UNIVERZITET SINGIDUNUM

Tehnički Fakultet

Upotreba metoda dubokog učenja u sintezi sistema za prepoznavanje lica

- diplomski rad -

Mentor:
prof. dr ***Milan Milosavljević***

Kandidat:
Davor Jordačević

Beograd, ***2020*****.**

Sadržaj

Uvod

Glavni problem koji je razmatran u ovom radu jesu komponente koje čine jedan sistem za prepoznavanje lica. Još od ranih dana sa pojavom kamera, pojavila se potreba za sistemima koji bi mogli da identifikuju osobe na njima. Tokom vremena su se razne metode smenjivale, od prepoznavanja pokreta, otisaka prstiju, do prepoznavanja lica. U ovom radu će biti reči upravo o tehnologijama koje se koriste za prepoznavanje lica, konkretno, fokusiraćemo se na tehnike dubokog učenja u sintezi sistema za prepoznavanje lica. Ovakvi sistemi su složeni i predstavljaju sintezu raznih tehnologija i metoda kako bi uspešno radili. Sam razvoj ovih sistema je zahtevan, i zahteva tim ljudi koji su sposobni za rešavanje kako matematičkih, tako i računarskih problema. Ovi problemi proizilaze iz potrebe za visokom tačnošću sistema, kao i od varijacija usled korišćenja širokog spektra opreme i alata. Svakoga dana se sve više softvera zasniva upravo na ovoj tehnologiji, a to podrazumeva njenu ispravnost, robusnost i tačnost. Jedna od tehnika koja je omogućila nagli razvoj ove oblasti jesu duboke neuronske mreže. Pored ovoga, ubrzan razvoj tehnologija i računarskih komponenti omogućili su znatno brži razvoj i trening ovakvih mreža, o čemu će biti reči u nastavku ovog rada.

Glavni cilj istraživanja u ovom radu jeste sinteza sistema za prepoznavanje lica koristeći već postojeće metode za detekciju, ekstrakciju vektora obeležja, i njihovo upoređivanje radi dobijanja željenih rezultata.

U ovom radu korišćene su analitičke metode kako bi se svaka celina razložila na delove i bolje objasnila. Sam sistem za prepoznavanje lica je jedna složena celina koja uključuje delove koji su se godinama razvijali i istraživali. Ovo znači da se kombinacijom ovih podsistema mogu dobiti novi sistemi sa specifičnom namenom ili performansama. Ovi delovi u produkucionom sistemu ne mogu da rade jedan bez drugog, dok je prilikom istraživanja moguće preskočiti neke od njih. Uzmimo za primer detekciju lica. Ukoliko je cilj sistema samo prepoznavanje, a za testiranje, razvoj i upotrebu se koriste slike koje sadrže samo detektovana lica, onda se sam korak detekcije može preskočiti. S obzirom na to da ovo često nije slučaj, fokusiraćemo se na sintezu kompletnog sistema. Treba uzeti u obzir da ovakav sistem mogu činiti i dodatne komponente, poput dela za predikciju da li je osoba stvarna ili lažna (eng. face anti-spoofing).

Analiza lica predstavlja jedan od bitnih procesa u našim životima. Ljudi analizom lica prikupljaju bitne podatke o drugim osobama. Ovo uključuje podatke o broju godina, polu, rasnoj pripadnosti. Takođe možemo prepoznati da li je osoba srećna ili tužna, ili pak neku drugu emociju. Pokreti usana su važni u oblasti prepoznavanja govora, kao i sve popularnijoj oblasti kao što je generisanje lažnih snimaka. Metode analize lica nam mogu reći gde je usmeren pogled neke osobe, odnosno šta privlači njenu pažnju, i ovo može biti posebno interesantno u marketingu i šopovima, kazinima. U medicini ove metode mogu biti od koristi za prepoznavanje nekih bolesti, poput autizma koji se odlikuje time što osobe imaju poteškoće da iskažu svoje emocije. Sve navedene metode koriste kako ljudi, tako i računari.

U ovom radu ćemo se fokusirati na metode koje se koriste u procesu detekcije lica, njegove ekstrakcije, obrade i zatim prepoznavanja.

Na početku ovog rada je prvo bitno da uvrdimo sta je zapravo prepoznavanje lica. Svaka osoba ima karakteristično lice, i to je ono što nas čini unikatnima. Većina metoda se bazira upravo na ovoj činjenici, i njihov cilj je ekstrakcija ovih obeležja (eng. features) za svaku osobu, a zatim njihova klasifikacija na osnovu određenih parametara.

Treba razlikovati algoritme za prepoznavanje po vise kriterijuma. Osnovna klasifikacija je na algoritme zasnovane na geometriji (eng. Geometry based) i algoritme zasnovane na šablonima (eng. template based). Geometrijski zasnovani algoritmi analiziraju određena područja i geometrijske veze na njima. Zbog ovoga su i poznati kao algoritmi zasnovani na obeležjima. Sa druge strane su algoritmi zasnovani na šablonima, i u ovu grupu spadaju: metoda nosećih vektora (SVM), analiza glavnih komponenti (PCA), linearna diskriminantna analiza (LDA), kernel metode i još mnogo drugih.

Većina prethodno spomenutih algoritma je jako brza, i mogu raditi na centralnoj procesorkoj jedinici (CPU). Budući da su se kako obim setova podataka, kao i zahtevi tržišta povećavali, bilo je potrebno dizajnirati sofisticiranije algoritme koji mogu da postignu veću tačnost, ali i brzine. U prethodnih 10 godina, oblast dubokog učenja je sa pojavom sve snažnijeg hardvera naglo dobija na značaju.

Pored hardvera, pojavio se i veliki broj javno dostupnih setova podataka. Ovo je značajno skratilo vreme potrebno za istraživanje i razvoj sistema.

Ove dve stvari su bile dovoljne kako bi sistemi zasnovani na dubokom učenju zamenili tradicionalne metode.

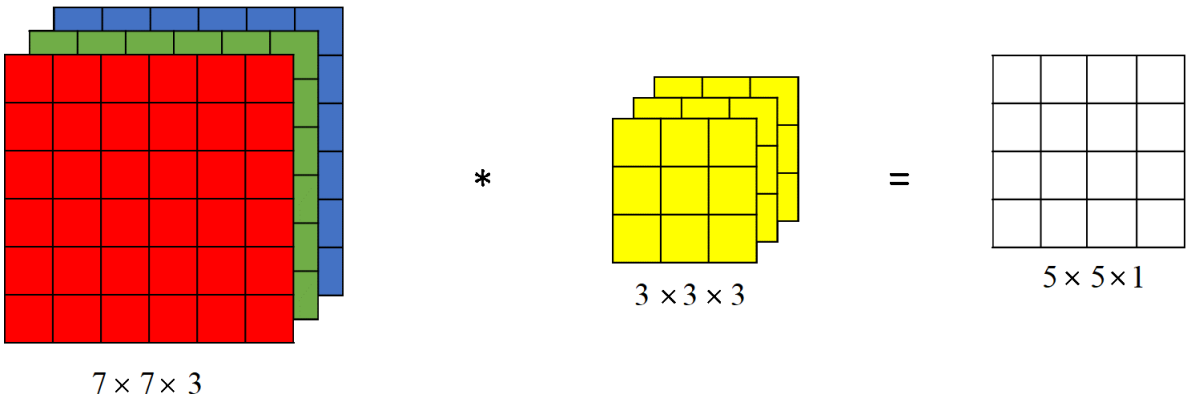
1. Neophodni koncepti

Arhitekture dubokih neuronskih mreža o kojima će biti reči su zasnovane na već postojećim principima i njihovom usavršavanju. Premda su ovi principi dosta izmenjeni, u velikom delu sistema ostaju nepromenjeni.

1.1 2D Konvolucija

Konvolucija predstavlja matematički operator koji od dve funkcije proizvodi treću. U oblasti dubokog učenja se pod konvolucijom podrazumeva primenu niza filtara na ulaznu sliku, kako bi se dobio željeni izlaz. Ukoliko želimo da sliku veličine D_u x D_u x M pretvorimo u izlaz dimenzija D_v x D_v x N, potrebno nam je N filtara, svaki dimenzija D_r x D_r x M.

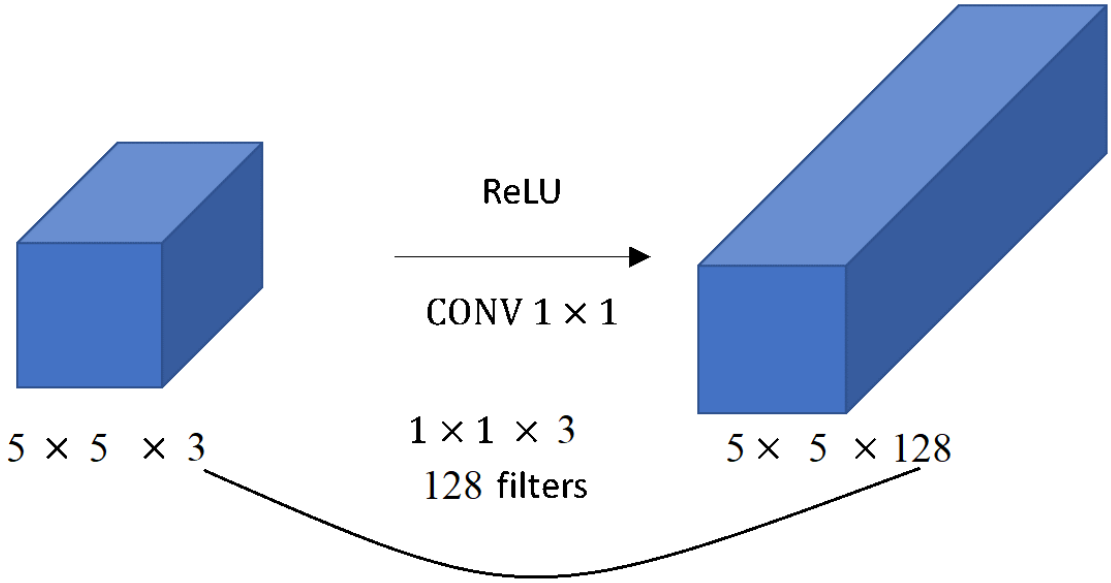
Veličina slike na izlazu iz mreže je (D_u - D_r + 2P) / S + 1, gde P predstavlja proširivanje(eng. padding) matrice, a S korak (eng. stride), dok je pomeraj (eng. bias) 1.



Slika 1. Primer 2D konvolucije

1.2 1x1 Konvolucija

Na prvi pogled se čini da 1x1 konvolucija nema smisla. Ovo je zapravo samo konvolucija sa filtrom velicine 1x1. U praksi, ako primenimo konvoluciju koristeći N filtra dimenzija 1x1 na sliku ulazne veličine D_u x D_u x M, rezultat je slika dimenzija D_u x D_u x N.



Slika 2. Primer 1x1 konvolucije

1.3 Višeklasna unakrsna entropija (eng. Cross-entropy) funkcija gubitka

Unakrsna entropija se često koristi kao funkcija gubitka za optimizaciju klasifikacionih modela. Često se naziva i Softmax funkcija gubitka (eng. softmax loss). Predstavlja Softmax aktivacionu funkciju u vezi sa unakrsnom entropijom.

1.4 MobileNetV2

Modul za detekciju u ovom sistemu kao osnovu (eng. backbone) koristi MobileNetV2 mrežu, isto kao i modul za ekstrakciju vektora obeležja.

Po rečima autora RetinaFace (Deng, arXiv:1905.00641v2) modela, MobileNetV2 arhitektura postiže drastično kraće vreme potrebno za predikciju (eng. inference) kada je u pitanju detekcija lica.

Backbones	VGA	HD	4K
ResNet-152 (GPU)	75.1	443.2	1742
MobileNet-0.25 (GPU)	1.4	6.1	25.6
MobileNet-0.25 (CPU-m)	5.5	50.3	-
MobileNet-0.25 (CPU-1)	17.2	130.4	-
MobileNet-0.25 (ARM)	61.2	434.3	-

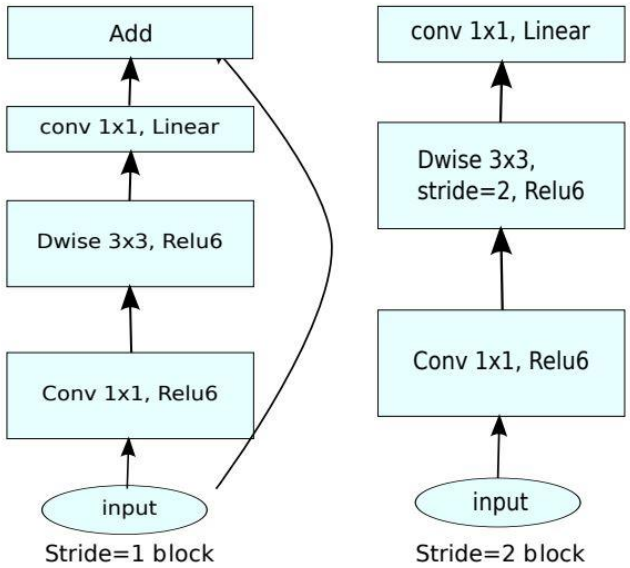
Slika 3. Komparacija vremena inference RetinaFace modela za detekciju lica

Arhitektura MobileNetV2 mreže je prikazana na slici.

Input	Operator	t	c	n	s
$224^2 \times 3$	conv2d	-	32	1	2
$112^2 \times 32$	bottleneck	1	16	1	1
$112^2 \times 16$	bottleneck	6	24	2	2
$56^2 \times 24$	bottleneck	6	32	3	2
$28^2 \times 32$	bottleneck	6	64	4	2
$14^2 \times 64$	bottleneck	6	96	3	1
$14^2 \times 96$	bottleneck	6	160	3	2
$7^2 \times 160$	bottleneck	6	320	1	1
$7^2 \times 320$	conv2d 1x1	-	1280	1	1
$7^2 \times 1280$	avgpool 7x7	-	-	1	-
$1 \times 1 \times 1280$	conv2d 1x1	-	k	-	

Slika 4. Arhitektura MobileNetV2 modela (Sandler, Mark , arXiv:1801.04381v4)

Kao što se može videti na slici, MobileNetV2 arhitektura je zasnovana na slojeavima uskog grla (eng. botleneck). Botleneck sloj predstavlja kombinaciju 1x1 konvolucije sa Relu6 aktivacijom, 3x3 depthwise konvolucije sa Relu6 aktivacijom i linearne 1x1 konvolucije.



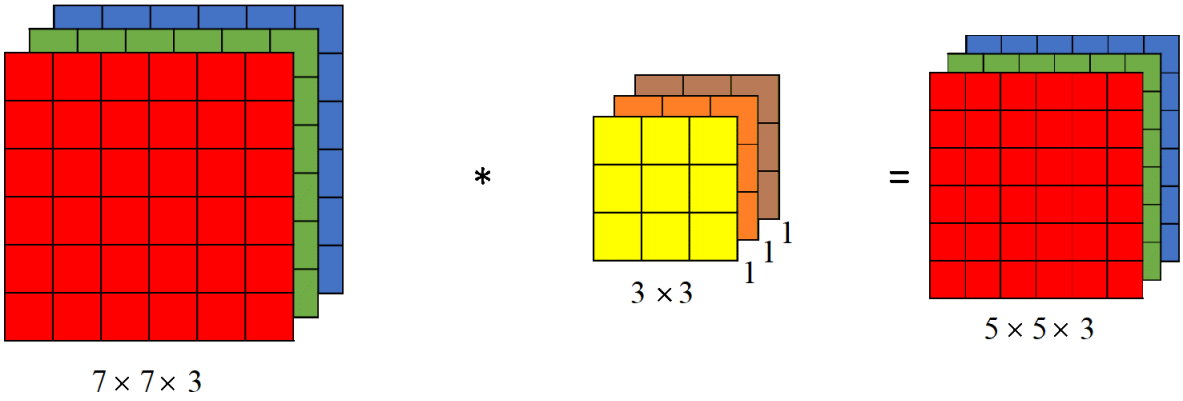
Slika 5. Arhitektura botleneck slojeva u MobileNetV2 mreži

1.4.1 Depthwise Separable Konvolucija

Potreba za poboljšanjem performansi dovela je i do nove vrste konvolucije. U prethodnom primeru sa 2D konvolucijom smo mogli videti kako nas može dovesti od ulaznih dimenzija 7x7x3 do 5x5x1. Ukoliko bi primenili 128 filtra dimenzija 3x3x3, ovo bi

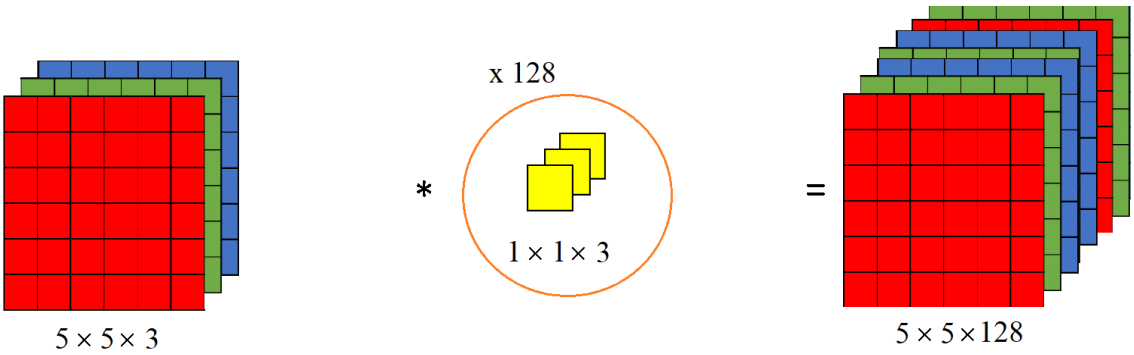
nas dovelo do izlaza dimenzija 5x5x128. U nastavku će biti prikazano kako se ovo može postići korišćenjem deptwise separable konvulucije.

Umesto da koristimo jedan filter dimenzija 3x3x3, možemo iskoristiti 3 kernela odvojeno. Svaki filter veličine 3x3x1. Svaki kernel vrši konvoluciju sa jednim ulaznim kanalom slike, dajući izlaz dimenzija 5x5x1. Spajanjem ovih izlaza, ponovo dobijamo dimenzije 5x5x3.



Slika 6. Primer separable konvolucije

Kako bi dobili željene dimenzije, sledeći korak je primena 1x1x3 filtra. Primenom ovog filtra na ulaz dimenzija 5x5x3, dobijamo izlaz dimenzija 5x5x1. Primenom 128 ovakvih filtra, dobijamo izlaz dimenzija 5x5x128.



Slika 7. Primer 1x1 konvolucije sa 128 filtra

2. Detekcija lica

Prvi korak u procesu prepoznavanja lica je njegova detekcija. Osnovna ideja procesa detekcije je pronalaženje svih lica na slici, odnosno njihovih koordinata. Ove koordinate služe za ekstrakciju lica sa slike, te se u nastavku radi samo sa slikama koje sadrže lice. Takođe se vrlo često koordinate čuvaju ukoliko je reč o radu sa video snimcima radi iscrtavanja. Pored ovoga, čuvanjem originalne slike i koordinata imamo mogucnost ponovne ekstrakcije lica sa raznim scale faktorima, što može biti korisno kao ulaz u neke druge neuronske mreže. Primer ovoga su anti-spoofing mreže koje imaju za cilj predikciju da li je osoba na slici ili snimku prava (eng. real) ili lažna (eng. fake).

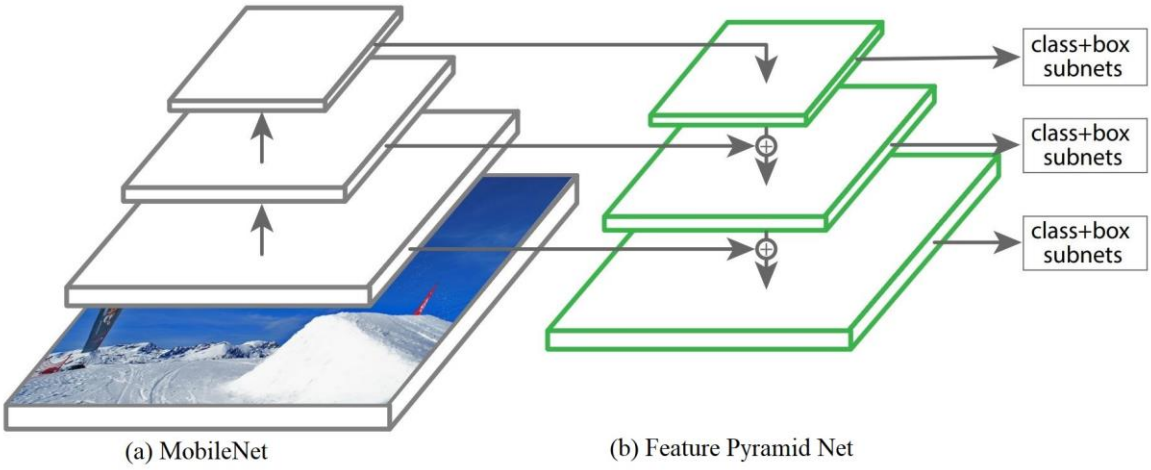
2.1 RetinaNet

Postoji zaista veliki broj razvijenih ideja i projekata na temu detekcije, kako objekata, tako i lica. Jedna implementacija se izdvaja od drugih, reč je o RetinaNet mreži (Lin, Tsung-Yi, arXiv:1708.02002v2).

Osnovna ideja sa RetinaNet detektorom je bila u uraditi sve predickije u jednoj fazi. U 2019. godini, state-of-the-art (SOTA) detektori su bili bazirani na mehanizmu sa dve faze (takozvani two-stage). Prva faza je podrazumevala generisanje skupa kandidata za lokaciju na kojoj je moguće naći traženi objekat, dok se druga faza sastoji iz klasifikacije svakog kandidata u jednu od klasa.

RetinaNet je jednofazni (single stage) detektor koji se sastoji iz jedne mreže (bacbone) i dve mreze sa specifičnom funkcijom. Kao backbone mreža se može koristiti bilo koja od poznathih arhitektura dubokih mreža (VGG, MobileNet, ResNet). Osnovni cilj ove mreže je računanje konvolucione feature mape. Nakon toga, prva podmreža radi klasifikaciju na osnovu izlaza backbone mreže, dok druga podmreža proračunava regresiju koordinata (eng. bounding box regression).

Pored toga što je ovaj detektor single stage, podržava koncept piramida. RetinaNet je zasnovan na konceptu piramida obeležja (Feature Pyramid Network - FPN) (Tang, Xu, arXiv:1803.07737v2). Koncept korišćenja piramida u obradi slika nije nov. Metode poput Gausovih (eng. Gaussian) ili Laplasovih (eng. Laplacian) piramida zasnivaju na radu sa početnom slikom, a zatim njenim downsamplingom ili upsamplingom. Ovo je ipak zahtevna operacija, kako procesorski, tako i memorijski. Korišćenjem FPN mreže kao backbone mreže omogućeno je raditi augmentaciju standardne konvolucione mreže sa vrha ka dnu sa lateralnim konekcijama (bočne veze). Ovo omogućava mreži da efikasno konstruiše piramidu sa različitim scale faktorima iz jedne slike. Svaki nivo piramide se može koristiti za detektovanje objekata u različitoj razmeri.



Slika 8 –RetinaNet arhitektura koja koristi FPN backbone mrežu preko MobileNet arhitekture

2.2 RetinaFace

Jedan od velikih problema koji se pojavljivao je bila detekcija lica različitih veličina u nekotrolisanim uslovima (prirodi, gužvama u gradu). Pošto već poznati koncept piramida može rešiti probleme detektovanja sa različitim scale faktorima, sledeća mreža koristi iste principe. RetinaFace (Deng, Jiankang, arXiv:1905.00641v2) je mreža nešto novijeg porekla i zasnovna je na istim principima na kojima se zasniva i RetinaNet, ali je namenjena isključivo detekciji lica.

RetinaFace predstvalja single stage detektor lica. Tvorci ovog modela su uneli nove ili unapredili već postojeće metode korišćene u ovu svrhu, kao što su multi-task obučavanje za istovremenu predikciju uverenja, bounding box-a, 5 ključnih tačaka na licu, i 3D poziciju (u originalnoj implementaciji).

Ono što je novo je 5 ključnih tačaka (eng. keypoints) koji će se kasnije koristiti za poravnanje lica.

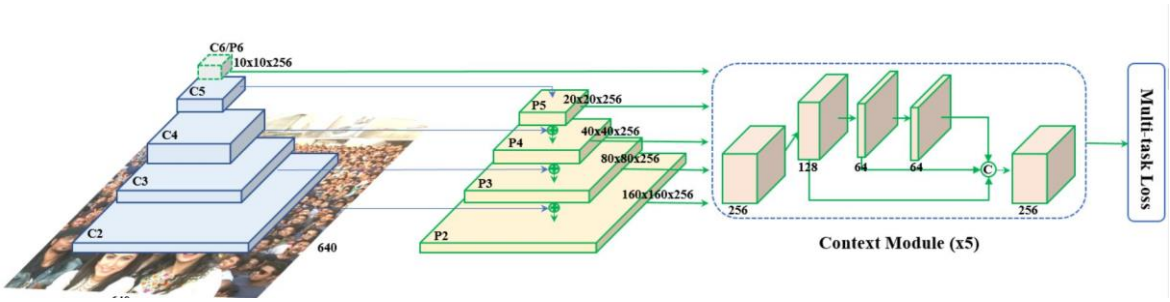
Kako bi se poboljšala detekcija lica, takozvanih Hard detekcija, korišćen je koncept modelovanja konteksta. Takozvana Hard lica su teška za detekciju zbog nedostatka vizualne konzistentosti, pozicije ili konteksta. Osnovna ideja je da mreža može da nauči ne samo obeležja koja su karakteristična za lice, već i kontekstualni deo kao što su vrat ili telo.

Kako bi se povećali efekti modelovanja nelinearnih (ne-rigidnih) transformacija (skaliranje, smicanje) korišćeni su kontekst moduli. Geometrijske varijacije predstavljaju jedan od velikih problema u oblasti detekcije i prepoznavanja. Metoda koja se pokazala korisnom u prevazilaženju ovih problema je korišćenje deformabilne (eng. deformable) konvolucije (Dai, Jifeng, arXiv:1703.06211v3).

Kao što je već pomenuto, RetinaFace mreža je zasnovana na pricipu multi-tast obučavanja. Samim tim se nameće korišćenje drugačije funkcije gubitka (eng. loss function). Loss funkcija korišćena u ovom slučaja je multi-task loss:

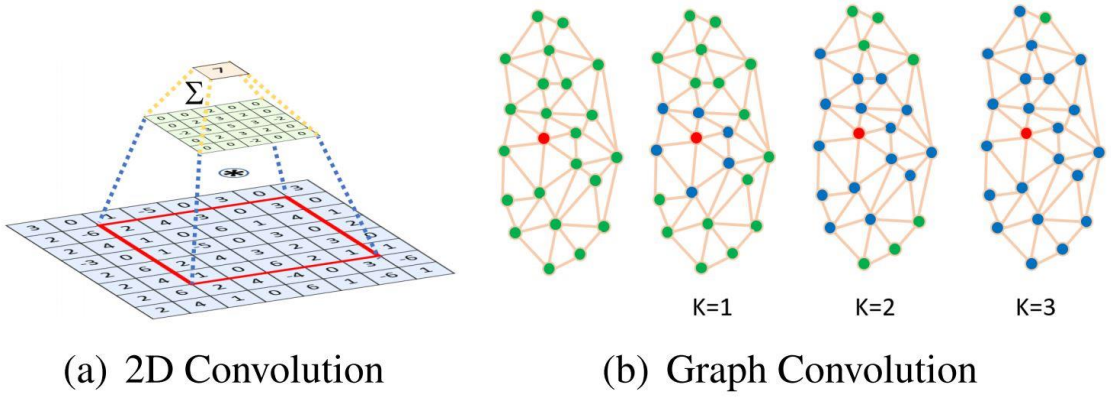
$$L = L_{cls}(p_i, p_i^*) + \lambda_1 p_i^* L_{box}(t_i, t_i^*) + \lambda_2 p_i^* L_{pts}(l_i, l_i^*) + \lambda_3 p_i^* L_{pixel}.$$

Ova funkcija se sastoji iz više delova, gde prvi deo L_{cls} predstavlja softmax gubitak binarne klasifikacije (ima lica/nema lica), drugi sabirak je gubitak regresije bounding box-ova, zatim sledi regresioni gubitak za predikciju 5 ključnih tačaka i dense regresioni gibitak.



Slika 9. Prikaz RetinaFace arhitekture. (Levo) Backbone mreža (MobileNetV2). (Sredina) Feature Pyramid Netword. (Desno) Konteks modul.

Kako bi ubrzao proces detekcije korišćen je takozvani mesh dekoder (mesh konvolucija i up-sampling). Ovo predstavlja vid graf konvolucionog metoda.

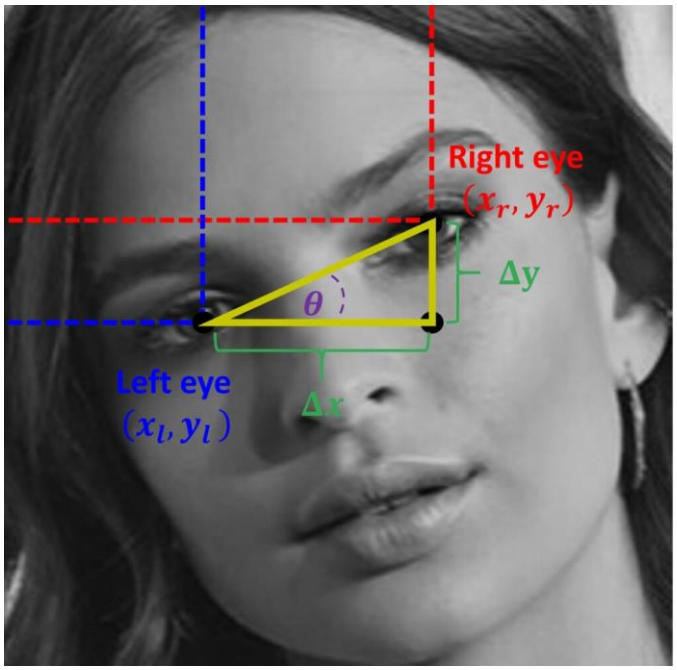


Slika 10. (Levo) 2D konvolucija. (Desno) Graf konvolucija.

3. Poravnanje lica

Nakon procesa detekcije i ekstrakcije lica i 5 ključnih tačaka, kako bi proces prepoznavanja bio što uspešniji, potrebno je uraditi poravnanje lica.

Jedan od jednostavnijih ali uspešnih metoda za ovo je pronalaženje arcus tangensa između dva oka (odnosno ugla između dva oka).



$$\Delta x = x_r - x_l$$

$$\Delta y = y_r - y_l$$

$$\theta = \arctan\frac{\Delta y}{\Delta x}$$

Slika 11. Primer nalaženja ugla izmedju očiju preko arkus tangensa

Ovde je bitno napomenuti da *arctan* funkcija u Numpy paketu vraća ugao u radijanima. Za dalju upotrebu potrebno je pretvoriti ugao u stepene. Za ovo je samo potrebno pomnožiti sa 10 i podeliti sa π. Nakon ovoga, potrebno je izračunati rotacionu matricu.

$$M = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Slika 12. Primer rotacione matrice

Sledeći korak je upotreba afinih transformacija kako bi postigli željeni efekat. Ovakva vrsta transformacija, odnosno preslikavanja preslikava tačke u tačke, prave u prave, ravni u ravni. Kod ovakvih transformacija, par paralelnih pravi ostaje paralelan i nakon preslikavanja, ali uglovi između pravih ili razdaljine između tačaka ne moraju nužno ostati iste.

4. Ekstrakcija vektora obeležja

Nakon što imamo sliku poravnanog lica, sledeći korak u procesu prepoznavanja je ekstrakcija vektora obeležja (eng. feature vector). Tokom godina su se smenjivale razne implementacije sa istom namenom, ali sa različitim metodama ekstrakcije vektora, kao i veličine vektora. Ranije SOTA implementacije poput FaceNet mreže su bile zasnovane na 128-dimenzionalnim vektorima. Novije metode imaju mogućnost ekstrakcije 512-dimenzionalnih obeležja, pa čak i 1024. U ovom slučaju, zadržaćemo se na 512.

Metode poput FaceNet (Schroff, Florian, Dmitry Kalenichenko, and James Philbin, arXiv:1503.03832v3) mreže su za cilj imale direktno učenje vektora obeležja zasnovano na triplet loss funkciji. Ideja je minimiziranje distance između vektora iste osobe (positive), dok se vektori različite osobe (negative) udaljavalju. Ovo je podrazumevalo da u svakom trenutku tokom treninga imamo tri vektora (anchor, positive, negative). Proces odabiranja tripleta je jako zahtevan i spor, što je bio prvi nedostatak ovakvog i sličnih metoda.

Jedna od implementacija koja je postigla SOTA rezultate je ArcFace (Deng, Jiankang, arXiv:1801.07698v3). Osnovna razlika i ideja je bila napraviti klasifikator koji može da razdvoji različite identitete u trening setu na osnovu određene loss funkcije, kao i korišćenje 512-dimenzionalnih vektora obeležja. Problem sa triplet loss obučavanjem je i eksponencijalni skok broja kombinacija kod velikih setova podataka, dok je problem sa tradicionalnim funkcijama poput Softmax funkcije što se linearna transformaciona matrica povećava linearno, što nije problem kod manjih setova podataka, ali je kod velikih setova ili produkcionih sistema neupotrebljivo.

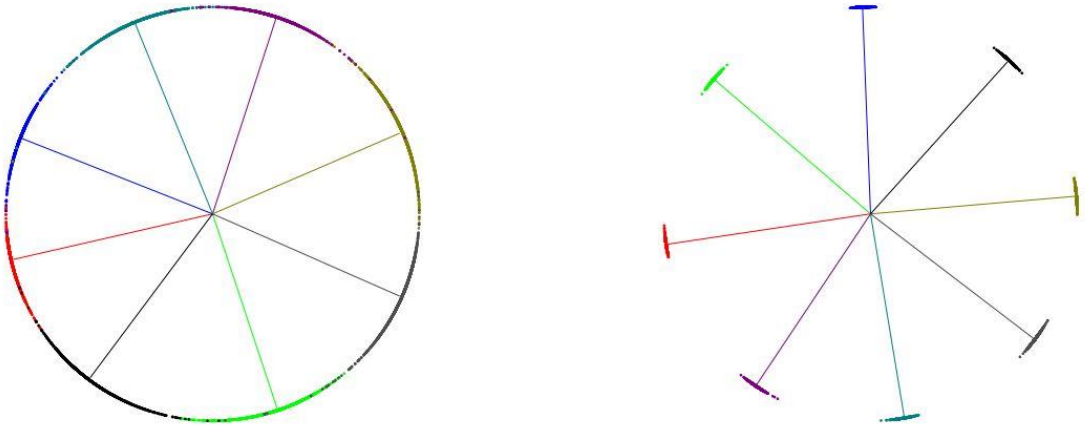
Kako bi se povećala margina između klasa, predstavljena je nova funkcija gubitka. Reč je o ArcFace funkciji. U nastavku sledi transformacija iz Softmax u ArfFace funkciju.

$$L_1 = -\frac{1}{N}\sum_{i=1}^{N} log\frac{e^{W_{yi}^T x_i + b_{y_i}}}{\sum_{i=1}^{n} e^{W_j^T x_i + b_j}}$$

$$L_2 = -\frac{1}{N}\sum_{i=1}^{N} log\frac{e^{scos(\theta_{j\,i})}}{e^{scos(\theta_{j\,i})} + \sum_{j=1, i\neq y_i}^{n} e^{scos(\theta_j)}}$$

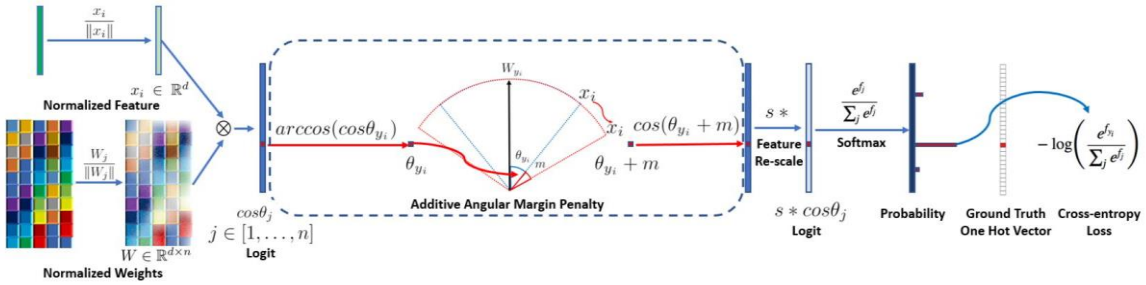
$$L_3 = -\frac{1}{N}\sum_{i=1}^{N} log\frac{e^{s(cos(\theta_{y_i}+m))}}{e^{s(cos(\theta_{y_i}+m))} + \sum_{j=1, j\neq y_i}^{n} e^{scos(\theta_j)}}$$

Na slici 13 prikazano je kako izgleda separacija 8 klasa korišćenjem Softmax i ArcFace funkcjie.



Slika 13. Primer separacije 8 klasa korišćenjem Sotfmax i ArcFace funckije gubitka

Kao backbone mreza je u originalnom radu korišćen ResNet, ali u našem slučaju je izabrana jednostavnije i brža arhitektura pod nazivom MobileNetV2.



Slika 14. Proces treninga mreže korišćenjem ArcFace funkcije

5. Pretraga vektora obeležja (prepoznavanje)

Ranije implementacije sličnih sistema su koristile razne metode pretrage. Od najjednostavnijih poput brute force pretrage, k najbližih suseda (KNN), metode nosećih vektora (SVM), kd-trees, LSH, pa i neuronskih mreža za pretragu vektora i predickiju identitieta.

Pomenute metode su se pokazale kao spore, ili nedovoljno precizne. Stoga, bilo je potrebe za novim i bržim metodama pretrage velikih skupova podataka. Metoda koja se izdvaja je pretraga aproksimiranih k najbližih suseda korišćenjem Hierarchical Navigable Small World grafova (HNSW) (Malkov, Yury A., and Dmitry A. Yashunin, arXiv:1603.09320v4).

KNN algoritam za svaki element iz skupa podataka prvo definiše razdaljinu od susednih elemanta. Ta razdaljina može da predstavlja rastojanje između dva vektora u n-dimenzionalnom prostoru i može se koristiti metrika po izboru. Parametar k biramo sami i on predstavlja broj suseda sa minimalnom razdaljinom od željene tačke, ili u našem slučaju vektora. Setovi podataka i njihove razmere u ovoj oblasti su veliki, kao i njihova dimenzionalnost. Stoga su metode kao što je linearno skeniranje spore. Treba napomenuti da sve novije metode ne garantuju vraćanje korektnog rezultata osim takozvane exhaustive pretrage. Ovo je poznato kao “curse of dimensionality” problem.

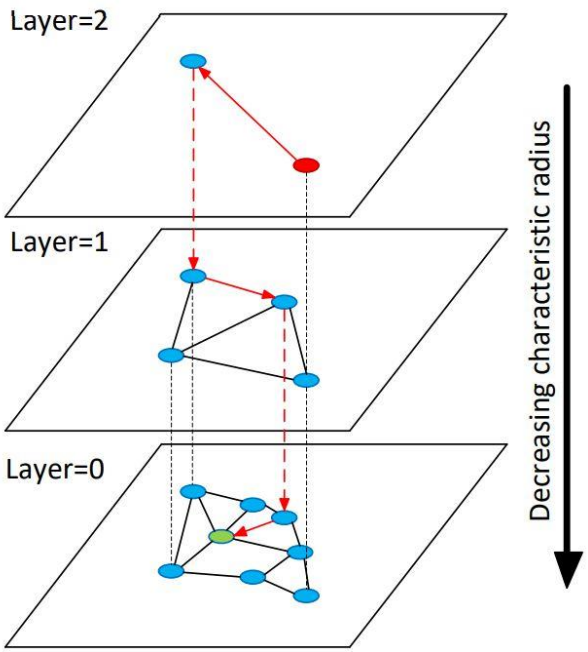
Kako bi se ovaj problem rešio, pojavljuje se metoda pod nazivom aproksimirani k najblizih suseda (Approximate Nearest Neighbors - ANN). Ova metoda dozvoljava mali broj grešaka, a kvalitet pretrage je definisan kao odnos tačno pronađenih suseda i parametra k.

Kako bi se ubrzala pretraga ANN algoritmom korišćene su tehnike transformacije vektora (smanjenje dimenzija, rotacija) pre indeksiranja, ali i enkodovanje vektora.

Vektori se mogu enkodovati koristeći stabla (Annoy), LSH, kvantizaciju, ali i grafove.

HNSW predstavlja implementaciju graf inkrementalnog ANN algoritma.

Teorija iza NSW i HNSW algoritma je previše obimna za ovaj rad, stoga će u nastavku ukratko biti predstavljeni algoritmi.



Slika 15. Vizualizacija procesa pretrahe HNSW grafa

Na slici 16 prikazani su algoritmi dodavanja čvora u graf, kao i pretraga jednog nivoa grafa.

Algorithm 1
INSERT(*hnsw*, *q*, *M*, M_{max}, *efConstruction*, m_L)
Input: multilayer graph *hnsw*, new element *q*, number of established connections *M*, maximum number of connections for each element per layer M_{max}, size of the dynamic candidate list *efConstruction*, normalization factor for level generation m_L
Output: update *hnsw* inserting element *q*
1 $W \leftarrow \emptyset$ // list for the currently found nearest elements
2 $ep \leftarrow$ get enter point for *hnsw*
3 $L \leftarrow$ level of ep // top layer for *hnsw*
4 $l \leftarrow \lfloor -\ln(unif(0..1)) \cdot m_L \rfloor$ // new element's level
5 **for** $l_c \leftarrow L \dots l+1$
6 $W \leftarrow$ SEARCH-LAYER(q, ep, ef=1, l_c)
7 $ep \leftarrow$ get the nearest element from W to q
8 **for** $l_c \leftarrow \min(L, l) \dots 0$
9 $W \leftarrow$ SEARCH-LAYER(q, ep, *efConstruction*, l_c)
10 $neighbors \leftarrow$ SELECT-NEIGHBORS(q, W, M, l_c) // alg. 3 or alg. 4
11 add bidirectionall connectionts from $neighbors$ to q at layer l_c
12 **for** each $e \in neighbors$ // shrink connections if needed
13 $eConn \leftarrow neighbourhood(e)$ at layer l_c
14 **if** $|eConn| > M_{max}$ // shrink connections of e
// if $l_c = 0$ then $M_{max} = M_{max0}$
15 $eNewConn \leftarrow$ SELECT-NEIGHBORS(e, $eConn$, M_{max}, l_c)
// alg. 3 or alg. 4
16 set $neighbourhood(e)$ at layer l_c to $eNewConn$
17 $ep \leftarrow W$
18 **if** $l > L$
19 set enter point for *hnsw* to q

Algorithm 2
SEARCH-LAYER(q, ep, ef, l_c)
Input: query element q, enter points ep, number of nearest to q elements to return ef, layer number l_c
Output: ef closest neighbors to q
1 $v \leftarrow ep$ // set of visited elements
2 $C \leftarrow ep$ // set of candidates
3 $W \leftarrow ep$ // dynamic list of found nearest neighbors
4 **while** $|C| > 0$
5 $c \leftarrow$ extract nearest element from C to q
6 $f \leftarrow$ get furthest element from W to q
7 **if** $distance(c, q) > distance(f, q)$
8 **break** // all elements in W are evaluated
9 **for** each $e \in neighbourhood(c)$ at layer l_c // update C and W
10 **if** $e \notin v$
11 $v \leftarrow v \cup e$
12 $f \leftarrow$ get furthest element from W to q
13 **if** $distance(e, q) < distance(f, q)$ or $|W| < ef$
14 $C \leftarrow C \cup e$
15 $W \leftarrow W \cup e$
16 **if** $|W| > ef$
17 remove furthest element from W to q
18 **return** W

Slika 16. (Levo) Algoritam za dodavanje novog čvora u graf. (Desno) Pertraga sloja u grafu.

Na slici 17 prikazani su jednostavni algoritami za pretragu komšija, kao i algoritam pretrage koristeći heuristike.

Algorithm 3
SELECT-NEIGHBORS-SIMPLE(q, C, M)
Input: base element q, candidate elements C, number of neighbors to return M
Output: M nearest elements to q
return M nearest elements from C to q

Algorithm 5
K-NN-SEARCH(*hnsw*, q, K, ef)
Input: multilayer graph *hnsw*, query element q, number of nearest neighbors to return K, size of the dynamic candidate list ef
Output: K nearest elements to q
1 $W \leftarrow \emptyset$ // set for the current nearest elements
2 $ep \leftarrow$ get enter point for *hnsw*
3 $L \leftarrow$ level of ep // top layer for *hnsw*
4 **for** $l_c \leftarrow L \dots 1$
5 $W \leftarrow$ SEARCH-LAYER(q, ep, ef=1, l_c)
6 $ep \leftarrow$ get nearest element from W to q
7 $W \leftarrow$ SEARCH-LAYER(q, ep, ef, l_c =0)
8 **return** K nearest elements from W to q

Algorithm 4
SELECT-NEIGHBORS-HEURISTIC(q, C, M, l_c, *extendCandidates*, *keepPrunedConnections*)
Input: base element q, candidate elements C, number of neighbors to return M, layer number l_c, flag indicating whether or not to extend candidate list *extendCandidates*, flag indicating whether or not to add discarded elements *keepPrunedConnections*
Output: M elements selected by the heuristic
1 $R \leftarrow \emptyset$
2 $W \leftarrow C$ // working queue for the candidates
3 **if** *extendCandidates* // extend candidates by their neighbors
4 **for** each $e \in C$
5 **for** each $e_{adj} \in$ *neighbourhood*(e) *at layer* l_c
6 **if** $e_{adj} \notin W$
7 $W \leftarrow W \cup e_{adj}$
8 $W_d \leftarrow \emptyset$ // queue for the discarded candidates
9 **while** $|W| > 0$ and $|R| < M$
10 $e \leftarrow$ extract nearest element from W to q
11 **if** e is closer to q compared to any element from R
12 $R \leftarrow R \cup e$
13 **else**
14 $W_d \leftarrow W_d \cup e$
15 **if** *keepPrunedConnections* // add some of the discarded // connections from W_d
16 **while** $|W_d| > 0$ *and* $|R| < M$
17 $R \leftarrow R \cup$ extract nearest element from W_d to q
18 **return** R

Slika 17. (Levo) Pretraga suseda. (Desno) Pretraga suseda pomoću heuristike.

Jedna od osnovnih stvari prilikom pretrage je izbor odgovarajuće distance. S obzirom na to da je ArcFace model treniran korišćenjem ArcFace funkcije, automatski se nameće korišćenje cos-inusne distance ili angular distance.

6. Implementacija u programskom jeziku Python

Za implementaciju sistema za prepoznavanje lica je korišćen programski jezik Python 3.7. Za upravljanje paketima je korišćena Conda.

U nastavku se nalazi lista paketa koji su korišćeni za implementaciju:

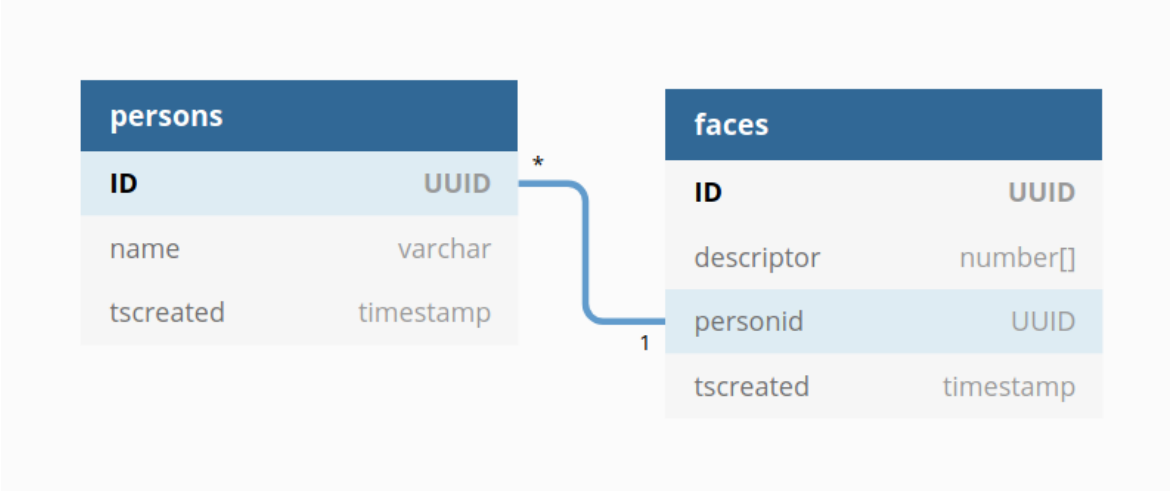
1. N2
2. Flask
3. Pillow
4. Pyfiglet
5. MTCNN
6. Psycopg2
7. Flask-cors
8. OpenCV 4.4.0
9. build-essential
10. Tensorflow-gpu
11. googledrivedownloader

Za kompletnu instalaciju sistema potrebno je instalirati tri odvojena dela, bazu, Python servis i frontend servis. Instalaciju je moguće uraditi ručno, ili uz pomoć dostupnih skripti. Sistem je testiran samo na Linux operativnom sistemu, tačnije na Ubuntu 20.04 LTS.

Ukoliko su performance bitan faktor, potrebno je instalirati TensorFlow sa podrškom za grafičku karticu, ukoliko ista postoji. Grafička kartica u ovom slučaju mora imati CUDA podršku, kao i instalirane adekvatne drajvere, CUDA, i CUDNN biblioteke. Moguće je konfigurisati i OpenCV biblioteku za rad sa grafičkom karticom, ali je zbog kompleksnosti i raznovrsnosti arhitektura preskočeno u instalacionoj skripti.

U nastavku neće biti prikazan ceo kod zbog svoje obimnosti, ali će biti dostupan. Fokisiraćemo se na delove koji su ključni i povezuju sve u jednu celinu. Treba napomenuti da se u sistemu nalaze i moduli koji nisu uključeni, na primer SSD detektor lica, MTCNN detektor lica, anti-spoofing modul. Promenom paramtera u JSON konfiguracionom fajlu moguće je menjati detektor, ili uključiti i isključiti eksperimentalni deo za anti-spoofing. Ovime je dobijeno na modularnosti sistema, novi moduli se mogu lako dodavati, kao i podešavati parametri već postojećih. Ovo omogućava jednostavnije testiranje.

Za početak je bitno kreirati bazu podataka. U ovom slučaju je korišćena PostgreSQL baza, ali je vrlo jednostavno zameniti je. U bazi imamo samo dve tabele, tabelu *persons* gde čuvamo informacije o imenima osoba, datum kreiranja i *ID* (UUIDv4) i drugu tabelu *faces* gde čuvamo vektore obeležja osoba (deskriptore), *ID* (UUIDv4) zapisa, *personid* (FK na tabelu persons) i datum kreiranja zapisa. Na slici 18 prikazan je model baze podataka.



Slika 18. Model baze podataka

U nastavku slede delovi koda neophodni za proces prepoznavanja osobe.

Za čitanje podataka iz baze kreirana je funkcija *read_descriptors*. Ona čita sve podatke iz tabele faces koji se kasnije koriste za kreiranja indeksa za pretragu.

```
def read_descriptors(db):
    """
    read_descriptors
    The function for reading all face descriptors and ids from database
    :param db: cursor
    :return: {} or [], [], []
    """
    try:
        db
    except NameError:
        print('Problem with the database connection')
        return -1

    query = 'SELECT "ID", descriptor, personid FROM public.faces'
    db.execute(query)
    records = db.fetchall()

    ids = []
    descriptors = []
    persons_ids = []

    if records:
        for r in records:
            ids.append(r[0])
            descriptors.append(r[1][0])
            persons_ids.append(r[2])

        return ids, descriptors, persons_ids
    else:
        return {'status': 'ERROR'}
```

Nakon čitanja vektora iz baze, potrebno je dodati ih u HNSW graf. Ovo radimo metodom *make_base*. Ova funkcija se nalazi u klasi *RecognitionEngine*. Ideja je dodati samo vektore i kreirati indeks. Ovo možemo sačuvati radi kasnijeg učitavanja ukoliko dođe do neke neželjene situacije. N2 paket podržava multithreading, tako da je ova opcija korišćena radi bržeg kreiranja indeksa.

```
def make_base(self, descriptors: []) -> dict:
    """
    make_base
    The function used for adding data and building the index
    :param descriptors: []
    :return: img: numpy.array()
    """
    self.recognizer = HnswIndex(512, 'angular')

    # add vectors to the ann
    for d in descriptors:
       self.recognizer.add_data(np.array(d))

    # build ann
    self.recognizer.build(m=5, max_m0=10, n_threads=4)

    self.recognizer.save('index.hnsw')
    return {'status': 'SUCCESS'}
```

Potrebno je inicijalizovati detektor lica koji se nalazi u klasi *ImgProcessor*. U ovom delu, učitavamo RetinaFace model sa diska i inicijalizujemo težine mreže vrednostima iz fajla.

```
if self.detector_type == "RetinaFace":
    # set config and checkpoints path
    self.face_det_cfg_path = load_yaml(cfg['face_det_cfg_path'])
    self.face_det_checkpoints_path = cfg['face_det_checkpoints_path']
    # load our serialized model from disk
    # Here we need to read our pre-trained neural net
    self.detector = RetinaFaceModel(self.face_det_cfg_path,
                                    training=False,
                                    iou_th=cfg["face_det_iou_th"],
                                    score_th=cfg["face_det_score_th"])

 self.face_det_down_scale_factor = cfg["face_det_down_scale_factor"]
 # load checkpoint
 checkpoint_dir = self.face_det_checkpoints_path +
                  self.face_det_cfg_path['sub_name']
 checkpoint = tf.train.Checkpoint(model=self.detector)
 if tf.train.latest_checkpoint(checkpoint_dir):
     checkpoint.restore(tf.train.latest_checkpoint(checkpoint_dir))
 else:
     exit()
```

Pored detektora, potrebno je inicijalizovati i modul za ekstrakciju vektora obeležja. Ovde se vrlo lako može zameniti backbone tip mreže sa MobileNet na ResNet.

```
# set config and checkpoints path
self.face_reco_cfg_path = load_yaml(cfg['face_reco_cfg_path'])
self.face_reco_checkpoints_path = cfg['face_reco_checkpoints_path']

# initialize the ArcFace model
self.model = ArcFaceModel(size=self.face_reco_cfg_path['input_size'],

backbone_type=self.face_reco_cfg_path['backbone_type'], training=False)

# load model weights
ckpt_path = tf.train.latest_checkpoint(self.face_reco_checkpoints_path
+

self.face_reco_cfg_path['sub_name'])
if ckpt_path is not None:
    self.model.load_weights(ckpt_path)
else:
    exit()
```

Za proces detekcije je zbog preglednosti prikazan je samo deo koji koristi RetinaFace detektor. U ovom delu, prvi korak je uraditi promenu veličine slike ukoliko je to potrebno, a nakon toga dodati padding kako bi se izbegao problem sa neodgovarajucim dimenzijama slike i ulaza u mrežu. Nakon inference, potrebno je ukloniti padding efekat kako bi imali početnu sliku.

U petlji prolazimo kroz svaku detekciju i uzimamo koordinate bounding box-a. Pored ovog, koristeći informacije o poziciji očiju vršimo poravnanje po već pomenutom algoritmu. Svaku sliku na kojoj postoji lice dodajemo u niz koji metoda prosledjuje narednoj metodi. Ukoliko je potrebno, sliku možemo sačuvati na disk promenom konfiguracionog fajla.

```
def detect(self, img: np.array([])) -> np.array([]):
    """
    detect
    The actual function for performing detection based on configuration
    parameter.
    Crops all faces from the image and returns the numpy array
    containing all of them.
    :param img: numpy.array()
    :return: numpy.array()
    """

    faces_array = []

    if self.detector_type == "RetinaFace":
        img_height, img_width, _ = img.shape

        if self.face_det_down_scale_factor < 1.0:
            img = cv2.resize(img, (0, 0),
                             self.face_det_down_scale_factor,
```

```
                                self.face_det_down_scale_factor,
                                cv2.INTER_LINEAR)
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)

            # pad input image to avoid unmatched shape problem
            img, pad_params = pad_input_image(img,
                              max_steps=max(self.face_det_cfg_path["steps"]))

            faces = self.detector(img[np.newaxis, ...]).numpy()
            # recover padding effect
            faces = recover_pad_output(faces, pad_params)

            for face in range(len(faces)):
                # get coordinates
                x1, y1, x2, y2 = int(faces[face][0] * img_width),
                                 int(faces[face][1] * img_height),
                                 int(faces[face][2] * img_width),
                                 int(faces[face][3] * img_height)
            x1 = 0 if x1 < 0 else x1
            y1 = 0 if y1 < 0 else y1

            f = img[y1:y2, x1:x2]

            if self.experimental:
                if not self.is_alive(f)['isAlive']:
                    continue

            # landmark
            if faces[face][14] > 0:
                right_eye_x, right_eye_y = int(faces[face][4] * img_width),
                                           int(faces[face][5] * img_height)
                left_eye_x, left_eye_y = int(faces[face][6] * img_width),
                                         int(faces[face][7] * img_height)

                delta_x = right_eye_x - left_eye_x
                delta_y = right_eye_y - left_eye_y
                angle = np.arctan(delta_y / delta_x)
                angle = (angle * 180) / np.pi

                h, w = f.shape[:2]
                # Calculating a center point of the image
                # Integer division "//"" ensures that we receive whole
                # numbers
                center = (w // 2, h // 2)
                # Defining a matrix M and calling
                m = cv2.getRotationMatrix2D(center, (angle), 1.0)
                # Applying the rotation to our image using the
                aligned_face = cv2.warpAffine(f, m, (w, h))
                faces_array.append(aligned_face)

        if self.write == "true":
            for face in faces_array:
                cv2.imwrite(str(uuid.uuid4())+'.jpg', face)
```

Nakon što imamo poravnanu sliku lica, sledeći korak je uraditi inferencu koristeći ArcFace mrežu i dobiti vektore obeležja. Za ovo je kreirana metoda klase *ImgProcessor* pod nazivom encode. Pre inference je potrebno normalizovati sliku i pretvoriti je u *float32* format. Nakon inference, L2 normalizacijom postižemo da koeficijenti vektora budu manji, i manje kompleksnosti.

```
def encode(self, img: np.array([])) -> np.array([]):
    """
    encode
    Function for extracting face embeddings using ArcFace model
    :param img: numpy.array()
    :return: img: numpy.array()
    """
    img = cv2.resize(np.array(img),
                     (self.face_reco_cfg_path['input_size'],
                      self.face_reco_cfg_path['input_size']),
                      0, 0, cv2.INTER_LINEAR)
    img = img.astype(np.float32) / 255.
    if len(img.shape) == 3:
        img = np.expand_dims(img, 0)
    # extract face embeddings and normalize
    embeds = l2_norm(self.model(img))
    return embeds
```

Sledeći korak u procesu prepoznavanja je pretraga vektora i traženje onog sa najmanjom distancom. Za ovo koristimo već pomenutu klasu *RecognitionEngine* i metodu *identification*. Metoda kao ulazne parametre dobija vektor obeležja i spisak *ID*-jeva osoba. Nema potrebe za celim nizom vektora jer su oni već učitani korišćenjem *make_base* metode.

```
def identification(self, descriptor: [], person_ids: []) -> dict:
    """
    identification
    Efficient and robust approximate nearest neighbor search using
    Hierarchical Navigable Small World graph
    :param descriptor: []
    :param person_ids: []
    :return: dict
    """
    idx = self.recognizer.search_by_vector(
                                np.array(descriptor).flatten(),
                                2, 1, include_distances=True
    )

    if idx[0][1] < self.threshold:
        person_id = person_ids[idx[0][0]]
    else:
        person_id = "Not recognized"
    return {
            'status': 'SUCCESS',
            'personid': person_id
    }
```

Ceo proces prepoznavanja se može povezati u jednu funkciju koja služi kao Rest endpoint. U ovoj funkciji je pre početka detekcije prvo bitno proveriti rezoluciju slike. Ovo radimo kako ne bi došlo do problema sa preopterećenjem resursa, odnosno potpunog iskorišćenja memorije grafičke kartice. Ovo može biti samo jedno od rešenja u sistemima sa ograničenom memorijom, ali je moguće uraditi i promenu veličine slike, pri čemu se mora voditi računa da se odnos širine i visine slike ne promeni. Za ovo je i implementirana funkcija, ali u ovom slučaju nije korišćena.

```
# Create a URL route in our application for
# "/@app.route('/identification')"
# The purpose of this endpoint is to predict the person identity
@app.route('/identification', methods=["POST"])
@cross_origin()
def predict_rest() -> dict:
    """
    predict_rest
    The actual function for performing the recognition.
    Extract all faces from the image, check if faces are real,
    extract face descriptors and search the database using HNSW
    and angular distance metric.
    :return: dict
    """
    pil_image = Image.open(request.files['image'])
    img = np.float32(pil_image)

    # check image size (important for gpus with less vram)
    if img.shape[0]*img.shape[1] > 1920*1080:
        return {
            'status': 'ERROR',
            'response': 'Inappropriate image size (use smaller images).'
        }

    # detect faces
    faces = imgProcessor.detect(img)
    response = []

    global ids, descriptors, persons_ids

    start = time.time()
    if persons_ids:
        for face in faces:
            # encode face
            descriptor = imgProcessor.encode(face)
            # find personid in the database
            person_id = recEngine.identification(descriptor,
                                                 persons_ids)
             if person_id['personid'] is not None:
                 # find name in the database
                 person = dbase.find_person_by_id(db,
                                                  person_id['personid'])
                response.append(person)
            else:
                response.append(person_id)
```

```
    else:
       return {
            'status': 'ERROR',
            'response': 'Empty database'
        }
    logging.info('Identification time: ' + str(time.time()-start))
    return {
        'status': 'SUCCESS',
        'response': response
    }
```

Ukoliko je osoba prepoznata, odnosno njen *ID* je vraćen kao rezultat funkcije, u bazi možemo pronaći njeno ime jednostavnim upitom. Ovo je prikazano u sledećem delu koda.

```
def find_person_by_id(db, person_id) -> dict:
    """
    find_person_by_id
    The function for search the database for name
    :param db: cursor
    :param person_id: str
    :return: dict
    """
    query = 'SELECT p.name FROM public.persons p WHERE "ID" = %s;'
    db.execute(query, (person_id,))
    records = db.fetchall()
    if records[0][0] is not None:
        return {
            'status': 'SUCCESS',
            'name': str(records[0][0])
        }
    else:
        return {'status': 'ERROR'}
```

Proces prepoznavanja je beskoristan ukoliko je baza prazna. U ovom slučaju se ni indeks ne može kreirati, samim tim ni pretraga. Kako bi se ovo izbeglo, potrebna je funkcija koja vektore upisuje u bazu.

Već pomenute funkcije detect i encode se mogu iskoristiti. Nakon detektovanja lica, poravnanja, i ekstrakcije vektora obeležja, možemo preskočiti proces prepoznavanja i vektore upisati u bazu. Treba napomenuti da je nakon upisivanja vektora u bazu, potrebno ponovo kreirati indeks, odnosno pozvati *make_base* metodu. Ovo se može izbeći inkrementalnim dodavanjem ukoliko biblioteka koja se koristi to podržava.

```
def receive_descriptors(db, db_conn, name, embeds) -> dict:
    """
    receive_descriptors
    The function for writing records in database
    :param db: cursor
    :param db_conn: connection
    :param name: str
```

```
    :param embeds: np.array([])
    :return: dict
    """
    query = 'INSERT INTO public.persons (name) VALUES (\'' + str(name) +
            '\') RETURNING "ID";'
    db.execute(query)
    records = db.fetchall()
    if records[0][0] != '':
        person_id = records[0][0]

    for emb in embeds:
        emb = np.array(emb).tolist()
        query = 'INSERT INTO public.faces (descriptor, personid)
                  VALUES (%s, %s);'
        db.execute(query, (emb, (person_id,)))
    db_conn.commit()
    return {'status': 'SUCCESS'}
```

Kao i u slučaju prepoznavanja, i ovde se ceo proces enkodovanja i upisivanja u bazu može spojiti u jednu funkciju i izložiti eksterni endpoint.

```
# Create a URL route in our application for
# "/@app.route('/encodeAndInsert)"
# The purpose of this endpoint is to encode the face
@app.route('/encodeAndInsert', methods=["POST"])
def encode_and_insert() -> dict:
    """
    encodeAndInsert
    The actual function for adding a new person into database.
    After person is added successfully, make base is performed.
    :return: dict
    """
    name = request.form.get('name')
    uploaded_files = request.files.getlist("images")

    embeds = []
    if not uploaded_files:
        return {"status": "ERROR"}
    for image in uploaded_files:
        pil_image = Image.open(image)
        img = np.float32(pil_image)

        # check image size (important for gpus with less vram)
        if img.shape[0] * img.shape[1] > 1920 * 1080:
            return {
            'status': 'ERROR',
            'response': 'Inappropriate image size (use smaller images).'
            }
        # detect face
        faces = imgProcessor.detect(img)
        if len(faces) != 1:
            return {
            'status': 'ERROR',
```

```
            'response': 'Images must contain only one face. Please try
                         again.'
            }
        embeds.append(imgProcessor.encode(np.array(faces[0])))

    _ = dbase.receive_descriptors(db, db_conn, name, embeds)

    global ids, descriptors, persons_ids
    ids, descriptors, persons_ids = dbase.read_descriptors(db)
    return recEngine.make_base(descriptors)
```

Pri pokretanju servisa pokreće se *main* funkcija. Ono što je ovde bitno napomenuti je da se u njoj inicijalizuju instance svih klasa, i da se u njoj mora dozvoliti TenorFlow biblioteci da koristi grafičku karticu i da ne alocira memoriju unapred. Tensorflow-gpu podržava samo kartice sa CUDA podrškom, tako da je sistem namenjem računarima sa NVIDIA grafičkim karticama. U suprotnom će biti prisutan veliki pad u performansama. U nastavku sledi kod *main* funkcije.

```
# Create the application instance
app = Flask('FR APP')
app.config['CORS_HEADERS'] = 'Content-Type'
app.config['UPLOAD_EXTENSIONS'] = ['.jpg', '.png']
cors = CORS(app)

# initialize empty lists
ids, descriptors, persons_ids = [], [], []

if __name__ == '__main__':

    # remove old log file
    os.remove("FRAPP.log")

    # set environment variables
    os.environ['FLASK_ENV'] = 'development'
    os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'

    # turn off tensorflow logger
    logger = tf.get_logger()
    logger.disabled = True
    logger.setLevel(logging.FATAL)

    # open file for logging
    logging.basicConfig(filename='FRAPP.log', level=logging.DEBUG,
                        format='%(asctime)s %(levelname)-8s
                                %(message)s')

    # allow GPU memory grow
    gpus = tf.config.experimental.list_physical_devices('GPU')
    if gpus:
        try:
            os.environ['CUDA_VISIBLE_DEVICES'] = '0'
            for gpu in gpus:
                tf.config.experimental.set_memory_growth(gpu, True)
```

```
        except RuntimeError as e:
            logging.info(str(e))
            print(e)

    logging.info('Allow GPU memory grow successful.')

    # parse arguements
    parser = argparse.ArgumentParser(description='Process some
                                                  arguments.')
    parser.add_argument('--cdp', type=str, help='the path to config
                                                 file')
    logging.info('Parsing arguments successful.')

    args = parser.parse_args()
    config_path = args.cdp

    cfg = config.readConfig(config_path)
    db_conn, db = dbase.db_connect(cfg["host"], cfg["port"],
                          cfg["name"], cfg["user"], cfg["password"])

     # initialize image processor
     imgProcessor = ImgProcessor(cfg)
     # initialize recognition engine
     recEngine = RecognitionEngine(cfg['threshold'])
     logging.info('All models initialized successfully.')
    try:
        # read ids, descriptors and person_ids from database
        ids, descriptors, persons_ids = dbase.read_descriptors(db)
        logging.info('Read descriptors successful.')
        recEngine.make_base(np.array(descriptors))
        logging.info('Make base successful.')
    except:
        logging.info('The database is empty.')

    # Run the flask rest api
    # This can be updated to use multiple threads or processors
    # In addition, some type of queue should be used
    # print starting text
    ascii_banner = pyfiglet.figlet_format("F R     A P P", font="slant")
    print(ascii_banner)

    logging.info('FR APP IS RUNNING.')
    logging.info('---------------' * 4)
    # threaded=False, processes=3
    app.run(debug=True, host='127.0.0.1', port=5000)
```

Svi delovi ovo sistema su konfigurabili. Putanja ka fajlu sa konfiguracijom se prosleđuje kao parameter pri pokretanju servisa. U nastavku sledi objašnjenje bitnih delova JSON konfiguracionog fajla.

Promenom vrednosti pod ključem *face_detector* se može promeniti detektor. Postavljanjem vrednosti na *RetinaFace* koristi se RetinaFace detektor, postavljanjem vrednosti na *MTCNN* koristi se Multi-task Cascaded Convolutional Networks (Kaipeng

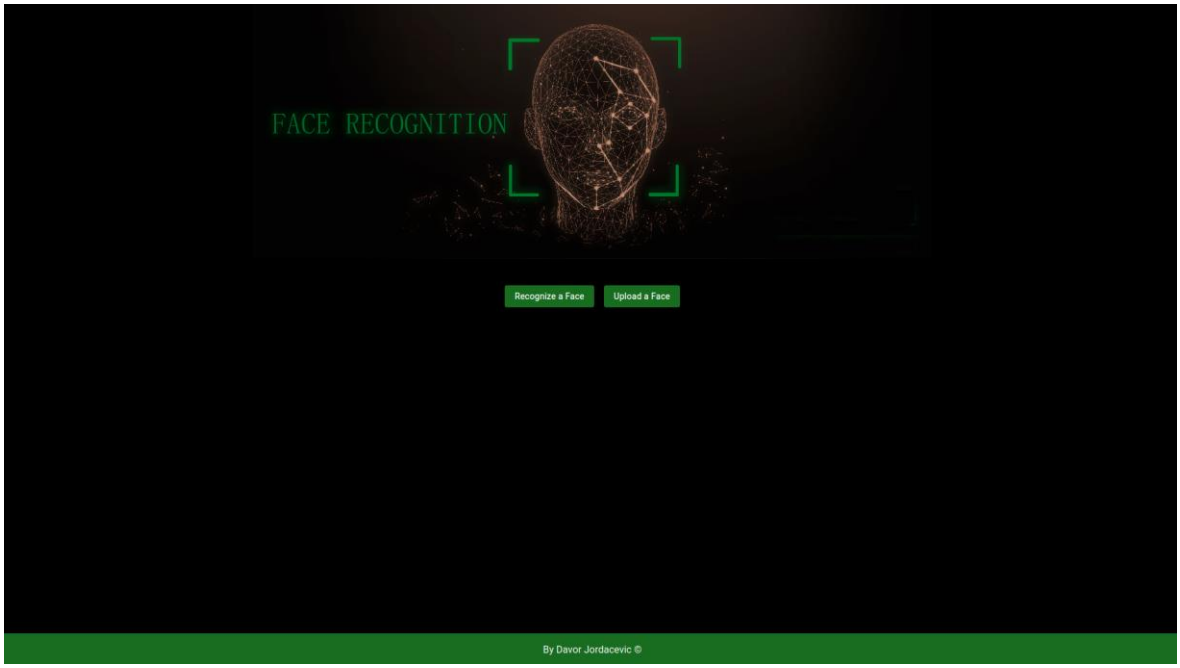
Zhang, Zhanpeng Zhang, Zhifeng Li, Yu Qiao, arXiv:1604.02878v1) detektor, i postavljanjem vrednosti na *SSD* koristi se Single Shot Object Detector (Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg, arXiv:1512.02325v5).

IoU prag za detekciju se može promeniti preko ključa *face_det_iou_th*, dok se prag za detekcije može promeniti preko ključa *face_det_score_th*.

Promena vrednosti praga za prepoznavanje moguća je promenom vrednosti pod ključem *threshold*.

```
{
  "host": "127.0.0.1",
  "port": 5432,
  "name": "FRAPP",
  "user": "dbuser",
  "password": "dbpassword",
  "face_detector": "RetinaFace",
  "face_det_model_path": "",
  "opencv_face_detector_uint8": "opencv_face_detector_uint8.pb",
  "opencv_face_detector_pbtxt": "opencv_face_detector.pbtxt",
  "threshold": 0.5,
  "face_reco_cfg_path": "./arcface_tf2/configs/arc_mbv2.yaml",
  "face_reco_checkpoints_path": "./arcface_tf2/checkpoints/",
  "face_det_cfg_path": "./retinaface_tf2/configs/retinaface_mbv2.yaml",
  "face_det_checkpoints_path": "./retinaface_tf2/checkpoints/",
  "face_det_iou_th": 0.4,
  "face_det_score_th": 0.6,
  "face_det_down_scale_factor" : 1.0,
  "write_to_file": "false",
  "experimental" : "false",
  "debugMode": "false",
  "log": "./",
  "ssl_mode": "false",
  "version": "0.1dev"
}
```

Za potrebe demonstracije sistema, kreiran je frontend deo koristeći ReactJS biblioteku i JavaScript programski jezik. Namena ovog servisa je uploudovanje slike i pozivanje odgovarajućeg endpointa za prepoznavanje, ili uploudovanje slika i slanje imena osobe radi ubacivanja osobe u bazu podataka.



Slika 19. Izled frontend-a

7. Zaključna razmatranja

U svom istraživanju u ovom radu došao sam do sledećih važnih nalaza.

RetinaFace, kao i ArcFace mreže su u ovom radu bazirane na MobileNet arhitekturi. Ova arhitektura pruža bolje performanse od prethodno pomenutog ResNet-a, ali to nije reč sa preciznošću. Prvi korak u postianju tačnijeg sistema bi bio implementacija ResNet arhitekture.

TensorFlow ima mogućnost konvertovanja modela u TensorFlow Lite, što omogućava optimizaciju modela za uređaje male snage, što podrazumeva znatno bolje performanse na računarima koji poseduju Tensor processing unit (TPU). Pored ovoga, NVIDIA-in set alata pod nazivom TensorRT pruža znatno veći set mogućnosti za optimizaciju modela, ali je direktno vezano za platformu na kojoj se koristi, te se ne može konverzija uraditi unapred.

N2 paket koji je korišćen u ovom radu podržava jako mali broj dostupnih distanci. Premda angular distanca pruža bolje performanse, kod vektora slika koji su jako blizu može doći do grešaka (slučaj kod većih setova podataka), pa je bolja opcija koristiti pakete koji podržavaju cos-inusnu distancu, ili je implementirati i dodati u N2 paket.

Iako je MobileNet arhitektura brza, implementacija korišćena u ovom radu ne postiže SOTA rezultate, te se performanse mogu dodatno poboljšati korišćenjem originalne implementacije koja koristi dodatne trikove za optimizaciju.

RetinaFace mreža sa MobileNetV2 arhitekturom korišćena u ovom radu je kao trening set podataka koristila WiderFace dataset, dok je ArcFace mreža sa MobileNet arhitekturom trenirana na MS-Celeb-1M setu podataka. Premda ovi setovi podataka sadrže veliku količinu podataka, za najbolje rezultate je potrebno dodati setove koji sadrže uniformno distribuirane slike za sve rase. Ovo je jako bitno za jedan produkcioni sistem zbog takozvanog bias-a koji se javlja ukoliko setovi podataka nemaju uniformnu distribuciju.

Za jedan produkcioni sistem je bitno korišćenje anti-spoofing metoda kako bi se zaštitili od napada. Sistem poput ovog je modularan, pa se ubacivanjem dodatnog modula između dela za detekciju i dela za prepoznavanje ovo može postići. Treba imati na umu da su implementacije anti-spoofing sistema kompleksne, i većina SOTA modela koristi RGB, depth i IR slike.

Delove koda koji se često ponavljaju, kao i delovi koji isključivo rade sa Numpy nizovima, mogu se prepraviti da koriste CuPy paket koji omogućava korišćenje grafičke kartice pri radu sa nizovima, ili iskoristiti brzinu C programskog jezika korišćenjem Cython paketa i refaktorisanjem koda. Drugo rešenje može biti korišćenje Numba kompajlera koji osim što prevodi kod u mašinski, omogućava i paralelizaciju koda.

Delovi koda se mogu prepraviti po konkurentnom principu kako bi se izbeglo zaključavanje procesa i čekanje odgovora. Takođe, premda lak za implementaciju, Flask razvojni okvir je namenjem web programiranju, i nije pogodan za veliki broj zahteva koje bi ovaj sistem mogao da očekuje, te bi njegova zamena bila neophodna.

Trenutni sistem nije skalabilan, a kako bi se ovo omogućilo, potrebno je implementirati mehanizme koji će skladištiti sve zahteve i raspoređivati nekom procesu po potrebi. Ovi alati su poznati pod nazivom brokeri poruka.

Sistemi dubokog učenja su zahtevni za deployment i održavanje. Rešenje koje je idealno za ovakve sisteme je korišćenje Docker-a, odnosno principa kontejnera. U ovom slučaju ne samo da bi se samo Python deo sistema morao prebaciti u kontejner, već i cela baza i frontend deo.

Do sada je bilo reči o radu sa slikama. Ukoliko ima potrebe raditi sa video snimcima, situacija postaje komplikovanija. Premda video snimak jeste samo niz slika, i moguće je uraditi prepoznavanje lica na svakoj slici (frejmu), ovo je prilično zahtevan posao za računar. U ovim slučajevima je potrebno koristiti tehnike praćenja (eng. tracking) i reidentifikacije.

Literatura

1. Deng, Jiankang, et al. "Retinaface: Single-stage dense face localisation in the wild." *arXiv preprint arXiv:1905.00641* (2019).
2. Sandler, Mark, et al. "Mobilenetv2: Inverted residuals and linear bottlenecks." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018.
3. Tang, Xu, et al. "Pyramidbox: A context-assisted single shot face detector." *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018.
4. Lin, Tsung-Yi, et al. "Focal loss for dense object detection." *Proceedings of the IEEE international conference on computer vision*. 2017.
5. Deng, Jiankang, et al. "Retinaface: Single-stage dense face localisation in the wild." *arXiv preprint arXiv:1905.00641* (2019).
6. Dai, Jifeng, et al. "Deformable convolutional networks." *Proceedings of the IEEE international conference on computer vision*. 2017.
7. Schroff, Florian, Dmitry Kalenichenko, and James Philbin. "Facenet: A unified embedding for face recognition and clustering." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015.
8. Deng, Jiankang, et al. "Arcface: Additive angular margin loss for deep face recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019.
9. Malkov, Yury A., and Dmitry A. Yashunin. "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs." *IEEE transactions on pattern analysis and machine intelligence* (2018).
10. Zhang, Kaipeng, et al. "Joint face detection and alignment using multitask cascaded convolutional networks." *IEEE Signal Processing Letters* 23.10 (2016): 1499-1503.
11. Liu, Wei, et al. "Ssd: Single shot multibox detector." *European conference on computer vision*. Springer, Cham, 2016.