Ananth, S. *Faster R-CNN for object detection* [online]. 2019. Aktualizováno 9. 8. 2019 [cit. 17. ledna 2022]. Dostupné z: https://towardsdatascience.com/faster-r-cnn-for-object-detection-a-technical-summary-474c5b857b46.

[3] Banks, A. a Porcello, E. *Learning React: Functional Web Development with React and Redux*. 1. vyd. 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA: O'Reilly Media, květen 2017. 350 s. ISBN 9781491954621.

[4] Bewley, A., Ge, Z., Ott, L., Ramos, F. a Upcroft, B. Simple Online and Realtime Tracking. *CoRR*. únor 2016, abs/1602.00763. Dostupné z: http://arxiv.org/abs/1602.00763.

[5] Bibile, U. *NodeJS Architecture & Concurrency Model* [online]. A Medium Corp., 2020. Aktualizováno 15. 1. 2020 [cit. 13. prosince 2021]. Dostupné z: https://chathuranga94.medium.com/nodejs-architecture-concurrency-model-f71da5f53d1d.

[6] Brown, E. *Web Development with Node and Express*. 1. vyd. 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA: O'Reilly Media, červen 2014. 329 s. ISBN 9781491949306.

[7] Caetano, J. *React Component Lifecycle* [online]. 2019. Aktualizováno 15. 11. 2019 [cit. 4. ledna 2022]. Dostupné z: https://dev.to/jaylcaetano/react-component-lifecycle-2npl.

[8] Casciaro, M. *Node.js Design Patterns*. 1. vyd. Livery Place, 35 Livery St, Birmingham B3 2PB, UK: Packt Publishing Ltd., prosinec 2014. 454 s. ISBN 9781783287314.

[9] Chand, S. *Node.js Tutorial – A Complete Guide For Beginners* [online]. 2021. Aktualizováno 30. 11. 2021 [cit. 14. prosince 2021]. Dostupné z: https://www.edureka.co/blog/nodejs-tutorial.

[10] Channe, A. *The Node.js Architecture!* [online]. A Medium Corp., 2020. Aktualizováno 2. 12. 2020 [cit. 12. prosince 2021]. Dostupné z: https://medium.datadriveninvestor.com/the-node-js-architecture-f86e2337bcd2.

[11] Dalal, N. a Triggs, B. *Histograms of Oriented Gradients for Human Detection*. San Diego, United States: IEEE Computer Society, červen 2005. Dostupné z: https://hal.inria.fr/inria-00548512.

[12] Deng, J., Guo, J., Zhou, Y., Yu, J., Kotsia, I. et al. RetinaFace: Single-stage Dense Face Localisation in the Wild. *CoRR*. květen 2019, abs/1905.00641. Dostupné z: http://arxiv.org/abs/1905.00641.

[13] Gandhi, R. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms* [online]. 2018. Aktualizováno 9. 7. 2018 [cit. 16. ledna 2022]. Dostupné z: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e.

[14] Girshick, R. B. Fast R-CNN. *CoRR*. duben 2015, abs/1504.08083. Dostupné z: http://arxiv.org/abs/1504.08083.

[15] Girshick, R. B., Donahue, J., Darrell, T. a Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*. listopad 2013, abs/1311.2524. Dostupné z: http://arxiv.org/abs/1311.2524.

[16] Gradilla, R. *Multi-task Cascaded Convolutional Networks (MTCNN) for Face Detection and Facial Landmark Alignment* [online]. 2020. Aktualizováno 28. 7. 2020 [cit. 11. května 2022]. Dostupné z: https://medium.com/@iselagradilla94/multi-task-cascaded-convolutional-networks-mtcnn-for-face-detection-and-facial-landmark-alignment-7c21e8007923.

[17] Great Learning Team . *Face Detection using Viola Jones Algorithm* [online]. 2020. Aktualizováno 2. 9. 2020 [cit. 11. února 2022]. Dostupné z: https://www.mygreatlearning.com/blog/viola-jones-algorithm.

[18] Great Learning Team . *Introduction to Resnet or Residual Network* [online]. 2020. Aktualizováno 28. 9. 2020 [cit. 10. ledna 2022]. Dostupné z: https://www.mygreatlearning.com/blog/resnet.

[19] Gudikandula, P. *A Beginner Intro to Neural Networks* [online]. A Medium Corp., 2019. Aktualizováno 24. 3. 2019 [cit. 7. ledna 2022]. Dostupné z: https://purnasaigudikandula.medium.com/a-beginner-intro-to-neural-networks-543267bda3c8.

[20] Gupta, R. *Breaking Down Facial Recognition: The Viola-Jones Algorithm* [online]. 2019. Aktualizováno 7. 8. 2019 [cit. 11. února 2022]. Dostupné z: https://towardsdatascience.com/the-intuition-behind-facial-detection-the-viola-jones-algorithm-29d9106b6999.

[21] Hahn, E. *Express in Action: Writing, building, and testing Node.js applications*. 1. vyd. 20 Baldwin Road, Shelter Island, NY 11964, USA: Manning Publications Co., 2016. 258 s. ISBN 9781617292422.

[22] He, K., Zhang, X., Ren, S. a Sun, J. Deep Residual Learning for Image Recognition. *CoRR*. duben 2015, abs/1512.03385. Dostupné z: http://arxiv.org/abs/1512.03385.

[23] Herron, D. *Node.js Web Development*. 3. vyd. Livery Place, 35 Livery St, Birmingham B3 2PB, UK: Packt Publishing Ltd., červen 2016. 376 s. ISBN 9781785881503.

[24] Imran, D. *Everything React – All about React!* [online]. A Medium Corp., 2019. Aktualizováno 5. 2. 2019 [cit. 28. prosince 2021]. Dostupné z: https://medium.com/@danyal_imran/everything-react-all-about-react-6d7a8de4bb05.

[25] Jain, P. *Components Lifecycle In React* [online]. 2019. Aktualizováno 29. 7. 2019 [cit. 4. ledna 2022]. Dostupné z: https://www.c-sharpcorner.com/article/components-lifecycle-in-react.

[26] Kanjee, R. *DeepSORT — Deep Learning applied to Object Tracking* [online]. 2020. Aktualizováno 31. 8. 2020 [cit. 12. května 2022]. Dostupné z: https://medium.com/augmented-startups/deepsort-deep-learning-applied-to-object-tracking-924f59f99104.

[27] Karimi, G. *Introduction to YOLO Algorithm for Object Detection* [online]. 2021. Aktualizováno 15. 4. 2021 [cit. 9. února 2022]. Dostupné z: https://www.section.io/engineering-education/introduction-to-yolo-algorithm-for-object-detection.

[28] Krumov, E. *Neural Networks: For beginners. By beginners.* [online]. Towards Data Science Inc., 2019. Aktualizováno 9. 10. 2019 [cit. 8. ledna 2022]. Dostupné z: https://towardsdatascience.com/neural-networks-for-beginners-by-beginners-6bfc002e13a2.

[29] Kumar, D. *Introduction to FaceNet: A Unified Embedding for Face Recognition and Clustering* [online]. 2019. Aktualizováno 26. 7. 2019 [cit. 14. února 2022]. Dostupné z: https://medium.com/analytics-vidhya/introduction-to-facenet-a-unified-embedding-for-face-recognition-and-clustering-dbdac8e6f02.

[30] Latiyan, M. *ReactJS | Virtual DOM* [online]. Shubham Yadav, Sri Ritwik, Neeraj Badam. 2021. Aktualizováno 26. 7. 2021 [cit. 1. ledna 2022]. Dostupné z: https://www.geeksforgeeks.org/reactjs-virtual-dom.

[31] Le, K. *A quick overview of ResNet models* [online]. A Medium Corp., 2021. Aktualizováno 31. 3. 2021 [cit. 10. ledna 2022]. Dostupné z: https://lekhuyen.medium.com/a-quick-overview-of-resnet-models-f8ed277ae81e.

[32] Lo, M. *RESTful API 101* [online]. 2021. Aktualizováno 5. 7. 2021 [cit. 13. dubna 2022]. Dostupné z: https://medium.com/geekculture/restful-api-101-b61671e5a3ea.

[33] Mardan, A. *Pro Express.js*. 1. vyd. 233 Spring Street, New York, NY 10013, USA: Apress Media, LLC, prosinec 2014. 352 s. ISBN 9781484200384.

[34] Nain, A. *EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks* [online]. A Medium Corp., 2019. Aktualizováno 7. 6. 2019 [cit. 11. ledna 2022]. Dostupné z: https://medium.com/@nainaakash012/efficientnet-rethinking-model-scaling-for-convolutional-neural-networks-92941c5bfb95.

[35] Nayeem, M. *Exploring Other Face Detection Approaches(Part 1) — RetinaFace* [online]. 2020. Aktualizováno 19. 7. 2020 [cit. 12. května 2022]. Dostupné z: https://medium.com/analytics-vidhya/exploring-other-face-detection-approaches-part-1-retinaface-9b00f453fd15.

[36] Pereira, R., Carvalho, G., Garrote, L. a Nunes, U. J. Sort and Deep-SORT Based Multi-Object Tracking for Mobile Robotics: Evaluation with New Data Association Metrics. *Applied Sciences*. leden 2022, sv. 12, č. 3. DOI: 10.3390/app12031319. ISSN 2076-3417. Dostupné z: https://www.mdpi.com/2076-3417/12/3/1319.

[37] Powers, S. *Learning Node*. 2. vyd. 1005 Gravenstein Highway North, Sebastopol, CA 95472, USA: O'Reilly Media, Inc., červen 2016. 288 s. ISBN 9781491943120.

[38] Redmon, J., Divvala, S. K., Girshick, R. B. a Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. *CoRR*. červen 2015, abs/1506.02640. Dostupné z: http://arxiv.org/abs/1506.02640.

[39] Ren, S., He, K., Girshick, R. B. a Sun, J. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *CoRR*. červen 2015, abs/1506.01497. Dostupné z: http://arxiv.org/abs/1506.01497.

[40] Sadílek, J. *Webová aplikace pro intuitivní sestavení filtrů textu*. Brno, CZ, 2020. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/23014.

[41] Saha, S. *A Comprehensive Guide to Convolutional Neural Networks – the ELI5 way* [online]. 2018. Aktualizováno 15. 12. 2018 [cit. 9. ledna 2022]. Dostupné z: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53.

[42] Saxena, P. *Deep Face Recognition* [online]. Aman Sood. 2021. Aktualizováno 11. 6. 2021 [cit. 14. února 2022]. Dostupné z: https://www.geeksforgeeks.org/deep-face-recognition.

[43] Saxena, P. *FaceNet – Using Facial Recognition System* [online]. 2021. Aktualizováno 14. 12. 2021 [cit. 14. února 2022]. Dostupné z: https://www.geeksforgeeks.org/facenet-using-facial-recognition-system.

[44] Schroff, F., Kalenichenko, D. a Philbin, J. FaceNet: A Unified Embedding for Face Recognition and Clustering. *CoRR*. březen 2015, abs/1503.03832. Dostupné z: http://arxiv.org/abs/1503.03832.

[45] Shorten, C. *Introduction to ResNets* [online]. 2019. Aktualizováno 24. 1. 2019 [cit. 10. ledna 2022]. Dostupné z: https://towardsdatascience.com/introduction-to-resnets-c0a830a288a4.

[46] Singh, A. *Feature Engineering for Images: A Valuable Introduction to the HOG Feature Descriptor* [online]. 2019. Aktualizováno 4. 9. 2019 [cit. 12. února 2022]. Dostupné z: https://www.analyticsvidhya.com/blog/2019/09/feature-engineering-images-introduction-hog-feature-descriptor.

[47] Socket.IO. *How it works* [online]. 2022, Aktualizováno 26. 3. 2022 [cit. 12. dubna 2022]. Dostupné z: https://socket.io/docs/v4/how-it-works.

[48] Soviany, P. a Ionescu, R. T. Optimizing the Trade-off between Single-Stage and Two-Stage Object Detectors using Image Difficulty Prediction. *CoRR*. březen 2018, abs/1803.08707. Dostupné z: http://arxiv.org/abs/1803.08707.

[49] Taigman, Y., Yang, M., Ranzato, M. a Wolf, L. DeepFace: Closing the Gap to Human-Level Performance in Face Verification. *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 2014, s. 1701–1708.

[50] Tan, M. a Le, Q. V. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. *CoRR*. květen 2019, abs/1905.11946. Dostupné z: http://arxiv.org/abs/1905.11946.

[51] Versloot, C. *Overview of activation functions for neural networks* [online]. 2020. Aktualizováno 24. 1. 2020 [cit. 7. ledna 2022]. Dostupné z: https://www.machinecurve.com/index.php/2020/01/24/overview-of-activation-functions-for-neural-networks.

[52] Wojke, N., Bewley, A. a Paulus, D. Simple Online and Realtime Tracking with a Deep Association Metric. *CoRR*. březen 2017, abs/1703.07402. Dostupné z: http://arxiv.org/abs/1703.07402.

[53] Yamashita, R. *Convolutional neural networks: an overview and application in radiology* [online]. Mizuho Nishio, Richard Kinh Gian Do, Kaori Togashi. 2018. Aktualizováno 22. 6. 2018 [cit. 8. ledna 2022]. Dostupné z: https://insightsimaging.springeropen.com/articles/10.1007/s13244-018-0639-9.

[54] Yamazaki, S. *Understanding Functional Components vs. Class Components in React* [online]. 2020. Aktualizováno 18. 8. 2020 [cit. 3. ledna 2022]. Dostupné z: https://www.twilio.com/blog/react-choose-functional-components.

[55] Zaidi, S. S. A., Ansari, M. S., Aslam, A., Kanwal, N., Asghar, M. N. et al. A Survey of Modern Deep Learning based Object Detection Models. *CoRR*. duben 2021, abs/2104.11892. Dostupné z: https://arxiv.org/abs/2104.11892.

[56] Zhang, K., Zhang, Z., Li, Z. a Qiao, Y. Joint Face Detection and Alignment using Multi-task Cascaded Convolutional Networks. *CoRR*. duben 2016, abs/1604.02878. Dostupné z: http://arxiv.org/abs/1604.02878.

Příloha A

Instalační manuál

V této příloze je uveden postup pro zprovoznění aplikace a serveru na lokálním počítači.

- Stáhněte a nainstalujte python 3.9.7 z https://www.python.org/downloads/release/python-397
- Stáhněte a nainstalujte Node.js v16.14.0 z https://nodejs.org/en/download/releases
- Pro Microsoft Windows byl použit video kodek OpenH264. V případě jiné systémové distribuce navštivte https://github.com/cisco/openh264/releases/tag/v1.8.0

Přejděte do adresáře `/server/src` a nainstalujte požadované knihovny.

1. `cd server/src`
2. `pip install -r requirements.txt`

Přejděte do adresáře `/server`, nainstalujte závislosti a spusťte lokální server.

1. `cd server`
2. `npm install`
3. `npm start`

Otevřete nový terminál, přejděte do adresáře `/app`, nainstalujte závislosti a spusťte aplikaci.

1. `cd app`
2. `npm install`
3. `npm start`

- Aplikace bude dostupná v prohlížeči na adrese http://localhost:3000
- Server používá sokety na portu `3001`
- Express naslouchá na portu `3002`

Příloha B

Akcelerace pomocí grafické karty

Tato příloha obsahuje postup pro akceleraci zpracování videozáznamu pomocí grafické karty, konkrétně neuronové sítě pro detekci a klasifikaci objektů. Pro zprovoznění detektoru na grafické kartě je nutné nainstalovat další software, včetně Nvidia CUDA a cuDNN, stáhnout zdrojový kód knihovny OpenCV a následně ho zkompilovat. Na konci tohoto procesu bude umožněno DNN modulu OpenCV přistupovat na GPU.

1. Nastavte python 3.9.7 jako výchozí verzi pythonu
2. Odinstalujte současnou knihovnu OpenCV následujícím příkazem:

   `pip uninstall opencv-python`
3. Nainstalujte Microsoft Visual Studio 2019 s doplňkem C++
4. Stáhněte a nainstalujte CUDA 11.0 Update 1 z
   https://developer.nvidia.com/cuda-11.0-update1-download-archive
5. Stáhněte a extrahujte cuDNN v8.0.5 pro CUDA 11.0 z
   https://developer.nvidia.com/rdp/cudnn-archive
6. Přesuňte obsah adresáře cuDNN do `"C:\Program Files \NVIDIA GPU Computing Toolkit\CUDA\v11.0"` a nahraďte všechny soubory
7. Stáhněte a nainstalujte CMake (aktuální verze 3.23.0) z
   https://cmake.org/download
8. Vytvořte adresář `opencv_build` (například na disku `C:\`)
9. Stáhněte zdrojový kód OpenCV 4.5.5 z https://opencv.org/releases
10. Stáhněte zdrojový kód OpenCV-contrib 4.5.5 z
    https://github.com/opencv/opencv_contrib/releases/tag/4.5.5
11. Extrahujte oba archivy `OpenCV-4.5.5` a `OpenCV-contrib-4.5.5` do adresáře `opencv_build`, výsledkem budou uvnitř 2 adresáře s odpovídajícími soubory
12. V adresáři `opencv_build` vytvořte adresáře `build` a `install`
13. Spusťte CMake a nastavte zdrojový kód na adresář `opencv_build/opencv-4.5.5` (tj. zdrojový kód OpenCV)

14. Nastavte adresář pro sestavení do `opencv_build/build` a zaškrtněte možnost *grouped*
15. Klikněte na *Configure*, specifikujte generátor na Visual Studio 2019, nastavte platformu podle vaší architektury (např. x64) a potvrďte *Finish*
16. Po skončení doplňte konfiguraci zaškrtnutím:
    - `WITH/WITH_CUDA`
    - `BUILD/BUILD_opencv_dnn`
    - `OPENCV/OPENCV_DNN_CUDA`
    - `ENABLE/ENABLE_FAST_MATH`
    - `BUILD/BUILD_opencv_world`
    - `BUILD/BUILD_opencv_python3`
17. Přejděte na volbu `OPENCV/OPENCV_EXTRA_MODULES_PATH`, klikněte na procházet a vyberte adresář `opencv_build/opencv_contrib-4.5.5/modules`
18. Stiskněte znovu *Configure*
19. Po skončení konfigurace zaškrtněte `CUDA/CUDA_FAST_MATH`
20. V konfiguraci `CUDA/CUDA_ARCH_BIN` nechte pouze architekturu vaší grafické karty, kterou lze dohledat na https://en.wikipedia.org/wiki/CUDA podle vašeho modelu (např. GTX 1050Ti používá verzi 6.1).
21. Nastavte `CMAKE/CMAKE_INSTALL_PREFIX` na adresář `opencv_build/install`
22. Z možnosti `CMAKE/CMAKE_CONFIGURATION_TYPES` odstraňte `Debug;` a ponechte pouze `Release`
23. Naposledy klikněte na *Configure*
24. Po dokončení stiskněte *Generate*
25. Otevřete terminál a spusťte kompilaci následujícím příkazem:

    ```
    "C:\Program Files\CMake\bin\cmake.exe" --build "C:\opencv_build\build"
        --target INSTALL --config Release
    ```

26. Kompilace zabere asi 2 hodiny
27. Uvnitř adresáře `.../AppData/Local/Programs/Python/Python39/Lib/site-packages/cv2` vytvořte novou složku `data`
28. Zkopírujte obsah adresáře `opencv_build/install/etc/haarcascades` do `.../AppData/Local/Programs/Python/Python39/Lib/site-packages/cv2/data`
29. Na konci tohoto procesu by měla být v pythonu dostupná knihovna OpenCV verze 4.5.5 a vygenerovaný soubor by měl být umístěn v:
    - `opencv_build/build/lib/python3/Release/cv2.cp39-win_amd64.py`
    - `.../AppData/Local/Programs/Python/Python39/Lib/site-packages/cv2/python-3.9/cv2.cp39-win_amd64.py`