Arcfelismerő rendszer kliens-szerver architektúra alapon

Romhányi Ármin

2020

Tartalomjegyzék

1. Bevezetés

1.1. A vastagkliensben rejlő lehetőségek

A bevezető alapján tehát egy olyan *többrétegű architektúrával*[1] rendelkező szoftverrendszer kialakítása a cél, mellyel nem csak az arcok képekből történő kinyerése lehetséges, hanem azok összehasonlítása is. Ehhez legalább két komponens implementálása szükséges: a mobilkliensé, mely elkészíti a képet és megkezdi annak előfeldolgozását, illetve a szerveré, mely elemzi a megkapott adatot. Tradicionálisan, az ehhez hasonló rendszerekben a képek releváns részeinek felismerését nem a mobileszközökre bízták azzal érvelve, hogy az eszközök számítási - illetve adattárolási kapacitása nem elegendő a feladat megoldásához [1]. Ehelyett ezt a feladatot ugyanazon komponens (vagy komponens réteg) végezte el, mely a következő lépcsőfoknak számító összehasonlítást is: a központi szerver. Való igaz, hogy (jelenleg) ezek az eszközök az arcok közötti különbségek felismeréhez szükséges kiterjedt adatbázisok tárolására alkalmatlanok és belegondolva eme adatbázisok folyamatos frissíthetőségének és biztonságának szükségességébe, egy ilyen elvű megoldás nem is célszerű. Azonban a technológia fejlődésének köszönhetően ma már az átlagosnak számító mobil eszközök is rendelkeznek a szerver adatkinyerő funkciójához szükséges kapacitással, így érdemes megvizsgálni, hogy - némileg áthelyezve a feladatköröket elválasztó határvonalat - milyen előnyökkel, illetve hátrányokkal járhat egy olyan rendszer alkalmazása, melynél a mobilkliens végzi el a *prepocesszálásnak* nevezett folyamatokat.

Habár biztonsági szempontból is elemezni lehet egy ilyen rendszer előnyeit, - hiszen a kinyert adatokat jóval könnyebb titkosítani, majd továbbküldeni - azonban ugyanezzel az erőráfordítással komolyabb titkosítással is le lehet védeni a képeket (pl. AES[2]), így ez az érv nem igazán meggyőző. Bármilyen vizsgálat nélkül belátható viszont, hogy egyértelmű előnyökkel járhat egy ilyen rendszer, ha valamilyen okból kifolyólag korlátozott internet sávszélesség áll rendelkezésre, vagy nagy számú mintát kell a küldőtől a fogadóig eljuttatni. Ebben az esetben a kép helyett - melyek a feladat bonyolultságából adódóan egészen nagy méretűek is lehetnek - elég lenne csak a feladat megoldásának szempontjából megfelelő adatokat továbbítani. Ehhez kapcsolódóan másik jelentős előny lehet az adott rendszer kiértékelési hibáinak csökkentése, melyek az adatátvitel során fellépő veszteségek miatt állnak elő. Ennek triviális megnyilvánulása, hogy az alacsonyabb volumenű adatátvitelnek köszönhetően vélhetően csökken a hibalehetőségek száma is, azonban ha a feladat megkövetel a rendszertől valamilyen integritás ellenőrző mechanizmust is (pl. a kép pixeladataiból MD5 hash számítása, majd ezt összevetni a célállomáson számított hash-el) mely

[1] multi/N-tier architecture
[2] Advanced Encryption Standard

hiba esetén újrakéri az adatokat, akkor a sávszélességből adódóan már többszörös különbségekről van szó. Ezeken kívül előfordulhatnak olyan feladatok is, melyekhez ma már nem feltétlenül szükséges hálózati kapcsolat, azonban az arcképből kinyert adatok igen: például arcképből kinyert adatok használhatóak érzelem felismerésre [2], arckifejezések felismerésére [3], vagy akár lokális hazugságvizsgáló rendszerekhez is [4]. Ugyan eme utolsó gondolat érvül szolgálhatna egy teljesen független kliens kialakításához is, nem nehéz belátni, hogy kombinálva a szerver-kliens kialakítás nyújtotta centrális adatelosztás nyújtotta lehetőségekkel (pl. kiterjedt adatbázisok, statisztikák) ma már van létjogosultsága egy ilyen rendszernek.

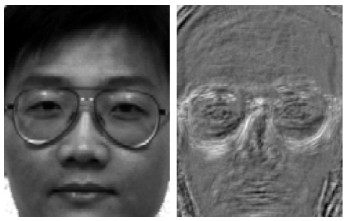
További előny lehet még, hogy üzleti szempontból egy ilyen rendszer biztosíthatja, hogy a fellépő igényekhez szabják annak kialakítását és kompozícióját: új komponensek tetszőleges mértékben adhatók és vehetők el, amennyiben egy közös bemeneti és kimeneti réteggel vannak ellátva.

A jelen dolgozat témája - az előbb felsorolt érveket szem előtt tartva - egy ilyen rendszerhez szükséges ismeretek összefoglalása és ezek gyakorlati felhasználhatóságának vizsgálata, majd az erre alkalmas eszközök áttekintése után annak implementálása és végül a megépített rendszer elemzése. A dolgozat tematikája is ezt a nyomvonalat fogja követni.

2. Az arcfelismerés elvi módszerei

Eme téma kérdésköre szerteágazó és hatalmas irodalmat ölel fel, specifikus ismereteivel szakkönyvek egész sora foglalkozik. Épp ezért nem meglepő, hogy a téma több, jóllehet eszközkészletében megegyező tudományágat is rejt: *arcfelismerésnek*[3] nevezik a területet mely az emberi arc felismerésének formai és alaki követelményeinekk meghatározásával, annak háttérből történő elkülönítésével foglalkozik, míg ennek specifikus ága az *arcalapú azonosítás*[4], mely az előbbi vívmányait felhasználva próbálja eldönteni különböző arcokról, hogy azok mennyiben különböznek, illetve tartozhatnak-e ugyanazon személyhez. Habár eme 2 fogalom elkülönítése nem szigorúan tudományos jellegű - különösképp, hogy a szakirodalom is némiképp összemosva használja őket -, a dolgozat további részében megkülönböztető jelleggel bírnak majd.

2.1. Alkalmazott technikák



1. ábra. Yala arcadatbázis egy mintája és szemüvegességet megállapítani hivatott Fischer - arc (forrás: [5])

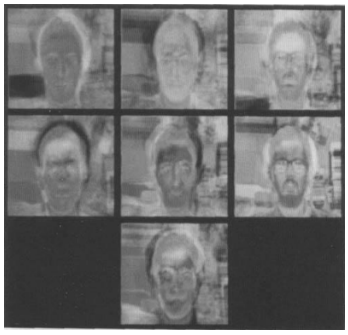
Az arcfelismerés tudományának története során többféle technikával is megkísérelték megoldani a feladatot. Bármilyen különböző is legyen egy-egy technika azonban, a legtöbb hátterében az alakfelismerésben jártasak számára jól ismert technika található, melynek során az azonosítandó alakból kiragadnak bizonyos jól beazonosítható jegyeket, melyek nyomait keresik aztán az azonosítás során. Ennek jegyében a legkorábbi megoldások az arc geometrikai jellegzetességein alapuló módszereket vetettek be a siker érdekében: gyakori volt az orr és a száj közötti távolság mérése (pl. [6]), de előszeretettel alkalmaztak az ezen arcjegyekből képzett sablonnal különböző képrészletek összehasonlításán alapuló praktikákat is (részletesebben [6]), nem is beszélve ettől kevésbé konkrét módszerekről, mint amilyen a különféle arcokra vetített szűrők rácshálókon képződött képei (pl. [7]).

Ehhez képest merőben eltérő elvet képviselnek a *lineáris projekción* alapuló tehnikák, melyek az eredeti képeket valamilyen leképezés alapján egy, az „eredetinél alacsonyabb dimenzióba” [5] vetítik le kiiktatva egyes, a módszer szempontjából jelentéktelennek vélt tulajdonságokat, miáltal a releváns jegyek könnyebben elemezhetőek különböző klaszterekbe, vagy osztályokba szervezve. Eme technikák legjelentősebb vívmánya, hogy az arc felismerhetőségét befolyásoló tényezőket (pl. meg-

[3] facial recognition
[4] facial authentication

világítás) ki lehet velük küszöbölni bizonyos mértékig. Ilyen technika például az úgynevezett *sajátarcok*[5] használata, melynek során több, ugyanabba a kategóriába tartozó arc lineáris kombinációjából képeznek egy, az adott kategóriát reprezentáló arcot, melyet aztán a beazonosítani kívánt arcképpel hasonlítanak össze (pl. a képekből kinyert együtthatókból skaláris szorzat képzésével) [8]. Ehhez hasonló az [5] által *Fischer-arcoknak*[6] nevezett elemeket alkalmazó módszer, mely az alakfelismerésből ismert technikákkal javítani próbálja bizonyos, a klasszifikáció során szerepet játszó elemek csoportosíthatóságát

Az általánosabb értelemben vett arcfelismeréssel szemben az arc alapú autentikáció sikerességében sokkal nagyobb szerepet kapnak a különféle arcok közötti minimális különbségek, nem csak az úgynevezett *interperszonális* - azaz személyek közötti - eltérések, de akár egyazon személy különböző arcképei között is jó eséllyel fennállhat olyan fokú *intraperszonális* különbség, mely megnehezíti, vagy egyenesen lehetetlenné teszi a művelet sikeres végrehajtását. Nem meglepő hát, hogy az évek során sokféle technikát kifejlesztettek a terület kutatói. Ezen autentikációs technikákat [9] 2 csoportba sorolja: *diszkriminatív* és *generatív*.



2. ábra. sajátarcok (forrás: [8])

Az általa az előbbi csoportba sorolt módszerek célja a teljes bemeneti arckép adatait felhasználva egyszerű gépi válaszokhoz jutni, azaz, egy adott képen az azonosítani kívánt személy található-e avagy sem. Ezek a technikák vagy az eredeti, szürkeárnyalatos arcképeket használják fel, vagy pedig annak valamilyen módszerrel átalakított változatát (pl. sajátarcok). Ezzel szemben a generatív módszerek megpróbálják kiszámítani, hogy a teljes arc, vagy annak a vizsgálat szempontjából fontos részletei milyen valószínűséggel találhatóak meg egy adott bemeneti mintán, s így az autentikációs folyamat végeredménye is egy százalékos arányszám. Az ilyen jellegű folyamatok akkor sikeresek, ha a végeredményként kapott arányszám egy bizonyos küszöbérték felett van.

Egyértelmű tehát, hogy az arcfelismerés, de méginkább az arc alapú felismerés és autentikáció témaköre nem csak szerteágazó problématérrel rendelkezik, de az alkalmazható megoldási módszerek is rendkívül sokszínűek, mi több, a tudományág kutatói folyamatosan új technikákon és elveken dolgoznak ezzel is tágítva a tudomány határait. Mindemellett nem szabad azonban elfelejteni, hogy eme módszereknek nem csak tisztán tudományos értéke van, hanem valódi, egyéb területekre átívelő hatása is: talán legjelentősebb ezek közül a bűnüldözés és az igazságszolgálta-

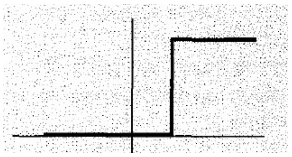
[5] Eigenface
[6] Fisherface

tás támogatása. Könnyen belátható viszont, hogy minél szélesebb az alkalmazandó technikák spektruma, annál nehezebb a való életben is hasznosítható megoldásokat alkotni velük, melyek nem csak laboratóriumi körülmények között képesek eredményeket előállítani. Pontosan ilyen okoktól vezérelve, konkrét feldolgozó algoritmusok helyett a kutatók többsége valamilyen *neurális hálón* alapuló megvalósítást alkalmaz (pl.*többrétegű perceptron* [9], vagy *mélytanulás*[10]), így érdemes áttekinteni ezen módszerek alapjait.

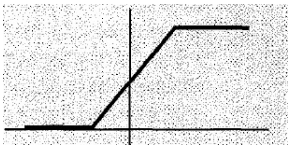
3. Arcfelismerés a gyakorlatban

3.1. Neurális hálók

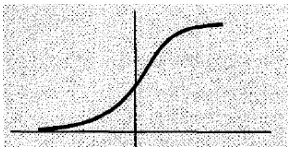
A mesterséges neurális hálók[7] a fejlettebb élő szervezetek agyát alkotó neuronokról kapták nevüket. Egy biológiai neuron képes elektromos jeleket fogadni és továbbítani a többi neuron számára *szinapszisain* keresztül lehetővé téve így az ingerületátvitelt [11]. A neuronok mind a fejlettebb állatok, mind pedig az emberek agyában rendkívül kiterjedt hálózatot alkotnak (biológiai neurális háló): különböző becslések szerint egy neuron akár 10^3 - 10^4 más neuronhoz is kapcsolódhat, míg az emberi agy hozzávetőleg 10^{14} - 10^{15} nagyságú neuronszámmal rendelkezhet [11]. Az ANN-ek szempontjából a neuronok egy másik jelentős tulajdonsága, hogy az információt kisebb darabokban továbbítják bármekkora is legyen annak teljes mérete. Ebből a terület kutatói arra a következtetésre jutottak, hogy a neuronok nem közvetlenül, hanem egymás között lévő kapcsolataik révén továbbítják az információt [11]. Ezek a megállapítások adták az alapot az ANN-ek megalkotásakor.



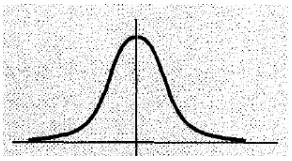
(a) Küszöbfüggvény



(b) Szakaszos lineáris



(c) Szigmoid



(d) Gauss

3. ábra. Az egyszerű küszöbfüggvény és néhány aktivációs függvény, forrás: [11]

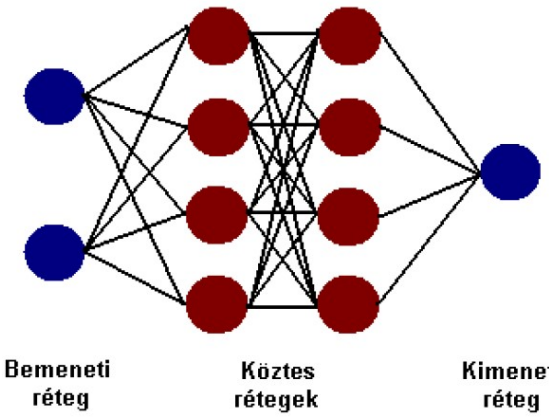
Az első neuronok matematikai modelljei egyszerűek voltak, egy bizonyos számú, súlyozott bemeneti jelre *1* kimenetet adtak, ha a bemenetek összege meghaladott egy előre megadott *u* értéket, különben 0-val tértek vissza [11]. Röviden, a korai neuronok[8] viselkedését Jain *et al.* így foglalja össze (p. 4):

[7] artificial neural network, ANN

[8] A szakirodalomban ez a modell *McCulloch-Pitts-féle neuron modell* néven ismert

$$y = \theta \left(\sum_{j=1}^{n} \right) w_j x_j - u$$

ahol θ egy *egységugrás függvény* 0-ban, x_j a j-edik bemenet, w_j az ehhez társított szinaptikus súly[9], u pedig maga az elérni kívánt határ.



4. ábra. Egy több rétegű neurális háló elméleti szerkezete, forrás: [13]

A gyakorlatban a $-u$ küszöbértéket w_0-ként szokás jelölni, és mint konstans, x_0 értékű súly hivatkoznak rá [14], Jain *et al.* azt is hozzáteszi, hogy a különböző előjelű súlyoknak más jelentésük van a neuron viselkedése szempontjából: pozitív súly esetén serkentő hatás éri a neuront az adott szinapszisán, míg negatív esetén gátlás (p. 4). A mai gyakorlatban a bemenetek száma pedig az eredeti modellhez képest teljesen eltérhet egymástól (de egyik elemszáma sem lehet 0) [13].

Habár ez a modell bizonyítottan alkalmas volt általános célú számítások végrehajtására [11], számos olyan egyszerűsítést vett alapul, melyek nem tükrözik a biológiai neuronok működését, így mai formáját különböző általánosítások után nyerte el. Ilyen például, hogy a küszöbfüggvény helyett különféle *aktivációs függvényeket* használnak. Akárcsak a küszöbfüggvény, ezek a függvények állítják elő az adott neuron bemenetre adott válaszát [15]. Ma az egyik leggyakrabban használt aktivációs függvény a *szigmoid* [13], melyet a következő képlettel határoznak meg:

$$y = \frac{1}{1 + e^{-(a-\theta)b}}$$

ahol a az aktiválás, b pedig a görbe alakját szabályozza.

A neurális hálók egyik legfontosabb tulajdonsága tanulási képességük, melynek alapja, hogy többrétegű hálókba szervezik őket és a különböző rétegek neuronjai kommunikálnak. Ezek a hálók általában (bár nem kizárólag) *bemeneti rétegből*, 1 *kimeneti rétegből* és az ezek között található *köztes rétegekből* állnak. A bemeneti, azaz a neurális háló inputrétegében minden neuron kapcsolatban áll a köztes réteggel, így tovább adhatja a bemenetként megkapott adatokat [13]. A bemeneti réteg neuronjai súlyozott szinapszisokkal kapcsolódnak a belső rétegekhez. E kez-

[9]Két neuron között lévő kapcsolat erőssége [12]

dő súlykiosztás meglehetősen feladatspecifikus és az alapján kerül kiosztásra, hogy az adott probléma szempontjából milyen tulajdonságok fontosak. A tanulás képessége abban nyilvánul meg, hogy míg a bemeneti réteg neuronjai előre beállított súlyozással rendelkeznek, addig a köztes rétegben a súlyok folyamatos változásban[10] vannak. E változásokat a háló rendelkezésére bocsátott példák, úgynevezett *tanító minták* befolyásolják [11]: „különböző, a kutatók által betáplált szabályok követése helyett úgy tűnik, [az ANN-ek] a megadott reprezentatív mintából tanulják meg a szabályokat"[11] (p. 4), másszóval, a neurális hálók tanulása tulajdonképpen a köztes elemek dinamikus súlykiosztásának folyamata.

A tanulás kezdeti stádiumában a neuronok véletlenszerű választ adnak a problémára, majd összehasonlítják az eredményt a tanítómintával és ez alapján változtatják meg a belső súlyozást [13]. Figyelemre méltó eredményeik ellenére a neurális hálók esetében nem garantált, hogy a folyamat végén keletkező kimenet az optimális eredmény: a tanulás folyamán a sorozatos mutációk által a háló beáll egy bizonyos *lokális minimum* értékre [13].

A hálók betanítása során alkalmazott módszerek közül a leggyakoribbak a *felügyelt*[11], a *felügyelet nélküli*[12], illetve az ezek kombinációjából kialakult *hibrid* tanítási módszerek, melyek a háló rendelkezésére álló információ minőségében térnek el. Az első módszer esetében a tanítóminták az elvárt kimenetek halmaza (az információmennyiség szempontjából ez a legmagasabb minőségű tanítóminta), a háló feladata pedig ezekhez minél hasonlóbb eredményeket előállítani a súlyozás manipulálásával [11]. Ennek egy variánsa a *megerősítéses tanulás*[13], melyben a háló nem kapja meg az elvárt eredményt, hanem egy az aktuális eredmény és az elvárt eredmény eltéréséből számított érték alapján dolgozik. Felügyelet nélküli tanulás esetén a hálók feladata nem az, hogy különböző bemenetekre kimeneteket generáljanak, hanem a tanítómintában rejlő belső mintázatokat és az azok közötti kapcsolatokat kell feltárniuk. Hibrid tanulás esetén a súlyok egy részét irányított tanulással állapítják meg, másik részüket pedig irányítás nélkül térképezik fel [11]. A hálók gyakorlati alkalmazásai alatt tulajdonképpen az értik, hogy a megfelelően kiképzett hálóknak olyan hasonló szerkezetű adatokat adnak meg, melyekkel eddig még nem találkoztak, az pedig a tanítási folyamat során felhalmozódott tapasztalatai alapján megoldja a feladatot.

[10] A köztes elemek a bemenetek különböző lineáris kombinációiként állnak össze [13]
[11] supervised learning
[12] unsupervised learning
[13] reinforcement learning

3.2. Néhány népszerűbb neurális háló jellemzése

Az eltérő elméleti modellekhez hasonlóan a gyakorlati alkalmazásoknak is széles palettája áll a kutatók rendelkezésére. Mivel eme alkalmazási módszerek többféle tulajdonság szerint is elemezhetőek (pl. topológia, rétegmélység, neuronösszetétel, információ terjedési iránya), így a jelen leírás - a teljesség igénye nélkül - az arcfelismeréssel kapcsolatos kutatásokban gyakran előtérbe kerülő tulajdonságokra koncentrál. Ennélfogva, *irodalom* kifejezéssel is az arcfelismerést illető tudományos szakirodalom kerül megnevezésre.

Attól függően, hogy milyen irányba áramolhat a háló neuronjai között az információ, az irodalomban kétféle konfiguráció fordul elő gyakran: úgynevezett *előreterjesztéses*[14] (pl. [16], [17], vagy [18]), illetve *recurrent* (pl. [19], [20]) hálók. Előbbi hálók neuronszerkezete körmentes gráfot alkot, így a kimenetek input irányból output irányba áramolnak, míg az utóbbi hálók esetében a szerkezetben több irányú kapcsolatokkal rendelkező neuronok is helyet kapnak, így szerkezetük nem marad körmentes. Ahogy korábban már erről szó esett, a neuronok itt is mindkét esetben rétegekben helyezkednek el és az előttük lévő rétegek által generált inputokból állítanak elő kimeneteket. Ezek a kimenetek végigáramlanak a háló teljes hosszában, miközben folyamatosan összehasonlításra kerülnek az elvárt eredménnyel. Ennek eredménye az úgynevezett *veszteség*[15], mely az elvárt és a kapott eredmény különbsége. A performanciavizsgálat során keletkező hibaarány visszaáramlik a korábbi rétegek felé egy *backpropagation-nek* nevezett eljárás során, ami a különböző bemeneti neuronok fontosságát jelképező súlyok dinamikus változásában játszik szerepet [16]. Az ilyen hálók tehát lényegében a súlyok újra és újra elosztásán keresztül önmaguknak tanítják meg, hogy mely elemek milyen arányban vesznek részt a probléma megoldásában.

A *topológia*[16] egy, a hálót az azt alkotó neuronok közötti kapcsolatok szempontjából jellemző tulajdonság, mely nagy befolyással bír a tanulási folyamat egészére [21]. Ennek leginkább kézzel fekvő megnyilvánulása a hálót alkotó rétegek száma (mélysége) és - habár egzakt számok általában nem kerülnek említésre - kellően nagy rétegszámmal rendelkező hálókat *mély neurális hálónak*[17] szokás nevezni, a rájuk jellemző tanulási folyamat pedig a *mélytanulás*[18]. Ezen típusú hálók szinten reprezentálva vannak az irodalomban [10].

Egy másik, az irodalomban nagy előszeretettel alkalmazott [22] topológiai jellegzetesség az úgynevezett *konvolúciós* rétegek megléte. Az ilyen elemekkel rendelkező

[14] feedforward
[15] *loss*
[16] network topology
[17] deep neural network
[18] deep learning

konvolúciós neurális hálók[19] a matematikában konvolúciónak nevezett folyamatról kapták a nevüket, melynek során 2 bemeneti függvényből előáll, hogy - sorrendtől függően - egyik függvény hogyan módosítja a másikat. A CNN-ek is több rétegű hálók, melyekben a bemenetként kapott adatot első körben az input réteg átalakítja *tulajdonság leképezésekké*[20], melyek a bemenet egy-egy, a CNN által észlelt részletét reprezentálják (pl. kép élei). A háló rejtett rétegei aztán ezeket a map-eket felhasználva újabb map-eket állítanak elő, míg végül a kimeneti réteg - amely elég gyakran valamilyen osztályozó[21] réteg - megállapítja, hogy az így kapott leképezések mekkora eséllyel tartoznak az előre megállapított kategóriákba. A CNN típusú hálók rejtett rétege közül kerülnek ki a konvolúciós (szűrő) rétegek, amik lényegükben integer mátrixok és feladatuk, hogy a konvolúciós *kernel* által kijelölt „hatáskörükbe tartozó" pixel területet megszorozzák a mátrixban a megfelelő számmal, majd a kernel méretébe eső pixelek összegét továbbítsák az adott réteg kimeneteként előálló feature map-be [23].

A fönti jellemzések könnyen azt a benyomást kelthetik, hogy az irodalomban alkalmazott hálók vagy egyik, vagy másik kategóriába tartoznak, ez azonban nem is lehetne távolabb az igazságtól: inkább úgy érdemes eme tulajdonságokra gondolni, mint egy dobókocka oldalaira. A leggyakrabban alkalmazott konfiguráció például a feedforward CNN [24], [25], az alkalmazott CNN-ek közül pedig több mélytanuláson alapszik [26], vagy [27], de akadnak recurrent CNN alkalmazások is [22].

Végezetül, érdemes megemlíteni, hogy nem csak neurális hálón alapuló megoldások léteznek, [28] például *rejtett markov modelleket* használt, melyek (igen nagy vonalakban) a bemeneti adatokon minták előfordulásának valószínűségéhez köthetőek, de szintén a statisztikából ismert *Bayes-hálók* is alkalmasak a feladatra [29], ahogy egyéb más technikák is, melyek név szerinti említése már túlmutat a jelen fejezet korlátain.

Az előbbiekben említettek alapján már nem nehéz belátni, hogy neurális hálók alkalmazásával javítható (ha nem is küszöbölhető ki teljesen) az arcfelismeréses technikákban rejlő bizonytalansági faktor. Ahhoz azonban, hogy a rendszer rendelkezzen a megfelelő hatékonysággal, nem elegendő csupán kihasználni a kiválasztott technológia kínálta lehetőségeket, ugyanis a neurális hálókon alapuló technológiák megfelelő előkészítés nélkül rugalmasságuk ellenére sem tudnának megfelelő eredményeket produkálni. A elkövetkezőkben eme előkészületek kerülnek rövid bemutatásra.

[19] convolutional neural network, CNN
[20] feature map
[21] classifier

4. Arcfelismerés a szoftveriparban

A korábbi szekciókban megemlített implementációk jórészt tudományos ihletésű, laboratóriumi körülmények között tesztelt megoldások voltak. Akadnak közöttük persze olyanok is, melyek kifejezetten azzal a céllal készültek, hogy valós(ágot idéző) körülmények között produkáljanak eredményeket. Ilyenek például a különböző, kutatóknak szóló kihívások, melyek lényege, hogy előre beállított képek helyett „in the wild", azaz a hétköznapi életből, esetleg különféle közmédiából kigyűjtött mintákon kell minél jobb eredményeket felmutatni (ilyen challange-re beküldött munka például [30] is, melyben filmek audiovizuális mintáit elemzik érzelemfelismerés céljából). Jóllehet az ehhez hasonló munkák jóval közelebb állnak a valóságos kihívásokhoz, azonban alapvető céljuk mégis a tudomány előremozdítása, nem pedig piaci körülmények között is hasznosítható megoldások leszállítása. Ez természetesen távolról sem jelenti azt, hogy ne lehetne az ehhez hasonló projekteket kereskedelmi forgalomba hozható szoftverekké fejleszteni, ennek demonstrálására a következő szekció a már meglévő szoftveripari megoldások bemutatására koncentrál.

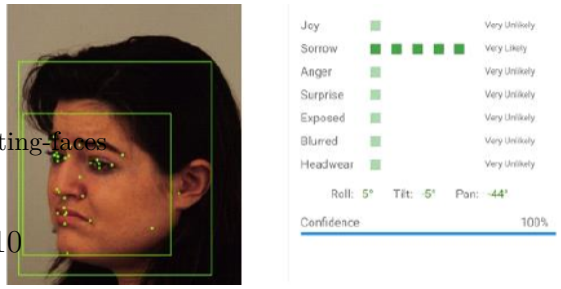
4.1. Felhő alapú megoldások

Az arcfelismerésen alapuló technológiáknak megszámlálhatatlan alkalmazási területük lehet, így egyáltalán nem meglepő, hogy az olyan szoftveróriások, mint a Google, a Microsoft vagy az Amazon nyújtanak valamilyen felhő alapú arcfelismerő *szolgáltatott platform*[22]/*szolgáltatott MI*[23] megoldást. Ezeket a szolgáltatásokat az alkalmazó szoftverek különféle *alkalmazásprogramozási felületekkel*[24]történő webes kommunikáció segítségével vehetik igénybe különféle konstrukciójú előfizetések alapján, melyekért cserébe javarészt általános célú arcfelismeréssel, landmarkpontok és arckontúrvonalak felismerésével, valamit érzelemfelismeréssel bíró szolgáltatást kapnak. Mivel a megoldások felhő mivolta garantálja, hogy sem a környezet, sem pedig a megoldást biztosító szoftver karbantartása nem a fejlesztő feladata, így az ilyen megoldások segítségével a legkevésbé komplikált arcfelismeréssel bíró szoftvereket piacra bocsátani, ám épp ezért ezen lehetőség egyben a legkevésbé alkalmas egyedi üzleti igények kielégítésére.

4.1.1. Google Vision API

A Google arcelemző szolgáltatása[25] - mely a szintén a Google által fejlesz-

[22]Platform as a Service, SaaS
[23]Artificial Intelligence as a Service, AIaaS
[24]application prgramming interface, API
[25]https://cloud.google.com/vision/docs/detecting-faces



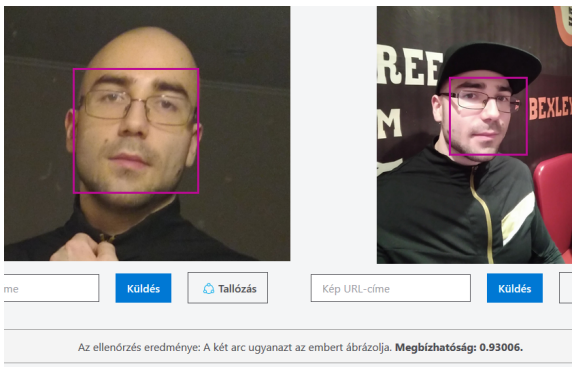
5. ábra. Google Vision API, forrás: [31]

tett Tensorflow mesterséges intelligencia keretrendszeren alapszik - széleskörű szolgáltatásokat biztosít: landmark pontok, arckontúrok és fejen hordott ruhadarabok detektálása mellett érzelemfelimserésre és híresség képeken történő beazonosítására is alkalmas, ellenben - mint ahogy arra a dokumentáció több helyen is felhívja a figyelmet - arcok összehasonlítására nem. Habár a platform zajtűrésével kapcsolatban fogalmaztak meg kritikákat [32], meredekebb beesési szögből is képes jó minőségű eredményeket produkálni, akár több alany esetében is. Maga a dokumentáció egyébként kifejezetten fejlesztőközpontú: többnyelvű, minimális példákon keresztül mutatja be az API használatát és része egy élő demonstrációként használható API végpont is. Az API használatához Google Cloud Platform hozzáférésre van szükség, melyen egy külön az adott termékhez illeszkedő projekt létrehozása szükséges.

4.1.2. Microsoft Azure Cognitive Services Face API

Habár a Microsoft felhő alapú fejlesztése[26] némileg elmarad a Google által kínálttól [31], így is rendkívül szolgáltatásgazdag megoldás: az alapvető arcfelismeréshez köthető tulajdonságok mellett mint a 27 landmarkpont, vagy az arckontúrok, a különféle, arcokhoz társítható elemek felismerését is támogatja mint az arcszőrzet (hajszínt és szakállt is beleértve), a szemüveg, vagy a fejviselet, de a képen található alanyok korára, nemére és érzelmi állapotára vonatkozóan is ad becslést. A megoldás olyan kategóriákat is magába foglal, mint a smink viselésének valószínűsége, de akár arcokat tartalmazó képek összehasonlítása és csoportosítására is van lehetőség autentikációval kapcsolatos problémák megoldásához [33].

A szolgáltatás weboldala egyébként felhasználóbarát, nem csak magas szintű nyelvi támogatással (köztük gépileg fordított magyarral), de kipróbálható API végponttal is rendelkezik, illetve kitér a szolgáltatás díjszabásaira is. Ugyan csak C# és Python nyelvű példatár áll rendelkezésre, SDK leírás található többféle technológiához is. Használatához Microsoft Azure előfizetés szüksé-



6. ábra. A szerző MSCS Face API által elemzett képei

[26] https://azure.microsoft.com/hu-hu/services/cognitive-services/face/

ges.

4.1.3. Amazon Rekognition

A cloudmegoldások megteremtőjének szolgáltatása[27] is képes a korábban említett alapvető szolgáltatásokra mint landmark pontok és a kontúr, illetve kiterjedt tulajdonságpalettával bír (pl. érzelmek, szemüveg, napszemüveg, nem, kor, szakáll, mosoly felismerése, vagy arcok összehasonlítása és híresség felismerése), azonban ezek mellett alapértelmezetten támogatja a videóanyagokból történő arcfelismerést is. Ugyan webfelületének nem része demo API végpont, azonban az előző cégekéhez hasonló, kiterjedt API dokumentációval és 3-4 nyelvű példakódbázissal segíti a fejlesztőket a megoldás saját környezetbe történő beintegrálásában. A Rekognition használatához Amazon Web Services account és IAM User account szükséges.

4.2. Személyes használatú szoftvermegoldások

A felhő alapú módszerek mellett természetesen *on-site* megoldások is rendelkezésre állnak, melyek alapértelmezetten nyújtanak arcfelismerő szolgáltatásokat, jóllehet, felhőbeli társaiktól jóval szerényebb képeségekkel bírnak és általában nem mesterséges intelligencia hajtja őket. Ezek többnyire nem önálló szoftverek, mint inkább különböző nyelvekre implementált keretrendszerek, és így használatuk valamilyen szintű programozás terén szerzett jártasságot feltételez, bár akad rá példa, hogy egy ilyen framework *stand-alone* szoftvercsomagot is tartalmaz. Közöttük vannak ugyan egyszeri pénzfizetést igénylő kereskedelmi szoftverek, azonban napjainkban egyre inkább elterjednek a nyílt forráskódú[28], ingyenesen felhasználható megoldások is. Jelen szakaszban ezek közül a két legnépszerűbb kerül bemutatásra.

4.2.1. Dlib

A dlib[29] egy alacsonyszintű, alapvetően C++ nyelvre írt, de Python támogatást is élvező *cross platform*, nyílt forráskódú szoftver, mely - általános mesterséges intelligenciával kapcsolatos könyvtárai részeként - arcfelismerésre használható eszköztárral is rendelkezik [34]. Az általa nyújtott lehetőségek között szerepel az arc landmarkpontjainak megtalálása és ez alapján egyéb megvalósításokra (pl arcklaszterezés) is alkalmas. Ingyenesen felhasználható *Boost szoftverlicence* ellenére, részletes dokumentációval és példatárral rendelkezik, mely kiterjedt instrukciókat ad a *fordítási folyamattól* egészen a fejlesztési feladatok végrehajtásáig. A csomag

[27] https://docs.aws.amazon.com/rekognition/latest/dg/faces.html
[28] open source
[29] http://dlib.net/ml.html

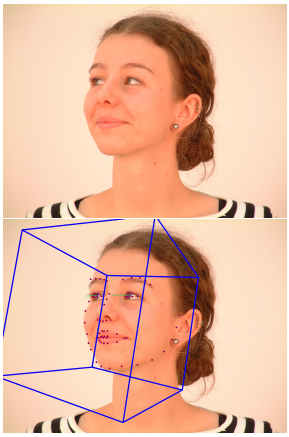
arcfelismerő modulja tartalmaz egy csak kizárólag elülső vetületű arcok felismerését támogató megoldást, mely az úgynevezett *HOG*[30] módszeren alapul, mely (igen csak leegyszerűsítve) a kép eleminek eloszlását használja fel különböző osztályok azonosítására. Újabban neurális hálón alapuló módszerek is részei a csomagnak[31].

4.2.2. OpenCV

A *BSD licenccel* rendelkező nyílt forráskódú, crossplatform csomag[32] főleg a valós idejű képfeldolgozást célozza [35] és ma a legnépszerűbb megoldás a képelemzés területén, különösen mivel natív nVidia CUDA támogatással is rendelkezik. Eredetileg C nyelvre írták az Intel mérnökei, azonban jelenleg C++ és Python nyelven is hozzáférhető, illetve számos nyelvi támogatással kibővítették különféle csomagolóosztályok segítségével. Arcfelismerés terén landmarkpontok detektálására (Facemark API), illetve ezzel kapcsolatos problémák megoldására (pl. arckeresés, landmarkpontok kirajzolása) alkalmas, azonban számos egyéb funkcióval is rendelkezik (pl. sajátarcok és Fischer-arcok használata). A megoldás egy *Haar-Kaszkád osztályozónak*[33] neveztt technológiát használ arcfelismerésre, mely egy gépi tanuláson alapuló módszer, mely kaszkádrétegeket használ [36].

4.2.3. Google Firebase ML Kit

A Google Android és iOS technológiákat célzó Machine Learning SDK-ja átmenet a felhő alapú módszerek és a lokális megoldások között. A dependencia managereken keresztül betölthető kit vonalkódolvasás és általános képfelismerés mellett arcfelismerő szolgáltatásokat is nyújt[34], mely rendelkezik a leggyakrabban előforduló tulajdonságokkal: landmark és arckontúr felismerés, arc dőlésszögének bemérése, vagy határoló négyzet[35], de valósidejű, videó-n történő arckövetésre is képes. Segítségével külön-külön és egyben is lekérhetőek az arcjegyeket határoló pontok, ami nagy segítség képszegmentálással foglalkozó feladatoknál. Az SDK API-ja a példakódokat is tartalmazó dokumentáció alapján könnyen



7. ábra. Eredeti és OpenFace-el elemzett arcképek

[30] histogram of oriented gradients
[31] http://dlib.net/cnn_face_detector.py.html, illetve http://dlib.net/dnn_face_recognition_ex.cpp.html
[32] https://docs.opencv.org/3.4/index.html
[33] haar-cascade classifier
[34] https://firebase.google.com/docs/ml-kit/detect-faces
[35] bounding polygon/box

használható és beépített aszinkron műveletei miatt grafikus felhasználói felületet sem bonyolult fölé integrálni. Használatához ugyan Firebase Projekt létrehozása szükséges a Google Cloud platformon, azonban a műveleteket az eszközön végzi, így működése közben internet kapcsolatra sincs szükség.

4.2.4. Standalone szoftverek - OpenFace

Több nyílt forráskódú szoftver is létezik, mely az arcfelismerés problémájára kínál azonnal használható megoldást azonban képességeikből adódóan kiváló segédeszközök lehetnek több adatmanipulációs eljárás és a nyers képadatok tanítómintává konvertálásában is. Ezek egyike az *OpenFace*[36], melyet Python nyelven, a Torch[37] nevű gépi tanuló könyvtár használatával írták a Google mérnökei [37]. Számos alkalmazási területe között megtalálható - természetesen - a landmarkpontok detektálása és az arckontúrvonalak érzékelése, de a szemek fókuszát és az arc helyzetét is képes felismerni, mindemellett - többek között - alkalmas arcképek automatikus levágására is.

5. Alkalmazható technológiák áttekintése

A fent részletezett feladatot több megközelítésből is elemezni lehet, a megvalósítási terv felvázolása előtt azonban javallott megfontolni, hogy milyen tehnloógiák játszhatnak szerepet az implementálás folyamán. Habár a megvalósítás legalapvetőbb eleme - szoftvertechnológiai oldalról - maga a programozási nyelv, melynek elemeiből a szoftvermegoldás építkezik, tervezési szempontból érdemesebb egy absztrakciós szinttel magasabbról tekinteni a problémára és különböző, konkrét nyelvi megoldások helyett keretrendszerekben és azok összeférhetőségeiben gondolkodni, majd utána azokhoz választani valamilyen implementálási nyelvet. Hasonlóan logikus döntésnek tűnik továbbá, ha egységes megoldás keresése helyett az összetartozó részek csoportosítása kerül górcső alá, épp ezért, az alábbi szekció a megoldásban szerepet kapható technológiákat - teljes mértékben önkényes besorolás alapján - a mesterséges intelligenciára, kliensre és szerverre vonatkozó eszközök szerint mutatja be.

5.1. Mesterséges intelligencia keretrendszerek

Az arcelemzésről szóló szekcióból egyértelműen kiderül, hogy érdemes a neurális hálók témakörét tanulmányozni bármilyen arcfelismeréshez köthető probléma esetén. Noha - mint az bemutatásra került - ehhez több „gyári” megoldás is rendelkezésre

[36] https://cmusatyalab.github.io/openface/
[37] http://torch.ch/

áll, az előző szekcióban definiált feladatra egyik sem nyújt automatikusan működő megoldást, így a következőkben olyan technológiák kerülnek bemutatásra, melyekkel saját szoftvermegoldás implementálható.

5.1.1. A Tensorflow framework-család

A Google nyíltforráskodú mesterséges intelligencia keretrendszere[38] a 2017-es 1.0-s verzió kibocsátása óta mára az iparágat alapvetően meghatározó eszközzé nőtte ki magát: használói között MI kutatóktól kezdve, cégeken keresztül egészen a „hétköznapi" szoftverfejlesztőkig sokféle szakág képviselői megtalálhatók. A framework és az azt kísérő részletes dokumentáció segítségével rendkívül gyorsan állíthatók össze mindössze néhány sor kódból álló, különféle feladatok megoldására alkalmas neurális hálók, melyek nem csak CPU-k, de GPU-k segítségével is futtathatók (nem is beszélve a specifikusan MI problémák megoldására szánt TPU[39] célhardverekről [38]).

A tensorflow működésének alapját egy adatfolyam elvű implementáció adja, melyben az algoritmusok számításait és állapotait *adatfolyam gráfok* reprezentálják, elősegítve ezzel a különböző folyamatok párhuzamosíthatóságát [38]. A rendszer része továbbá, hogy a belső állapotok folyamatosan változtathatóak[40], így a nagy modellek esetén fellépő nagy paraméterszám is kezelhetővé válik [38]. Az eredetileg C++ (és CUDA-C) nyelven írt framework ma sok programozási nyelven elérhető mint amilyen C#, vagy a Java, de leggyakrabban a Python nyelvi integráción keresztül használják.

A mai gyakorlatban tensorflow-val gyakran nem közvetlenül dolgoznak, hanem egy magasabb szinttű *Keras* nevű szoftvercsomagon keresztül, mely - ugyan több alacsonyabb szintű *backendet* is támogat[41] illetve nem is specifikusan tensorflow számára írták - ma már része magának a tensorflow-nak. A projekt eredeti célja egy felhasználóbarátabb API felület biztosítása volt - melyet kétségtelenül el is értek pár sornyi kóddal implementálható hálókonfigurációkkal - de ezen felül több olyan elengedhetetlen funkció érhető el egyszerűen az API-n keresztül mint a különböző, hálózatépítés során használt aktivációs függvények (pl. ReLu, Sigmoid, vagy Softmax[42]).

Kézenfekvőnek tűnhet, hogy a tensorflow-val megalkotott neurális hálók futtatásáért is - képletesen értve - ugyanaz a backend legyen felelős, melyen a háló nagyságával arányos számítási kapacitást igénylő betanítást elvégezték. A szoftver-

[38] https://www.tensorflow.org/
[39] Tensor Processing Unit
[40] mutuable state
[41] https://keras.io/backend/
[42] a teljes lista az írás keletkezésének idején megtekinthető: https://www.tensorflow.org/api_docs/python/tf/keras/activations

csomag népszerűségét mutatja azonban, hogy több megoldással is kiegészült az évek során: ilyen például a *TensorFlow.js*[43], mely a böngészőket és egyéb, JavaScript futtatását támogató futtatási környezeteket célozza, vagy a *TensorFlow Lite*[44], mellyel mobil környezetben hajthatóak meg a már előre betanított hálók, de a különféle nyelvekhez is írtak API-kat (pl. Tensorflow Java) Mindkét módszer használatához modell-konverzió szükséges, de előbbi esetén lehetőség van további tanításra is.

5.1.2. Torch

Noha az ugyan csak C++/Cuda-C alapú nyílt forráskódú framework[45] elsősorban C és Lua (LuaJIT) interfészre íródott, Python nyelven is elérhető a *PyTorch*[46] keretrendszeren keresztül. A keretrendszer kiterjedt GPU támogatással rendelkezik és - a hivatalos weboldal szerint - elsősorban a tudományos közösséget célozza, ennek ellenére számos megacég eszköztárában megtalálható (pl Facebook, Twitter, vagy maga a Google). A meglévő algoritmusok módosításának és újak implementálásának megkönnyítés végett kialakításában a modularitást helyezték előtérbe [39], így működési elve is négy alapszerkezet köré csoportosul: a *DataSet* nevű osztály kezeli az adatokat, a *Machine* feladata a különféle kimenetek előállítása, míg a *Trainer* az aktuális kritériumoknak és adatoknak megfelelő, optimális paraméterkiosztást választja ki, a *Measurer* osztály pedig a felhasználó számára értelmezhető, mérések során keletkező metrikák (pl. klasszifikációs hiba arány) kiíratásáért felelős [39]. A Torch natívan része az Ubuntu Linux disztribúciónak, így funkciói főleg erre az operációs rendszerre lettek optimalizálva, de képes megbirkózni több platformmal is - ugyanezek felhasználóbarát jellege viszonylag nagy skálán mozog: míg a Macintosh OS X verziója stabil, addig - hivatalos támogatás hiányában - a Windows támogatáshoz mellékelt tájékoztató egyenesen egy Ubuntu virtuális gép letöltését javasolja[47].

5.1.3. Microsoft Cognitive Toolkit

A Microsoft nyílt forráskódú, egyszerűen csak CNTK-nak rövidített megoldása[48] a saját úgynevezett *Brain Script* interfésze mellett rendelkezik Python, C# és C++ támogatással, mindemellett lehetőséget ad több CPU mag és GPU környezetben történő használatra is [40]. A Tensorflow-hoz hasonlóan adatfolyam gráf alapú kivitelezést tesz lehetővé annak minden előnyével (pl. késleltetett végrehajtás[49]), illetve

[43] https://www.tensorflow.org/js
[44] https://www.tensorflow.org/lite
[45] http://torch.ch/
[46] https://pytorch.org/
[47] https://github.com/torch/torch7/wiki/Windows
[48] https://docs.microsoft.com/en-us/cognitive-toolkit/
[49] deferred execution

könnyen érthető architektúrájának, valamint kiterjedt dokumentációjának köszönhetően gyorsan és egyszerűen lehet különféle hálókat építeni az olyan, gyakran használt funkciókat tömörítő osztályok segítségével, mint például a *Layers Library*, melyben népszerűbb ANN rétegminták kaptak helyet (pl. Konvolúciós, MaxPooling, vagy LSTM)

5.1.4. Egyéb keretrendszerek

A fent megemlített népszerűbb keretrendszereken kívül természetesen nem kevés MI framework áll még rendelkezésre. Vannak közöttük feltörekvő nyílt forráskódú rendszerek, mint például a Microsoft direkt .Net fejlesztőkre koncentráló megoldása az *ML.NET*[50], mely natív C# és F# támogatással rendelkezik, vagy a magát „tensor-mentes gépi tanulási keretrendszerként" hirdető[51] *Julia* nyelvre írt *Flux*, de a két magánkézben lévő quasi ipari sztenderd mérnököknek szánt szoftvercsomag - a Wolfram Mathematica[52] és a Matlab[53] - is rendelkezik gépi tanuláshoz használható csomagokkal.

5.2. Kliens-szerver architektúra technológiái

Ha az előbbi szakaszba sorolt „technológiacsaládra" elmondható, hogy népes tábort alkotnak, akkor a jelen fejezet tárgyát képző webes technológiák igazi nemzettség: az évek során rengeteg technológia megjelent - és el is tűnt - melyek alkalmasak egy kliensvégpont és egy központi szerver közötti kapcsolat kialakítására. Mivel ebből kifolyólag mind a kliens, mind pedig a szerver kiválasztása a választott MI keretrendszer függvénye lesz, így jelen fejezet a különféle technológiák szimpla felsorolása helyett a *kliens-szerver architektúra* elemeiből merítve, általános érvényű kvalitásokat fogalmaz meg röviden mindkét képviselővel szemben és ezen elvek mentén nyújt betekintést az erre alkalmas technológiai megoldások közé.

5.2.1. Szerver oldali technológiák

Nevéből adódóan is, a szerver - vagy magyarosított nevén a kiszolgáló - feladata az ügyféloldali végpontok ellátása adatokkal illetve minden olyan (általában számításigényesebb) szolgáltatással, mely azok működéséhez szükséges, így eme komponensek felelősek többek között: adatbáziskapcsolatok kiépítésért és fenntartásáért, üzleti szolgáltatások kiajánlásáért (pl. számlázás, foglalások, keresőmotorok), illetve az ezekhez történő hozzáférés irányításáért és koreografálásáért olyan mértékben,

[50] https://dotnet.microsoft.com/apps/machinelearning-ai/ml-dotnet
[51] https://github.com/FluxML/Flux.jl
[52] https://www.wolfram.com/featureset/machine-learning/
[53] https://www.mathworks.com/products/deep-learning.html

hogy a különféle végpontokon helyet foglaló felhasználóknak - a szolgáltatás minőségét tekintve - ne legyen egymásról tudomása. Épp ezért, a szerver gyakran a megszólított fél szerepét tölti be a kliensek hívására várva, bár természetesen kezdeményező fél is lehet a kommunikációs folyamban [41]. A szerverek általában egy vagy több, az üzleti igényeknek megfelelő, speciális igény kielégítését teszik lehetővé, úgynevezett *webszolgáltatásokra*[54] specializálódnak. Ezen szolgáltatásokért lehetnek egymagukban, vagy más kiszolgálókkal együtt felelősek, utóbbi - ma már igen gyakori - esetben képesnek kell lenniük a kliensektől kapott feladatok felosztására és a megfelelő felek között történő kiosztására is [41].

A fentebbi általános érvényű funckiók mellett érdemes megemlíteni egy olyan aspektust is, mely - egyre népszerűbb mivolta ellenére is - inkább egy ajánlás a webes adattovábbítás lebonyolítására, mintsem szabály, ez pedig az úgynevezett *Representational State Transfer*, vagy *REST*. Az ilyen elvű kommunikációban a felek között továbbítandó, előre feldolgozott információn (*representational state-en*) van a hangsúly, melyet a felek megosztanak egymással. A hívásokat előre definiált, mindkét fél által érthető, speciális HTTP protokoll alapú módszerekkel intézik: a kliens és a szerver számára egyaránt hozzáférhető erőforrásokat különböző linkeken (*URI*) érik el, az adatmenipulálás (*CRUD*) eszközei pedig feladatonként dedikált URI végpontokon található feldolgozó egységek (metódusok) (*HTTP Verbs*, pl. *GET* adatok lekérésére, vagy *POST* adatok létrehozására) [42]. Mivel a kommunikáció az URI végpontokra küldött információk gyakorlatában valósul meg, a kliensnek nincs szüksége bonyolult implementációra, elég ha ismeri a szerver oldali erőforrás URI címét, az üzenetküldés során használt közös objektumot (*Data Transfer Object* vagy DTO) és a használni kívánt szolgáltatás azonosítóját, így nem csak rugalmasabb, kevésbé összefüggő kapcsolat alakulhat a kliensek és a szerver között, melynek velejárója, hogy szerver oldali változtatások lekövetése sem feltétlenül szükséges kliens oldalon [42]. Hogy megfeleljen a jelen kor követelményeinek tehát, egy szervernek ma képesnek kell lennie a REST-alapú kommunikáció lebonyolítására is.

A ma használatos technológiák közül igen sok keretrendszer megállja a helyét a felvázolt rendszerek kiépítésében. Manapság, az ehhez hasonló problémákra népszerű megoldás például a Microsoft (elsősorban) C# nyelvű *ASP .Net Core*[55] keretrendszere, mely - a korábbi, .Net Standard frameworkre írt *ASP* keretrendszert felülmúlva - beépített *dependencia injektáló* komponenssel és REST controllerrel rendelkezik, de a moduláris, *Bean-nek* keresztelt konfigurációjáról ismert *Spring*[56], illetve *Spring Boot*[57] is hasonló funkcionalitást kínál Java (és hamarosan Kotlin) nyelven. Mindkét népszerű framework alkalmas az *aszinkron* és *párhuzamos* műveletvégrehajtásra,

[54] webservice
[55] https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-3.1
[56] https://spring.io/
[57] https://spring.io/projects/spring-boot

melyekkel az adott problémára szabható a feladatok szétválogatása, valamint *XML*, illetve *JSON* alapú webes kommunikációt megvalósító megoldásokkal is bővíthetőek különböző csomagok segítségével. Egy harmadik, egyre jobban elterjedő futtatási környezet a *NodeJS*[58], mely *JavaScript* (és az erre „forduló" egyéb nyelvek, mint pl. a *TypeScript*) alapú programok böngészőn kívüli futtatását teszi lehetővé. Ugyan a NodeJS esemény alapú architektúrát vesz alapul párhuzamos kivitelezés helyett, de képes több processzormagot is kihasználni. Tervezésénél alapépítő elemként szerepelt a webet meghatározó aszinkron funkcionalitás, így ez a technológia is kiválóan alkalmas a feladatra. Eme három elterjedt framework mellett érdemes megelíteni a Python nyelvű *Django* és *Flask* keretrendszereket is, illetve feltehetőleg a PHP *Symfony* és *Laravel* framework-jei is megbirkóznának a feladattal.

5.2.2. Kliens oldali technológiák

A kliensek legfőbb feladata általában a különféle funkciók felhasználók elé tárása és megjelenítése, fejlesztésükben egyarán fontos szerepet játszik a felhasználói felület[59] kialakítása, a funkciók lefejlesztése és a különféle, felhasználói élményt befolyásoló[60] döntések megválaszolása. Mivel a felhasználói kérések validálása és továbbítása is ezen réteg feladata, így általában a kliens a kommunikációs folyamatok kezdeményezője [41]. A mai modern kliensek döntő hányada grafikus, illetve tetszőleges számú szerver megszólítására képes. Ezek mellett természetesen a kliensek számára is elengedhetetlen, hogy képesek legyenek webes kapcsolatot kialakítani, illetve - amennyiben a szerver ilyen elven működik - REST hívásokat kezdeményezni és válaszaikat bevárni. Ugyan a kliensek általában nem oldanak meg számításigényes feladatokat, feladatkiírásban szereplő előfeldolgozás megvalósításához azonban egy ilyen feladatok végrehajtására képes, úgynevezett *vastag kliens* implementálása szükséges.

Hagyományos értelemben a klienseket három nagy csoportra osztják: asztali[61] (pl. Windows, Macintosh, vagy valamilyen Linux disztribúció), webes (melyeket valamilyen webböngésző hajt meg), illetve mobil (Android vagy iOS) kliensekre, ám ma már egyre növekvő igény mutatkozik olyan megoldások iránt, melyek esetében elmosódnak ezek a különbségek. Az egyes csoportokon belül is nem ritkán találni olyan technológiákat, melyek az adott csoporton belül több *platformra* is elérhetőek[62].

A kliensoldali technológiák témaköre talán az egyetlen, amit érdemesebb a különféle programozási nyelvek felől megközelíteni, hiszen a legtöbb népszerű nyelv

[58] https://nodejs.org/en/about/
[59] user interface vagy UI, illetve grafikus megjelenítő esetén GUI
[60] user experience, UX
[61] desktop
[62] cross-platform, multi-platform, de illetik még a platform-agnostic kifejezéssel is

rendelkezik mindhárom csoport, és azon belül is több platform igényeit kielégítő keretrendszerrel és szoftvercsomaggal. Jóllehet a C# nyelvű *WPF* asztali framework kizárólag Windows operációs rendszeren futtatható, a Microsoft tulajdonába csak később kerülő *Xamarin* alapú technológiák segítségével viszont írható szoftver Androidra és iOS-re (Xamarin Forms), illetve - noha kevésbé elterjedt - macOS-re is (Xamarin.Mac[63]), a *Razor*, illetve újabban a *Blazor* tecnológiák pedig a különböző webböngészőket célozzák. Ezen a nyelven jelenleg nincs széles körben is elfogadott Linux desktop támogatás, habár .Net Core webapplikációkkal némileg betölthető az űr. Az egykor az Android platform hivatalos nyelvének számító Java is sokszínű framework családdal rendelkezik: a Java-t letaszító (de szintén a Java runtime-ot[64] használó) Kotlin mellett természetesen továbbra is lehetséges Androidra fejleszteni, de a *Multi-OS Engine*[65] névre keresztelt framework-el, már iOS-re is lehetséges fejleszteni. Desktop odalon a Java SE (Linux), illetve a Java FX (Windows, vagy Macintosh) érhető el, míg webes oldalról jó választás lehet a korábbi szakaszban is szereplő Spring. Eme technológiák mellett azonban igen rohamosan növekszik a különféle Javascript alapú frameworkök népszerűsége is. Habár még a népszerűbb JS szoftvercsomagok felsorolása is kimerítene még egy mesterszakos szakdoglozat által szabott határokat is, desktopon a NodeJS keretrendszeren futó *Electron* terjedt el hála mindhárom platformon átfogó támogatottságának, míg mobilon a böngésző és natív mobil alkalmazás közötti átmenetet képező hibrid *Ionic*, illetve a „quasi-natív" *React Native* framework-ökhöz nyúlnak legszívesebben a JS fejlesztők. A Javascript ereje természetesen a webes techológiákban mutatkozik meg, így számos olyan népszerű keretrendszer JS alapú mint például a *React*, az *Angular 2+* vagy a *Vue*. Természetesen ezen technológiák mellett egyéb nyelvek keretrendszerei is teljes joggal kaphatnának itt helyet, azonban ezek felsorolása már kimeríteni a jelen írás korlátait.

6. A megoldandó feladat

Jelen munka célja egy olyan arcfelismerő rendszer implementálása, melyben az arcok elemzésének a feladata megoszlik a kliens és a szerver komponensek között. Mielőtt azonban a részletes feladatleírást be lehetne mutatni, szükséges megállapítani azokat a kritériumokat, melyek egy ilyen rendszert jellemezhetnek, így a részletes specifikáció előtt ez kerül bemutatásra.

[63] https://docs.microsoft.com/en-us/xamarin/mac/get-started/hello-mac
[64] Java Virtual Machine, vagy JVM
[65] https://multi-os-engine.org/

6.1. Egy rétegzett rendszer elvi alapjai

Az irodalomban leggyakrabban olyan megoldásokkal lehet találkozni, melyek a meglévő képfelismerő módszerek továbbfejlesztésére törekednek, nem pedig különféle architektúrális szemléletek felvonultatására [43], [44]. Ugyan akadnak „gyakorlatorientáltabb" kivételek [9][5], de nem áll rendelkezésre olyan forrás, mely alapján egy osztott gépi tanulással rendelkező megoldást értékelni lehetne, pedig jelen feladat elemzéséhez elengedhetetlen egy ilyen. Épp ezért a jelen szekcióban ezt kísérlem meg felvázolni szoftvertechnolológiából már ismert elemek segítségével.

6.1.1. Az osztott háló architektúra, mint szoftver tervezési elv

Ian Sommerwille *Software Engineering* című könyve ([45]) a rétegzés használatát a komponensek közötti függetlenséghez köti kiemelve a lokális természetű változtatások fontosságát. Az általa felvázolt rétegzett rendszerben minden komponens egymástól függetlenül alakítható, azonban függőségi viszonyban állnak egymással: minden egyes réteg a közvetlenül alatta lévővel kommunikál. Ez a lokális változtatásokkal operáló módszer lehetővé teszi az inkrementális jellegű fejlesztési folyamatokat, így az egyes rétegek által biztosított szolgáltatások egy része az ügyfelek rendelkezésére bocsátható. A rétegzés egy másik nagy előnye Sommerville szerint, hogy az egyes szinteken megvalósításra kerülő implementáció tetszőleges mértékben cserélhető, mely nem csak a különböző változásokra történő reagálást könnyíti meg, de egyben több, eltérő helyzetben alkalmazható megoldás kifejlesztését is megengedi. Érdemes hozzátenni, hogy ezzel együtt a szolgáltatások köre is bármikor bővíthető.

Természetesen az éremnek nem csak egy oldala van, így a mű figyelmeztet a rétegzett architektúrák néhány, a gyakorlatban gyakran előforduló hátrányára is: sokszor nehéz jól elszeparált rétegeket alkotni, mivel magasabb rétegeknek szüksége lehet több szinttel alacsonyabb rétegekkel történő kommunikációra és ebben az esetben az információt több rétegen keresztül kellene átvezetni, ami pedig performanica problémákhoz vezethet. Az itt leírtakat támpontként felhasználva már lehetséges kritériumokat megállapítani, melyek, jóllehet, nem kimerítő jellegűek, de elegendőek lehetnek a jelen feladat megoldásához.

Az idézett szerző a rétegzett komponensek esetében az elkülönülő változásokat emeli ki így érdemes jelen esetben is eköré építeni az összehasonlítást. Így tehát egy ilyen architektúrában a fentiekhez hasonlóan az egyes komponenseket - azaz különálló gépi tanuló elemeket - egymástól függetlenül is lehetségesnek kell lenni fejleszteni valamilyen mértékben. Ebből természetszerűleg következik az esetleges változások lokális mivolta is. Ekkor azonban ahhoz, hogy egységként tudjanak működni, szükségszerű, hogy ezek a gépi megoldások egymással függőségi viszonyban álljanak. Ha csak nem valamilyen ekvivalens átalakítást hajtanak végre egyes ele-

mek a bemeneti adatokon, feltételezhető, hogy körkörös hivatkozás az elemek között nem lehetséges, azaz a függőségek között sorrendi hierarchia áll fent: az információ egyiktől a másikig halad egy előre eltervezett irányban, és visszafelé áramlás nem lehetséges. Ez természetesen az egyes elemek *között* értendő, és nem keverendő össze a neurális hálókban előforduló *back-propagation* jelenségével (már csak azért sem, mert gépi tanuló módszer használata nem kizárólag a neurális hálók témakörére szorítkozik). Ugyanez lényeges különbség az előbbiekben említett szoftverekhez képest, ahol ez nem csak hogy lehetséges, de egyenesen megszokott (mint pl. a szerző által, a 157. oldalon említett MVC minta esetében is), de mégis magával vonja, hogy a komponensek csak a közvetlenül alattuk-fölöttük lévő elemekkel kommunikálnak, másszóval, a „felsőbb" rétegek által előállított kimenetek bemenetként szolgálnak az „alsóbb" rétegek számára.

A fentebb taglaltaknak két negatív következménye lehet: egyrészt, ha a kimenetek bemenetté alakítása a komponensek között különösebb előfeldolgozást igényel (ami megfeleltethető a rétegek közötti DTO-k egymásbakonvertálásának), akkor annak performanciacsökkentő következményei lesznek a rendszer egészét tekintve, másrészt az alá-fölé rendeltségi viszony megakadályozhatja, hogy egyes komponensek a felettük lévők nélkül is működhessenek. Így tehát az inkrementális fejlesztés csak abban az esetben igaz, ha az ügyféligény a folyamat bemenetét elsőként befogadó elemet, vagy azzal együtt valamely alsóbb szinten lévő implementációt priorizálja előbbre.

A korábbi fejezetekben taglaltak alapján kijelenthető még egy hasonlóság a nem gépi elvű szoftverekhez képest: mivel rengeteg megoldás áll rendelkezésre egyes problémák gépi tanuló módszerekkel történő megoldására, így ha sikerül egy elkülönülő megoldást létrehozni amely adott bemenetekkel és kimenetekkel operál, akkor ezek között az implementáció egy másik, azzal ekvivalens implementációra cserélhető. Példának okáért, az arcpontok azonosítása egyaránt megoldható konvolúciós neurális hálóval [46] és ún. *Viola-Jones-féle kaszkádos osztályzóval* ([47]) is, ahol mindkét esetben egy arcképi bemenetre arcpontok koordinátái a produkált kimenet. A fentebbi ismeretek fényében már meghatározható a részletes feladatleírás.

6.2. Részletes specifikáció

A felsoroltak alapján tehát egy kliens-szerver alapú rendszer megtervezése és implementálása a feladat, melyek egymással hálózaton keresztül kommunikálnak és hierarchikus kapcsolatban állnak. A két komponens között JSON üzenetek formájában, a REST szabályainak megfelelően kell lebonyolítani a kapcsolatot. Ezeken a közös elvárásokon kívül a komponenseknek külön-külön is meg kell felelniük bizonyos elvárásoknak.

6.2.1. A szerverrel szemben támasztott követelmények

A szerver központi egysége egy olyan, betanított neurális hálót tartalmazó rész kell legyen, mely a megkapott adatokból képes eldönteni, hogy azok melyik személyhez tartoznak az előre meghatározott személyek közül. Ezen felül egy olyan egységet is implementálni kell, mely képes JSON üzeneteket küldeni és fogadni hálózaton keresztül, valamint a hasonlóság megállapításához szükséges formátumba alakítja a megkapott adatokat, illetve vissza az ebből megkapott választ.

6.2.2. A klienssel szemben támasztott követelmények

A kliens egy olyan, mobileszközre telepíthető alkalmazás kell legyen, mely képes az eszközbe épített kamera segítségével képet készíteni, és megállapítani, hogy azon található-e arc. Amennyiben igen, úgy képesnek kell lennie abból kinyerni a felismeréshez szükséges adatokat elvégezve akár bizonyos képátalakító műveleteket is a feladat végrehajtásának értelmében. Mivel az adatok kinyerésének a szervertől függetlennek kell lennie, így ennek a szolgáltatásnak internetkapcsolat nélkül is elérhetőnek kell lennie. A szerverhez hasonlóan szükséges implementálni egy hálózati komponenst, mely képes a szerverrel kapcsolatot kezdeményezni és fent tartani, illetve felkészíti az adatokat JSON formátumba történő szállításra.

6.2.3. Egyéb implementálandó megoldások

Mivel a neurális hálónak a rendszer működése idején már betanított állapotban kell lennie, így szükség lehet olyan segédszoftverek/programok implementálására is, melyek elvégzik az ehhez szükséges műveleteket (pl. képek méretének egységesítése, képek alapján megfelelő méretű tanítóminta előállítása, helyességet ellenőrző módszerek). Ezek a megoldások csak közvetve részei a feladatnak, így erre irányuló elemzések nem szerepelnek a korábbi szekciókban sem.

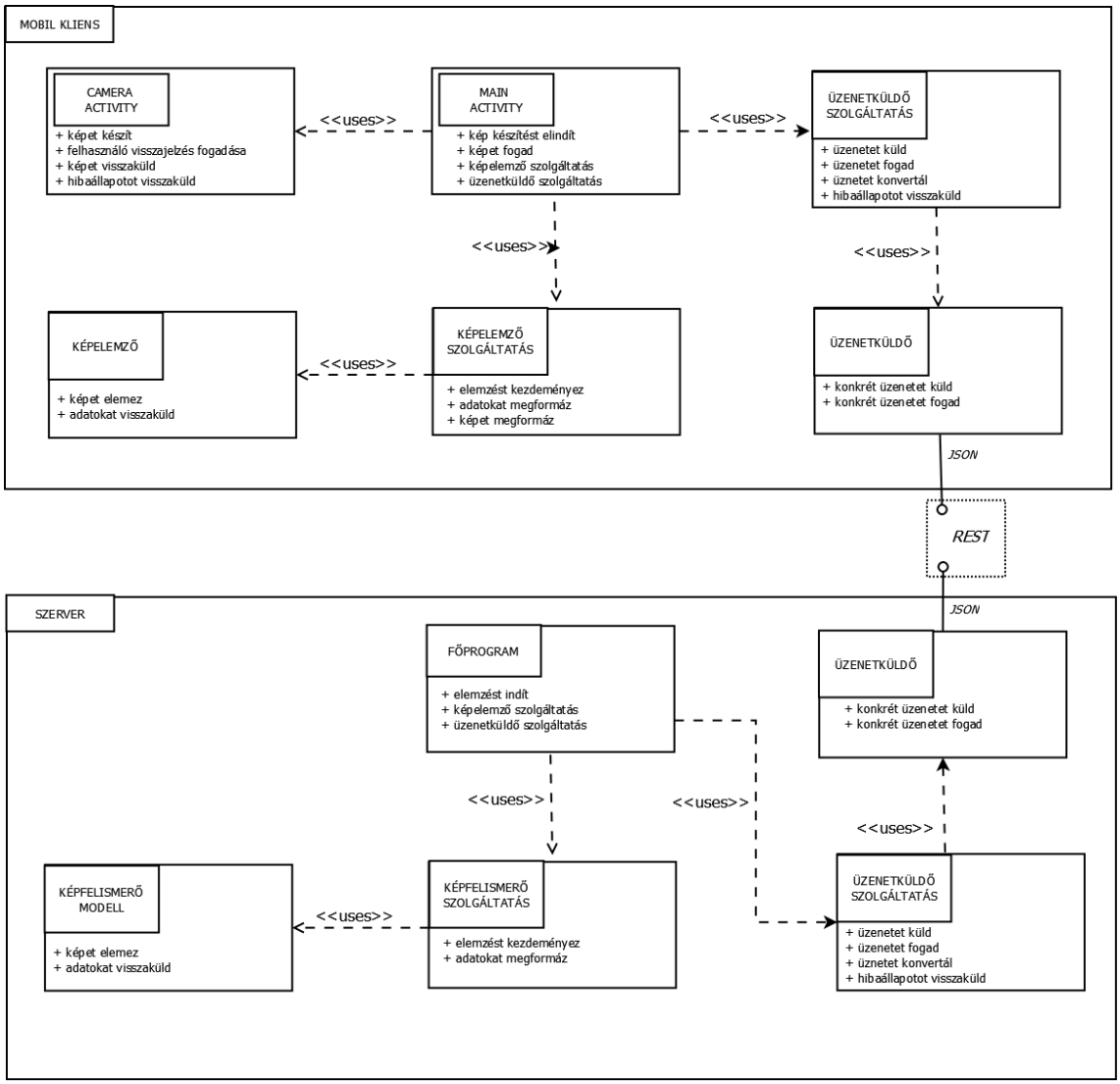
6.3. Architektúrális vázlat

6.3.1. Rendszerterv

A fent felsorolt követelményekből könnyen megállapítható az implementálandó szoftver vázlata. A két fő komponens további kompenensekre bontható, melyeknek speciális szerepük van, így implementálásuk is külön módszerekkel történik. Kivételt képeznek ezalól az üzenetküldésrét és a képfeldolgozásért felelős alkomponensek, melyet mind a kliens, mind a szerver esetén hasonló kialakítás jellemez: eme osztályokat érdemes egy csomagoló osztállyal ellátni, melyek alatt cserélhetővé válik a tényleges implementáció (homlokzat tervezési minta[66]), mivel többféle megoldás,

[66] facade pattern

esetenként többféle külső könyvtár is megoldja a problémát, de ezek tényleges implementációja a szoftverrendszer szempontjából nem lényeges, annak üzemelnie kell bármilyen működő megoldás alkalmazása mellett is. Ezeket a tényezőket leszámítva a rendszer maga visonylag egyszerű, vázlatát a 8. ábra mutatja be részletesebben.

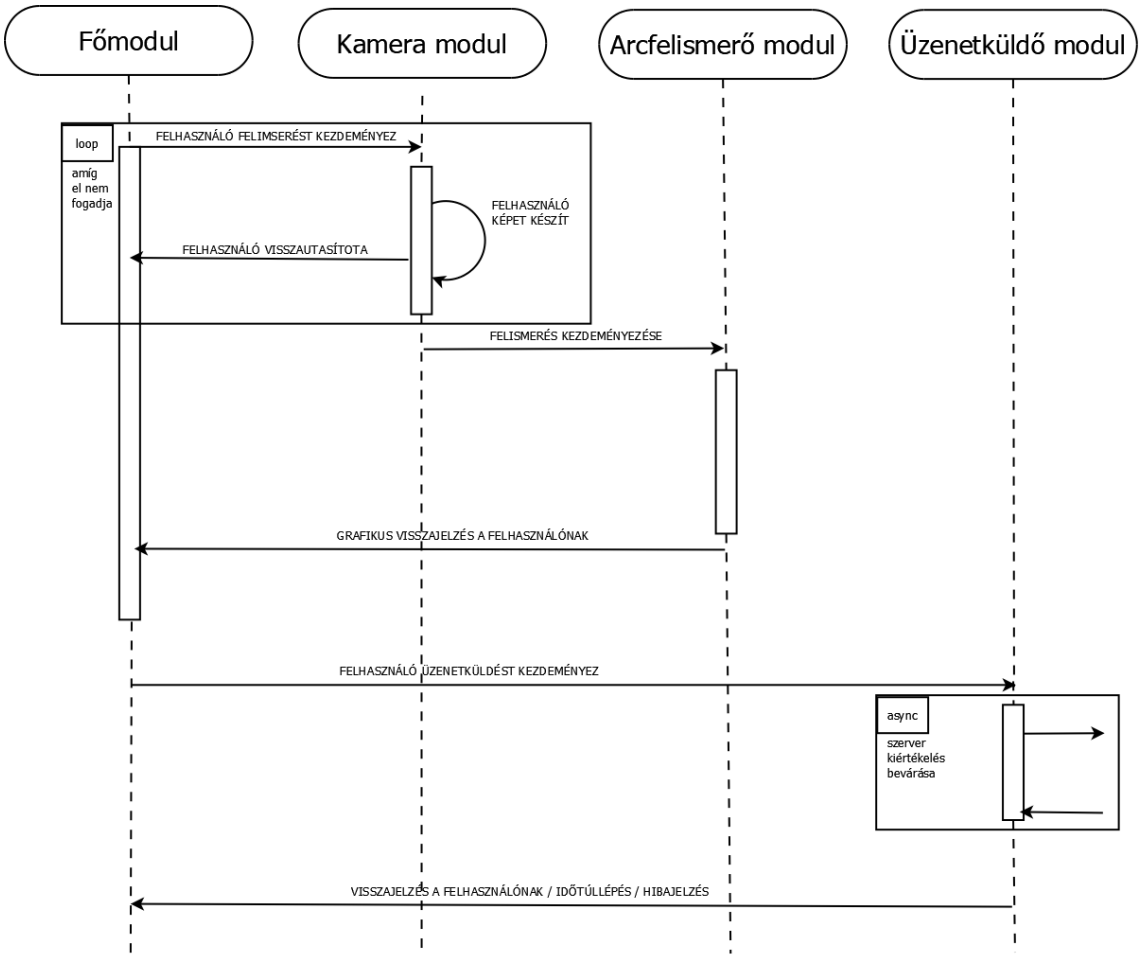


8. ábra. A rendszer vázlata

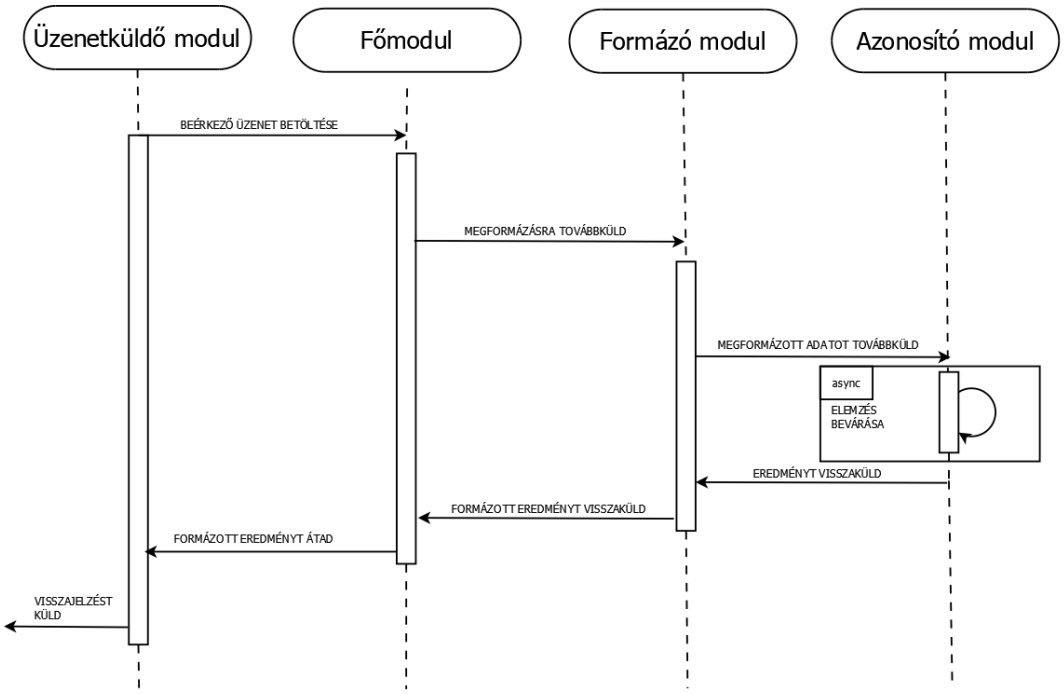
Emellett a szoftverrendszerben jól definiálható kommunikációs folyamatok is leírhatóak mind az adatok arcból történő kinyerésének folyamatát, mind pedig az adatokat felasználó arcfelismerés folyamatát illetően, melyeket a 10. és a 9. mutatnak be.

6.3.2. Használati esetek

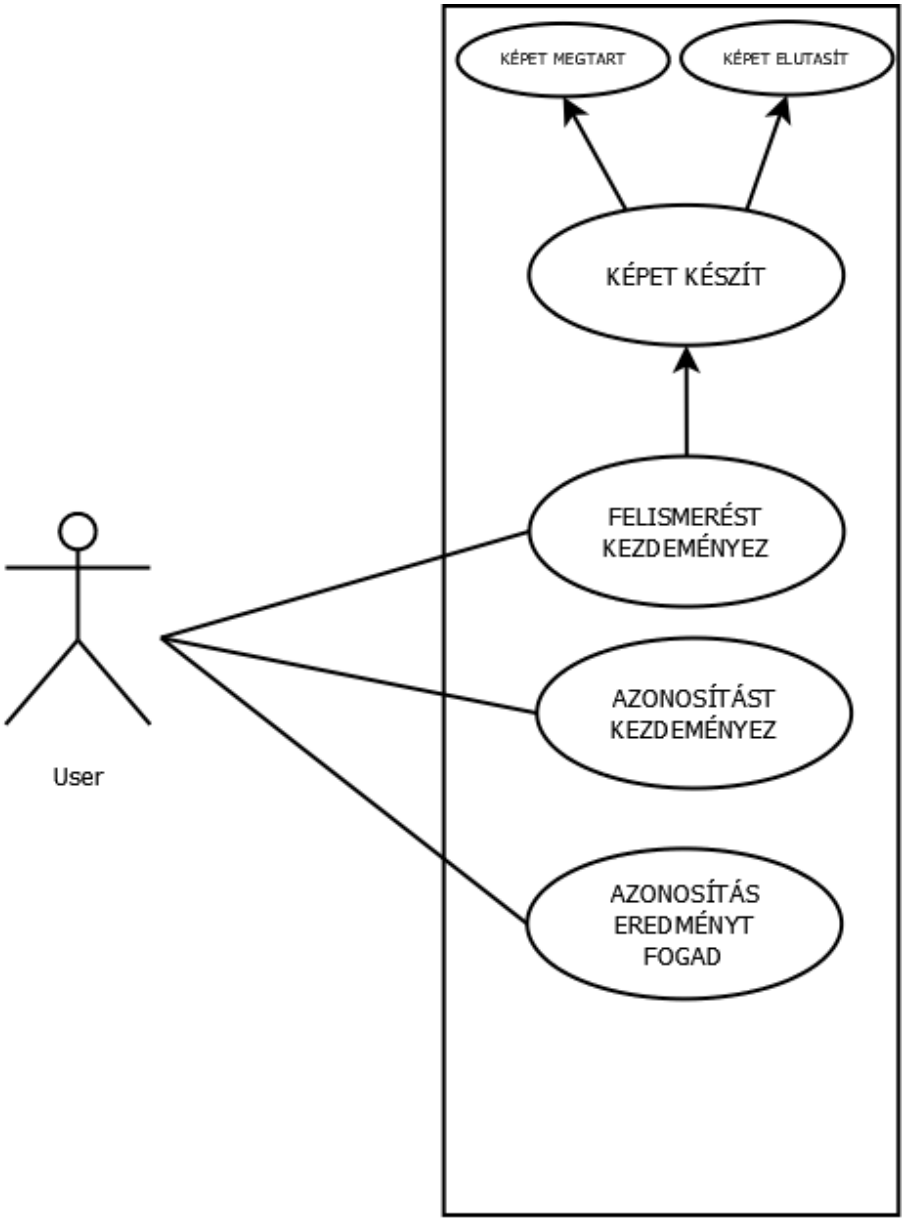
A felhasználó feladatköreit is az egyszerűség jellemzi, két fő feladata a kamera segítségével képet észíteni, illetve a kamera által visszaküldött eredményt elemzésre továbbküldeni a szervernek. Ezeket a tevékenységi köröket természetesen néhány köztes folyamat is kíséri. A különböző feladatköröket a 11 ábra mutatja be, míg az egyszerű választási útvonalat a 12.



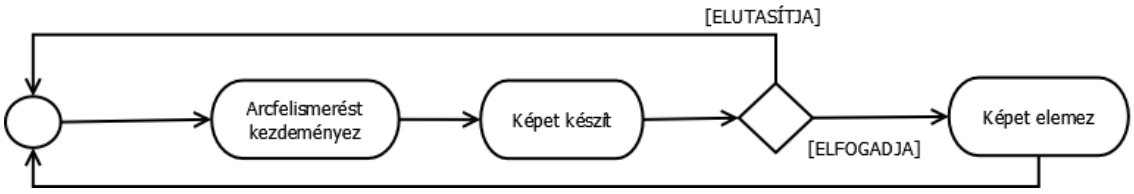
9. ábra. A képfelismerés folyamata



10. ábra. A képelemzés folyamata



11. ábra. A felhasználó feladatkörei



12. ábra. A felhasználó döntési útvonalai a kliensen

7. Az implementált szoftverrendszer bemutatása

A rendszer leírását a szoftver működésének bemutatása nyitja, mely részletesen taglalja majd a megvalósított rendszer funkcióit és hogy ezek hogyan fedik le a feladatkiírásban megfogalmazott követelményeket. Itt kerülnek megemlítésre a felhasználó véglegesített szerepkörei és a szoftver által nyújtott szolgáltatások. Ezt követően a rendszer alapköveit alkotó komponensekről lesz szó és az előbbi folyamatokban betöltött szerepükről.

7.1. Funkcionalitás

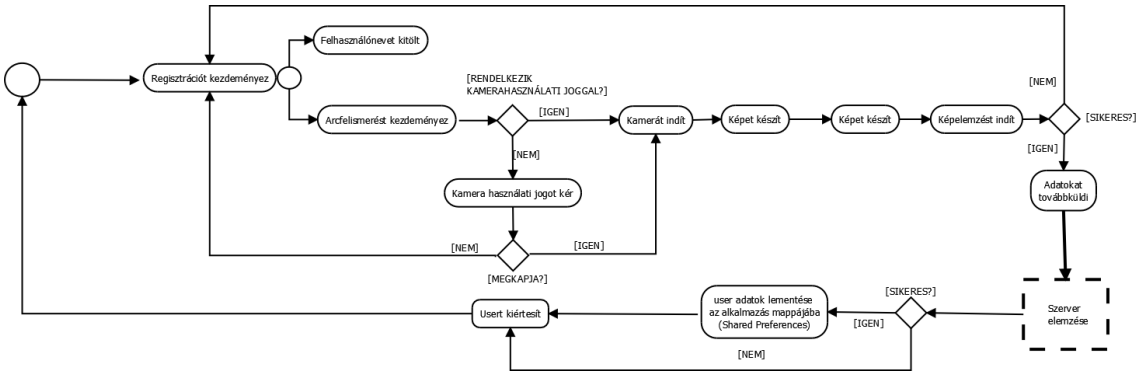
A szoftverrendszer nem rendelkezik szerteágazó funkciólistával, az implementáció során törekedtem arra, hogy kizárólag a feladatmegoldás során fontos elemek kerüljenek lefejelesztésre, így a szoftver által nyújtott szolgáltatások pontosan lefedik a feladatkiírásban előterjesztett követelményeket. Az életszerűbb működés érdekében azonban kerültem minden olyan megoldást, mely a kód minőségét rontaná (pl. bedrótozott felhasználók, vagy konfiguráció), épp ezért a követelményeken kívül az alkamazás rendelkezik egy regisztrációs funkcióval is. Az alábbiakban tehát ezen funkciók kerülnek bemutatásra.

7.1.1. Regisztrációs folyamat

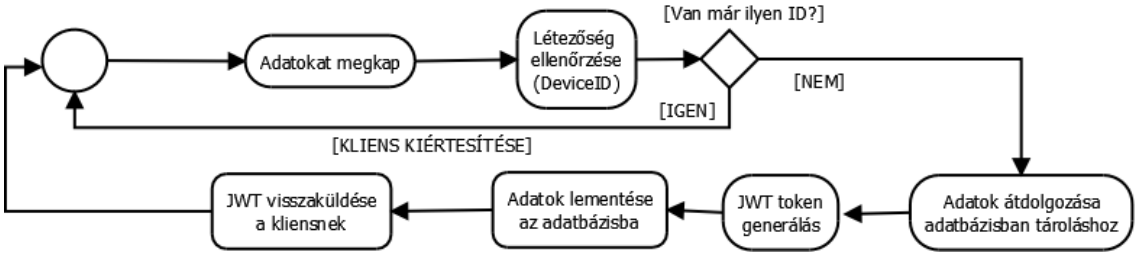
Ez a folyamat felelős a felhasználó és eszközének lementéséért a szerver által használt adatbázisba. A folyamatot a felhasználó kezdeményezi a mobil kliensről az erre szolgáló dialógusablak megnyitásával, majd miután kitölti a felhasználó név beviteli mezőjét, készítenie kell saját magáról egy képet az alkalmazáson belül, mely azonnal elemzésre is kerül. Ha az adatok megfelelően ki lettek töltve, dönthet a folyamat megszakítása mellett, vagy folytathatja azt elküldve a szervernek az adatokat.

A szerver ellenőrzi, hogy nincs-e még ilyen eszköz regisztrálva az adatbázisban és ha nincs, akkor lementi az adatokat. Az alkalmazás biztonságát növelendő, a szerver és a kliens is biztosít egy-egy azonosítót (az előbbi egy globálisan egyedi azonosítót [67], míg utóbbi egy JSON Web Token-t, azaz JWT-t), melyek az adatbázisban eltárolásra kerülnek. A regisztráció eredményéről a felhasználót kiértesíti az alkalmazás, sikeres regisztráció esetén pedig eltárolja a felhasználói nevet és a JWT-t az android rendszer speciális, csak az adott alkalmazás számára fenntartott *Shared Preferences*-en belül található mappában. A folyamatot a 13a és 13b ábrák mutatják be.

[67] globally/universally unique identifier, androidban UUID



(a) kliens



(b) szerver

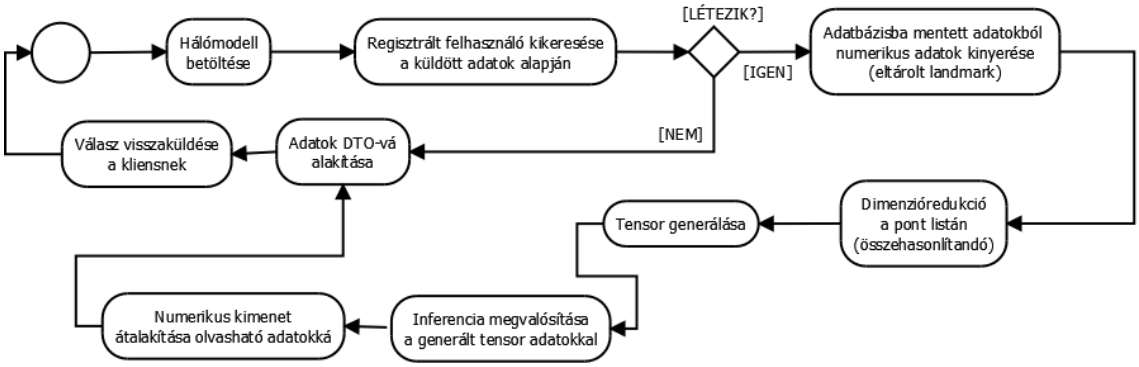
13. ábra. A regisztrációs folyamat

7.1.2. Autentikációs folyamat

A követelményeknek megfelelően az autentikáció az elkészített szoftver fő funkciója, mely azonosít egy előre regisztrált felhasználót. Sikeres regisztráció után a felhasználó kezdeményezheti a folyamatot a kliensen. Először készít egy képet a telefon kamerájával az alkalmazáson belülről, melyen jóváhagyás után azonnal megindul az arcpontok kielemzése. A szoftver egy határoló négyszöget határoz meg az arc körül, majd ennek mentén megvágja azt, hogy a lehető legkisebb felületen kezdődjön meg az elemzés optimalizálva ezzel a folyamat sebességét. Az elemzés aszinkron módon lett implementálva és az elemzés közben töltő képrnyő is megjelenik, azonban megszakítására ilyenkor csak hardware-es megoldás adódik (vissza, vagy home gomb). Ezután a képernyőn megjelenik egy értesítés a folyamat sikerességéről, illetve kedvező végkifejlet esetén a kép, melyre kirajzolódnak a landmark pontok és az arcot körülvevő határoló négyszög. Az autentikációhoz vezető kliensoldali lépéseket a 12 részletezi, így ennek bemutatása nem kerül ismétlésre. Az előre meghatározott személyek összehasonlítására képes folyamatot a 14a írja le, míg a felhasználó arcképét a kapott képpel összehasonlító módszer lépéseit a 14b részletezi.

A szerveren két lehetőség is van autentikációra, melyet 2 külön neurális háló valósít meg: az egyik 3 előre eldöntött személy közül tudja azonosítani, hogy melyik található a képen, a másik pedig a regisztrált felhasználó és a kapott kép között tudja eldönteni, hogy ugyanazt a személyt ábrázolja-e.

(a) előre meghatározott személyek összehasonlítása



(b) kapott és eltárolt kép összehasonlítása

14. ábra. Az autentikáció folyamatának két szerver oldali implementációja

A szerveren a két implementáció között egy egyszerű függvényhívással lehet váltani, erre azonban a kliens nem képes. Voltaképp pont ez a jelen dolgozat fő témája is: a kliens számára teljesen láthatatlan, hogy melyik implementáció kerül ténylegesen meghívásra, mert mindkét szerver oldali megoldás ugyanazt a bemenetet és kimenetet használja megvalosítva ezzel egy leválasztható, több rétegből[68] álló hálóarchitektúrát.

7.2. Felhasznált technológiák és indoklásuk

A gépi tanulásért felelős technológiák közül a Python nyelvű Tensorflow keretendszer rendelkezik Javascript alapú webes környezetbe könnyen átültethető gépi tanuló modelleket kezelő bővítménnyel (TensorflowJS), így ez a technológia illett leginkább a jelenlegi feladat webes környezetébe. A framework-öt nem közvetlenül, hanem a Keras API-on keresztül használtam, így maguk a modellek is ennek segítségével épültek.

A szerver oldali technológiát illetően a TensorflowJS nyelvi sajátosságai miatt döntöttem a NodeJS runtime mellett, mivel natívan támogatja a nyelvet. Ezen belül, az ExpressJS szerver oldali technológiára épülő NestJS framework hajtja meg a szerver oldali kódot, melyre a Typescript megbízható típusossága mellett a nagy-

[68] *Multitier/n-tier architecture*, melyben a funkcionálisan nem összetartozó komponensek fizikailag is elkülönülnek, nem összekeverendő a *multilayered architecture* fogalmával, mely alatt egyszerűen a funkcionális rétegek elkülönítését értik ugyanazon a fizikai komponensen.

fokú adatbázissal történő összehangoltság miatt esett a választás: több népszerű és könnyen kezelhető objektum-relációs leképező motort[69] is támogat, melyek közül a Typescript t0ámogatás miatt a TypeORM könyvtárt választottam. A szerver réteg „alatt" futó adatbázis MSSQL dialektusban készült, de rendkívül egyszerű mivolta miatt, bármilyen SQL vagy NO-SQL adatbázis ugyan úgy használható lett volna.

A kliens oldali arcfelismerő komponens megvalósítását a Google által fejlesztett Firebase segítségével oldottam meg, mely natívan támogatja az Android fejlesztői környezetet és széleskörű támogatással bír ezen a technológián, így a választás egyértelmű volt. Közismert, hogy Androidra jelenleg 2 nyelven is történik fejlesztés: az eredeti Java nyelv mellett megjelent az egyre nagyobb népszerűségnek örvendő, funkcionális alapokon nyugvó Kotlin nyelv is, melyet én is használtam a szoftver megírásakor. A döntés motivációja a nyelv által nyújtott magas fokú kontextusfüggő aszinkronitás, mely elengedhetetlen része bármilyen úgynevezett *hosszútávú folyamat*[70] akkomodálásának, melyek közé az arcelemzés is tartozik (relatíve rövid időtartama ellenére is). Ebben az esetben ugyanis nem elég egyszerűen „bevárni" a folyamatot (aszinkronitás), biztosítani kell azt is, hogy Android környezetben a felhasználó által kezdeményezett kontextusváltások (pl. más alkalmazás előtérbe hozása) zökkenőmentesen megtörténhessen, erre pedig kiválóan alkalmas a Kotlin nyelv *coroutine-nak* nevezett eleme, melyet lehetséges a alkalmazáskontextushoz kötni.

7.3. A rendszer komponensei

Az alábbiakban részletes leírásra kerülnek a implementált rendszer elemeinek komponensei és azok szerepei az előbb leírt funkcionalitásban. A leírás a többrétegű architektúra „legalsónak" tekintett adatbázis rétegétől indulva mutatja be szerver, majd az mobilkliens rétegeket, ezután pedig a rendszer gépi tanulásért felelős része következik. Ahogy az egy korábbi fejezetben részletesen bemutatásra került, többféle technológia is kínál eszközöket a feladat megoldására, így a legmeghatározóbb tényező a gépi tanulást biztosító réteggel történő interoperabilitás lett.

15. ábra. A rendszer egyetlen adatbázistáblája

[69] ORM, avagy *Object Relational Mapper*, mely lehetővé teszi az adatbázis objektumokkal történő kommunikációt megszokott, „POCO" runtime objektumok használatával.

[70] long running process

7.3.1. Az adatbázis

Az MSSQL alapú adatbázis végtelen egyszerű, egytáblás réteg, melyet a rendszer a regisztrált felhasználók eltárolására használ. A *DeviceId* mező tárolja a mobilklienstől származó generált azonosítót, míg a *Token* a server által generált JWT mezője, melyek a rendszerben lehetővé teszik, hogy csak olyan eszköz kerüljön kiszolgálásra, melyet előzetesen regisztráltak a serveren. A *Username* és a *Landmarks* mezők a felhasználó azonosításában játszanak szerepet, előbbi értelemszerűen annak a regisztráció során választott felhasználónevét, míg az utóbbi a kliens által a regisztráció során készített képből kinyert landmark pontokat tárolja sorfolytonos string adattípusként.

7.3.2. A webszerver

A szerver a NodeJS környezetben megszokott, különálló (type)script file-okból áll, melyek szerkezete csak részben felel meg az MVC-ből ismert konvencióknak egyfajta hibrid mintát követve a komponens alapú konvencióval. Míg a legtöbb rész komponens alapú mintát ölt egy-egy üzleti igény kizárólagos felelőseként, melyhez a repository-tól a egészen a controllerekig biztosítanak funkcionalitást, addig a webes adattovábbításért felelős közös elemek (pl. adattovábbító objektumok, vagy dto-k) közös csoportokba kerültek. Így például a webalkalmazásban előforduló adatfolyamok sem hagyományos MVC rétegek között, hanem az előbb említett komponensek között és azokban vertikálisan történik. Az alkalmazás injektált függőségekkel dolgozik, azonban ezeket nem egy központi elem közvetíti a többi komponens felé, hanem a modul alapú keretrendszerek konvencióinak megfelelően minden komponens rendelkezik egy dedikált szkrkipttel, mely a kiajánlott és igényelt függőségeket teszi elérhetővé. Ennek a kialakításnak nincs technológiai oldalról megindokolható oka, pusztán a feladat szempontjából összetartozó adatok összefogása végett döntöttem mellette. Az alábbiakban röviden bemutatásra kerülnek a különféle üzleti igények megtestesüléseit jelképező modulok ebben betöltött szerepük szerint.

Az alkalmazáson belül erőforrásként jelennek meg a statikus adatok mint amilyen az SSL tanúsítvány, vagy maguk a betanított modellek file-jai, de ilyenek a különféle konfiguráció során használt JSON formátumú file-ok is, vagy az alkalmazás gyökerét képező szkriptek. Ezeken kívül kisebb modulokba kerültek egyéb segédfunkciókat lebonyolító elemek, mint a globális kivételkezelő (NodeJS környezetben ún. *filter*), az adattovábbításért felelős csomagosztályok[71], vagy a neurális háló válaszait olvasható formátumra fordító komponens is.

Az autentikációs komponens az egyik legfontosabb elem, melynek szerviz szkriptje felelős a különféle autentikációs folyamatok implementálásért, melyet a JWT tech-

[71] DTO, azaz Data Transfer Object

nológiákat lehetővé tevő szerviz szkriptekkel karöltve tesz meg. A modulhoz tartozik természetesen egy controller is, mely fogadja a regisztrációhoz kötődő kliensoldali hívásokat, illetve itt kaptak helyett a Node-os környezetben *guardoknak* nevezett szkiptek is, melyek lehetővé teszik, hogy bizonyos „védett" olalak csak belentkezett felhasználók számára érhetőek el.

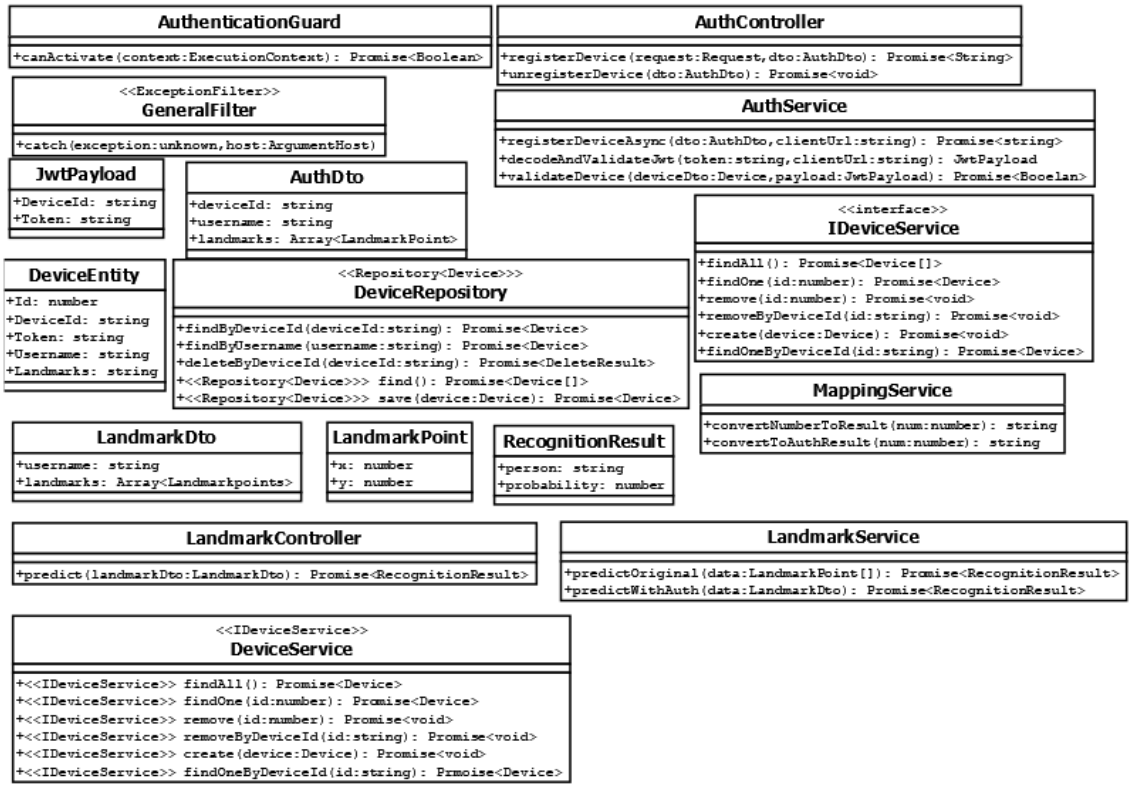
Az adatbázissal kapcsolatot tartó TypeORM által kezelt entitás szkriptek külön modulban helyezkednek el és interface-eket használó szerviz, illetve repozitori elem is tartozik hozzájuk, hogy a komponensen belül is minimalizálva legyen a típuskötődés. Eme folyamatok aszinkron megvalósítást kaptak melyekhez a többi komponens csak a szerviz szkripten keresztül férhet hozzájuk, mely konvenciótól függetlenül megszokott eljárás.

Az alkalmazás szívét a landmark modul adja, mely a regisztrált felhasználók számára elérhető autentikációs folyamatért felelős. E modul szerviz osztálya használja a betanított neurális háló állományait és vegzi el a konverziót a megfelelő, korábban említett modul segítségével az adatok oda-vissza konvertálását. Lényegében, ez a komponens végzi el magát az inferenciát, míg controller osztályát hívja és azon keresztül is értesül a válaszról a kliens. A szerver által alkalmazott osztályok sorát a 16a. mutatja be.

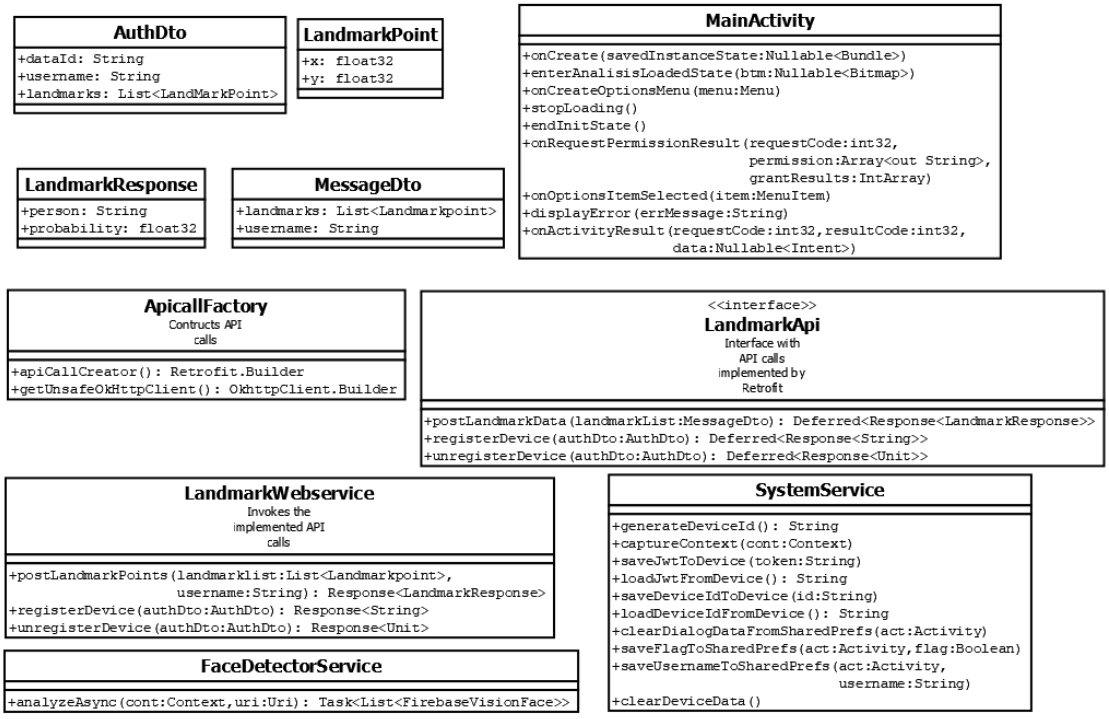
7.3.3. Az android kliens

Az Android kliens a szerverétől nagyban eltérő mintát kapott, lényegesen egyszerűbb és sokkal inkább emlékeztet a hagyományos MVC alapú architektúráktól megszokott képre. A szerveren helyett folglaló DTO osztályoknak itt is megtalálhatóak a tükörképként funkcionáló Kotlin adat osztályok. Az egyszerűen rétegzett alkalmazás az XML formátumú felhasználó felület leíró file-okat leszámítva egy szerviz rétegből és magából az alkalmazásból áll, mely Android környezet révén maga a *MainActivity*. Az alkalmazás egyszerű felépítése miatt nem injektált függések alapján működik, minden a működéshez szükséges rész magában az activity-ben kerül tárolásra.

A szervizek egyik fele a webes kommunikációért felel, mint amilyen a Retrofit API-t használó *ApicallFactory* is, mely az OkHttppClient segítségével megvalósítja magukat a http hívásokat, míg másik részük a landmark pontok felismerésében játszik szerepet. Így például itt kapott helyet a *FaceDetectorService* is, mely elrejti az alkalmazás elől a Firebase API konfigurációját és inicializációját, illetve magát a működés részleteit is. Ebben a tekintetben különálló a *SystemService*, mely az Android rendszerhez közelebb álló funkciókat valósítja meg, legfőképp a korábban már említett SharedPreferences könyvtárba történő adatok mentése és onnan való törlése került ebbe az osztályban, mely segédosztály-jellege miatt Kotlin Singleton-osztályként lett implementálva.



(a) szerver



(b) kliens

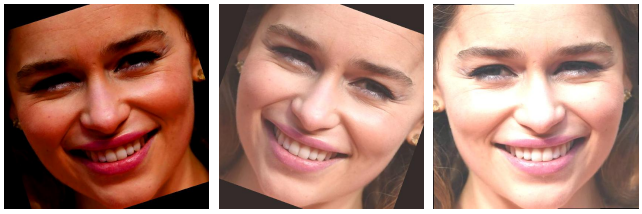
16. ábra. A rendszerarhcitektúra osztályai

Az alkalmazás legnagyobb osztálya maga az alkalmazás magja, a *MainActivity*, mely egyben az egyetlen belépési pont is. A szervizhívásokon kívül az Android rendszerben gyakori életciklusra jellemző metódusok (pl. *OnCreate()*, mely az alkalmazás első indításánál inicializálja azt) és kontexttusfüggő hívások kerültek ide különböző metódusokba tömörítve. Ezek mellett rengeteg *onACTION* konvenciót követő eseményfigyelő implementáció található még az Activity forráskódjában mint amilyen például az *onRequestPermissionsResult()*. A kliens osztálydiagramját a 16b ábra mutatja be.

7.4. Gépi tanulás az implementált rendszerben

Az alkalmazott gépi tanuló módszerek nem különálló komponensként vannak jelen a rendszerben, hanem az azok mintegy „végtermékeként" létrejövő, betanított modellek kerülnek felhasználásra. Ebben a részben kizárólag a szerver által alkalmazott két neurális háló és az azokhoz vezető preprocesszálási előmunkálatok kerülnek bemutatásra.

(a) eredeti és kivágott



(b) dlibbel módosított

17. ábra. Néhány feldolgozott példa az összeállított *emilia set*-ből

7.4.1. Preprocesszálás

Minthogy az a korábbi fejezetekből is megállapítható, a legtöbb mesterséges intelligenciát alapul vevő vagy gépi tanuló megoldás alapja valamilyen tanítóminta, melyet feldolgozva a rendszer megtanulja, hogy egyes inputokra milyen outputokat kell előállítania. Különösen igaz ez az irányított/felügyelt tanulási módszertanokat követő architektúrákra, mint amilyen technikával a jelen megoldás is készült. Külön nehézség volt azonban, hogy habár a kliens oldali Firebase API megbízható munkát végez az adatok kinyerésével, speciális 131 arcpontos megvalósítása egyedülálló az általában 30 vagy 60 pontos megvalósítások között ami magával vonta a problémát, hogy előfeldolgozott landmark adatokhoz rendkívül nehéz hozzájutni.Épp ezért az előfeldolgozási fázisban mindkét háló esetén előbb magukat a tanítómintákat kellett előállítani.

Mivel a jelen implementációval nem a már meglévő arcfelimserő módszerekkel

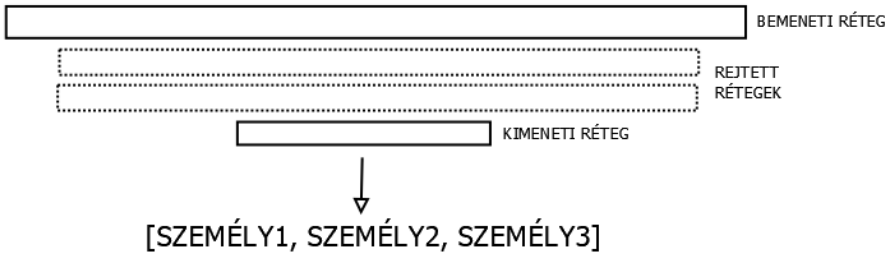
akartam versenyezni, hanem egy elosztott rendszer előnyeit bemutatni, így kiterjedt arcadatbázisok helyett híresség arcképeit gyűjtöttem ki egy erre a célra készített, Google keresés alapú *web crawler* segítségével. A jupyter notebookban Python nyelven írt néhány soros megoldás egy kulcsszót várt bemenetként és a kiadott, arcot is tartalmazó találatokat egy mappába mentette. Több híresség is indítottam keresést, de végül 3 színészre szűkítettem az alkalmazást: Emilia Clarke, Patrick Stewart és Natalie Portman. Képanyagokat gyűjtöttem még további 3 személyről is, azonban ezt már nem sikerült a megoldásba implementálni. Az adatgyűjtés fázis végén a színészekről végül egyenként 70-100, legálisan is felhasználható képet sikerült előállítani, ezek a számok viszont nem elegendőek egy háló megfelelő betanítására, így a Keras API nyújtotta képaugmentációs eljárásoknak vetettem alá a mintát feldúsítva azt tízezres nagyságrendre. Itt érdemes megjegyezni, hogy ez vitathatatlanul a háló specializálódását vonja magaután, de mivel nem volt célom a jelenlegi megoldásokkal versenyezni, így ezt megengedhető gyengeségnek tekintettem. Mivel a minta eredetileg a klienshez készült volna, így nem csak a képen lévő arc helyzetét, de kontrasztot és a világosságot is véletlenszerűen módosítottam, ami aztán később a már csak landmarkpontokat kapott háló esetében - az extém példáktól eltekintve - teljesen irreleváns. Feltehetőleg ez a tény is befolyásolja a háló megbízhatóságát. A képekből valódi tanítóminta előállítására az android kliens egy speciális, külön erre a célra átalakított változatát használtam. Ez a szoftver emulátoron futott és egy listányi képet olvasott be, melyeken egyesével végigiterálva elvégezte a felismerést, majd a folyamat végeredményeként elmentette ezt egy csv formátumú file-ba. Habár egy elemzés 1-2 másodperc alatt végbemegy, a teljes tanítómintán végigmenni nem sikerült volna belátható időn belül, így végül a dlib könyvtár segítségével az arcokat határoló négyszög mentén körbevágtam a képeket és ez került az alkalmazásba, jelentősen csökkentve a képmérettel együtt a generálás időtartamát is. A személyekhez integer kódolás tartozik, a háló ezek alapján tanulja meg a személyazonosságokat és így is jelez vissza az inferencia eredményéről. A tanítóminta alakja egy kilapított, egydimenziós lista, melyben vesszővel elválasztva állnak az egyes pontok X majd Y koordinátái a sorokat pedig a személy integer kódja zárja.

A második háló esetében sokkal kevesebb teeendő elvégzésére volt szükség, ugyanis arra a célre megfelelt az adatbázisokról értekező részben már bemutatott Poznań Egyetem által rendelkezésemre bocsátott adatbázis, mely 100 egyén egyenként 22 képéből áll. Ezeket elegendő volt rendszerezni, majd az előbb említett eljárással méretre vágni, miután az alkalmazás további gond nélkül feldolgozta azt. Jelen esetben a tanítóminta nem egyszerűen az adott személyek pontjaiból állt, hanem párokat kellett feldolgoznia (ugyanaz a személy, vagy sem), így a létrehozott mintákat pozitív és negatív párokba kellett állítani. Így a tanítóminta az adott párok egymás mellé rendezett pontjaiból állt, melyet a két személy megegyezőségét jelző

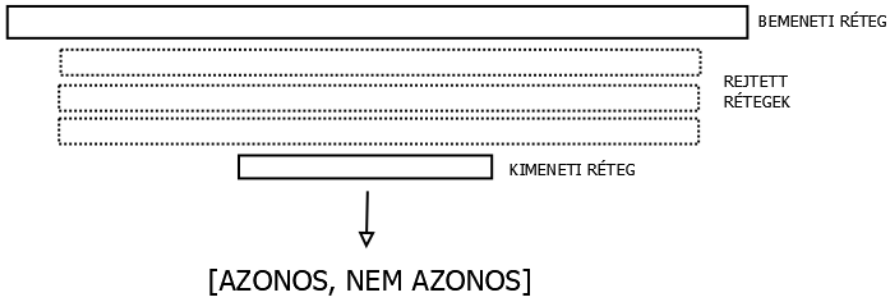
integer kódolás zárt.

7.4.2. Az implementált neurális hálók

Mivel jelen munka középpontjában az osztott gépi tanuló módszerek vizsgálata áll, így nem tekintettem feladatnak, hogy a jelenlegi modern megoldásokkal versenyre kelni tudó architektúrát hozzak létre, mivel hiperoptimalizált hálókkal sem tudtam volna jobban alátámasztani a jelen műben bemutatni kívánt szempontokat. Ehelyett a lehető legegyszerűbb, de még működő modelleket szerettem volna összeállítani, melyek betanítása belátható időn belül megejthető a rendelkezésre álló adatokkal. Emiatt döntöttem mindkét háló esetében úgy, hogy azokat egyszerű, teljesen csatlakozó rétegekből fogom összeállítani. Ez természetesen nem jelenti azt, hogy az egyszerűség keretein belül ne akartam volna a legjobb eredményeket felmutatni, így többféle súlyinicializálással, aktivációs függvénnyel, (adat)kötegmérettel[72], epok számmal[73], rétegsűrűséggel, hibafüggvénnyel és optimalizálóval is próbálkoztam, míg sikerült összeállítani az általam már megfelelőnek ítélt architektúrát. Az előre betanított személyeket felismerő háló (a továbbiakban I) pontossága 75-80% körül, míg a személyeket összehasonlító háló (melyre ezentúl úgy hivatkozok, mint II) 85% és 92% százalék körül mozog, mely számadatok többszöri, a hálók által a tanítás során nem látott adatokkal történő tesztek során mértem szintén a Keras API segítségével. A két háló architektúra felépítése elég hasonló, így egy direkt összehasonlítás helyett az egyes,



(a) I



(b) II

18. ábra. Implementált neurális háló architektúrák sematikus ábrája

[72] *batch size*
[73] *epoch*

betanítás során fontos úgynevezett hiperparamétereket fogom számba venni. A megoldandó probléma felépítése miatt az egyedüli teljesen eltérő paraméter a bemeneti és kimeneti rétegek száma: I esetében a bemeneti paraméterek száma megegyezik a kilapított koordináták számával, azaz 262, melyből az 3 db valószínűségi értéket állít elő a 3 személynek megfelelően így ekkora lesz a kimeneti réteg is, II-nél pedig két, az előző számsor áll egymás mellett bemenetként, így a bemenet nagysága is 524, melyből a háló egy egyszerű bináris megegyező (1) vagy különböző (0) értéket állít elő. Mindkét háló utolsó rétege egy teljesen összekötött, softmax aktivációs függvénnyel ellátott réteg, melyre a kategorizálás miatt volt szükség.

Mint ahogy az korábban a neurális hálók témakörénél is szerepelt, a súlyelosztás egy olyan paraméter, melyet a bemeneti paraméterértékek helyén álló neuronok „fontosságát" megadva a gépi tanuló módszerek a betanulás során állítanak elő, így a betanított háló lényegében maga is egy súlykiosztás, egy rács, úgymond, mely kiosztásának megfelelően engedi át a különféle bemeneti értékeket. Ezeket a súlyokat többféleképpen is inicializálni lehet, lehetséges például az összes értéket 0-ra állítani, vagy véletlenszerű számadatokkal feltölteni valamilyen eloszlás szerint. Többféle súlykiosztást is kipróbáltam míg végül a Keras API lecun_uniform inicializálása mellett döntöttem, mely az eredeti 1988-as Yann LeCun által kidolgozott megoldás [48] módosított változata. A megoldás egyenletes eloszlású véletlen számokat generál egy [- határ, + határ] halmazból, ahol a határértéket a következőképpen számolják:

$$határ = \sqrt{3/fan\_in}$$

ahol fan_in a bemeneti változók értéke.

Az aktivációs függvény a hálókat alkotó neuronok értékét befolyásoló elem, mely a függvény eredményének függvényében állítja be az adott neuronnal kapcsolatban álló neuronok súlyozottságát. A két megoldás esetében eltérő aktivációs függvényt eredményezett a kísérletezés: míg II-nél az igen gyakran alkalmazott ReLU[74], addig I-nél ennek egy változata a SELU[75] bizonyult a legeredményesebbnek. Az előbbi függvény negatívok esetén konzekvensen 0-t, míg pozitív számoknál magukat a számokat adja vissza (innen a linearitás). A SELU [49] függvény jóval bonyolultabb, mert negatív és pozitív értékeket is elő kell állítania, illetve abban az esetben használják, ha mérsékelni akarják az értékeket.

$$ReLU(x) = \begin{cases} 0, & \text{ha } x < 0 \\ x, & \text{ha } x <= 0 \end{cases} \quad (1) \qquad SELU(x) = \lambda \begin{cases} x, & \text{ha } x > 0 \\ \alpha e^x, & \text{ha } x <= 0 \end{cases} \quad (2)$$

[74]Rectified Linear Unit
[75]Scaled Exponential Linear Unit

Nagy vonalakban, a háló betanítása során a kötegméret azon halmazokak a számossága, amelyekre a tanítóminta felosztásra kerül: a folyamat mindig feldolgoz egy kötegméretnyit a mintából és csak ezután történik meg a jelenlegi előrehaladás „jóságának" megállapítása az előre elkészített elvárt adatokkal. Egy epok egy ilyen adatköteg feldolgozása, így a betanítás során a maximum epokszám meghatározza, hogy a folyamat maximum hányszor ismételheti meg a tanulás során a kiértékelést. Nem szükséges, hogy eljusson idáig: jelen esetben a maximum 1000-2000 között mozgott, de nagyjából 500 ismétlés után a túltanulást megelőzni hivatott korai leállítás[76] általában megállította a hálók tanulási folyamatát. A 10.000-es nagyságú minta miatt I esetében a kötegszám 500, míg a maximum epokok száma 1500, II esetében pedig a kötegszám 32, az epokok maximuma pedig 1000 volt. Ezeket az értékeket logaritmikus keresés alapú kísérletek alapján kaptam.

Igencsak leegyszerűsítve, a betanítási folyamatnál az optimalizáló[77] az a függvény melynek segítségével a folyamat egyre inkább igyekszik alacsonyabb és alacsonyabb hibaértékekkel dolgozni, másszóval meghatározza azt a sebességet, mellyel a betanítás folyamata konvergálni fog egyre alacsonyabb szintekre a hibatérben. Ha a függvény túlzottan kisléptékű, akkor a folyamat túl lassú lesz, ha túl magas lépéseket diktál, akkor pedig nő az esély a kedvezőbb lokális optimumok kihagyására. Jelen esetben is eltérő optimalizálókat használtam mindkét megoldás esetén. II esetében az úgynevezett *Adam* optimalizáló érte el a legpontosabb eredményeket. Ez a módszer a népszerű Sztohasztikus Gradiensen [78] alapul, ám míg az minden egyes iterációnál véletlenül választott irányba elmozdulva próbálja megtalálni a minnél kedvezőbb értékeket [50] és a tanulási folyamat során végig konstans tanulási rátával[79] dolgozik, addig az Adam dinamikusan alakítja azt a folyamat során [51] és több konvergáló algoritmust ötvözve hatákonyabb, kisebb memóriaigényű és a hiperparaméterekre is kevésbé érzékeny, mint az SGD. Az Adam esetében így a tanulási ráta meghatározásánál kezdő értékről érdemes beszélni, ez az érték 0.0001 lett végleleges megoldásban. A többi paraméter alapértéken működik, melyet a Keras API tervezői az idézett tanulmány alapján állítottak be. A kevésbé jó minőségű adathalmazból tanuló I az Adam algoritmus egy továbbfejlesztett változatát használja, melyet Adamaxnak neveznek. Ez az algoritmus is az előbb említett SGD módszeren alapszik, de egy lényeges változtatást eszközöltek rajta az algoritmus eredeti szerzői. Az Adamnél az egyes súlyok értékei fordítottan arányosak a jelenlegi és korábbi súlyokra jellemző gradiens konvergálásának mértékével, melyet a kiinduló ponttól L^p normával adnak meg. Állításuk szerint elegendően nagy p értékeknél az algoritmus instabil lehet, ám ha a rendszer megengedi, hogy p a végtelenbe tartson, akkor

[76] early stopping
[77] *optimizer*
[78] *Stochastic Gradient*, SGD
[79] *learning rate*, meghatározza az optimalizáló lépésközét

	I	II
Bemeneti paraméterek száma	262	524
Kimeneti paraméterek száma	3	2
Kötegméret	500	32
Maximális epokszám	1500	1000
Súlyinicializáció	egyenletes LeCun	egyenletes LeCun
Aktivációs függvény	SELU	ReLU
Hibafüggvény	SCC	SCC
Optimalizáló függvény	Adamax	Adam
Korai megállás türelmi lépésszáma	5	5

1. táblázat. Az implementált hálók hiperparaméterei

stabilizálódik, ezt nevezik a szerzők Adamaxnak [51]. Az Adamax hiperparaméterei is megegyeznek az Adaméval, így jelen esetben is csak a tanulási rátán állítottam, ennek értéke 0.00001.

Az utolsó általam részletezett hiperparaméter a hibaarány[80], melyet az aktuális adatköteg feldolgozása után a tanulási folyamat előrehaladásának megállapítására használnak. Mindkét modell a klasszifikációs problémák során előszeretettel alkalmazott *kereszt-entropikus hibafüggvény*[81] egy változatát használja, mely a Keras API alatt *SparseCategoricalCrossentropy* néven található[82] meg és a dokumentáció alapján kifejezetten a jelen esetben is használatos integer alapú bemenetekre lett kifejlesztve.

Összegzésképpen, a implementált hálók adatait 1. számú táblázat foglalja össze tömbösítve jelenítve meg az előbb kifejtett adatokat, a végeredményként kapott háló architektúrát pedig 18. ábra mutatja be.

7.5. A rétegeltség vizsgálata az implementált rendszerben

Az alábbiakban elemzem a lefejlesztett rendszert az előző fejezetben a rétegzett gépi tanuló architektúrák feltételezhető felépítése alapján. A fentiek alapján a cserélhető implementáció egyértelműen jellemző a rendszer mindkét komponensére, hiszen kliensoldalon több, az irodalomban szereplő implementáció közül is választhattam volna, szerveroldalon pedig két lehetséges implementáció is rendelkezésre áll. Éppígy teljesül az alá-fölé rendeltségi viszony és az egyirányú adatfolyam is, mivel a kliens kimenetét felhasználó szerver az általa kibocsátott végterméket (az azonosított személyt, vagy a hasonlóságot) nem képes visszaalakítani a kliens által értelmezhető adattá. Érdekesebb viszont a különálló fejlesztés lehetősége, ugyanis jelen esetben ez nem ál fent, ugyanis a landmarkokat előállítani tudó komponenst

[80] loss
[81] *cross entropy loss*
[82] SCC, szétszóródáson alapuló, kategorikus keresztentrópia

használtam fel tanítóminta generálására. Ez azonban csak részben probléma: a tanítóminta landmark részét fel lehetett volna használni a szerveren implementált két megoldás közül a II. betanítására is, az I. konfigurációt pedig betaníthattam volna utána is. Így elviekben megoldható, hogy a kliens és a szerver gépi módszereit egyazon időben fejlesszék.

Másik érdekesség, hogy amennyiben a felhasználó fél már rendelkezik egy a két komponens közül bármelyik funkcionalitására hasonlító megoldással, akkor minimális átalakítás után használatba veheti a rendszerben a másik szerepet betöltő elemet. Ez azonban jelen konkrét implementációra csak akkor igaz, ha a Firebase API által használt pontkonfigurációt használja az ügyfél megoldása is, egy általánosabb 32, vagy 64 pontos megoldás esetén jóval kedvezőbbek a lehetőségek. Hasonlóképp bővíthető lenne a szerver oldali megoldás egyéb szolgáltatásokkal is, melyek landmarkpontokat várnak bemenetként: [52] például hazugságvizsgáló rendszert, [2] pedig érzelemdetektáló megoldást dolgozott ki, de ezek nyomán feltehetőleg bármilyen olyan megoldás alkalmazható lenne, mely az arc helyzetének vagy mimikájának megváltozásával jár.

A fent említett indokok miatt tehát kijelenthető, hogy az általam implementált rendszer rendelkezik a fentebb leírt rétegzett gépi tanuló módszerektől várható jellegzetességekkel. Érdemes azonban megjegyzeni, hogy az implementáció nem használja ki a felvázolt többrétegű rendszertől várható teljes potenciált, így inkább úgy érdemes rá tekinteni, mint egy prototípus, mely igazolni próbálja a fent leírtak valóságalapját. Mint minden prototípus, a jelenlegi implementáció előnyei is csak más megoldásokkal összehasonlítva értékelhetőek, így a következő szekcióban ez kerül leírásra.

8. Eredmények az irodalom tükrében

A feladat egyediségéből kiindulva nehéz megfelelő összehasonlítási alapot találni a jelenleg is meglévő technológiák között. Ugyan az irodalomban nem ritka több neurális hálót alkalmazó módszerekről olvasni, azonban leggyakrabban ez a betanítás során jellemző, nem a már megvalósított rendszerben. Például [53] és [54] az úgynevezett *generatív ellenséges hálók*[83] technikáját alkalmazzák a feladatmegoldás során, melynek lényege, hogy több háló egymással versenyezve próbálja meg a másikat kicselezni és így minél jobb végeredményt elérni, míg [55] *bináris particionált neurális hálót*[84] használ, melyben külön hálót tanítanak be minden kategóriára a tanítási idő lerövidítésée végett. A gyakorlati megvalósítást háttérbe szorító implementációk nagy részéről nem is áll rendelkezésre elég adat egy átfogó összehasonlításhoz. Mivel a legtöbb megoldás újabb gépi látó komponensek kifejlesztésére törekszik, így legfeljebb ezt a részt lehetne összehasonlítani: a specializált megoldások általánosságban 90% fölötti pontossággal rendelkeznek.Viszont jelen munkának nem ez az egy komponens állt a középpontjában így különösebb magyarázat nélkül kijelenthető, hogy az implementáció ezen részei elmaradnak az irodalomi átlagtól.

Van példa az irodalomban azonban olyan forrásokra, melyek a jelenlegi módszerekhez hasonló megoldások kidolgozását taglalják, ezek között szerepel vékony kliensen alapuló megoldás [56], de találkozni vastagkliensű architektúrákkal is [1][57][58]. Az előző szekció elején részletesen bemutatásra került, hogy milyen architektúrális előnyei lehetnek egy ilyen implementációnak, így jelen fejezetben a kidolgozott rendszer technikai tulajdonságait mutatom be már létező megoldásokkal összehasonlítva.

8.1. Adattárolás

A 15. ábrán látható a rendszer egyetlen adattáblája, melyben a landmarkpontokon kívül néhány egyéb adat is letárolásra került. Természetesen a landmarkokkal történő azonosításhoz minimálisan elegendőek maguk a pontok, így nem csak a személyenkénti tárolásra szoruló adatok mennyisége minimális, de a keresési idő is lerövidíthető. Ennek belátásához érdemes áttekinteni röviden, hogy milyen formában nyernek ki és keresnek arcfelismeréshez használt adatokat adatbázisból.

Az arcfelismerésben felhasznált legalapvetőbb adat maga az arckép, ezek tárolására korábban felcímkézett képeket használtak, majd idővel különböző absztrakciókat próbáltak a képekből és azok egyes fontosnak tartott régióiból előállítani (pl. színek, szegmensek alapján klaszterezés), melyre az irodalomban *Tartalom alapú képelőhívás*[85] néven hivatkoznak [59]. Ezen ismereteket alkalmazták arcadatbázisok

[83] *generative adversarial network*
[84] *binary-pair partitioned neural network*
[85] *Content Based Image Retrieval*

készítéséhez. [44] például képszegmensekből egy, az iparban gyakran alkalmazott [59] diszkrét koszinusz transzformációnak nevezett folyamattal nyert ki adatokat JPEG formátumú képekből, majd az ezekből származó együtthatókat tárolta le. Az együtthatókban megjelenő különböző képi tulajdonságok (pl. szín) alapján histogrammokat készített, melykből euklidészi távolságot számolva történt meg az összehasonlítás.

Ami az adattárolást illeti, az ide vonatkozó irodalom nem fogalmaz meg pontos adatokat a tárolt adatok által elfoglalt helyről, de említi, hogy az együtthatókat a képek pixeleiből számítják [60]. Ezek a tárolási módszerek különféle képtömörítéses módszereket alkalmaznak, azonban nem próbálják kiválogatni a képek arcfelismerés szempontjából hasznos tulajdonságait. Hogy egyértelműbb legyen a landmark pontok eltárolásának előnye a jelenleg elterjedt módszerekhez képest, érdemes összevetni a jelen munka eredményeit néhány elterjedt képtárolási technika adataival.

```
1215/1215 [==============================] - 0s 115us/sample - loss: 0.1306 - accuracy: 0.9374
```

(a) I

```
8221/8221 [==============================] - 1s 64us/sample - loss: 0.2018 - accuracy: 0.9174
```

(b) II

19. ábra. I. és II. háló tesztjeinek eredményei

8.1.1. Az adattárolás mérése

Összehasonlítás képpen megnéztem egy kép méretét a két legelterjedteb képtárolási módszer - azaz adatbázis, illetve fájlrendszer - szerinti összehasonlításban. Az előbbi esetében megmértem a letárolt kép méretét több adatbázisformátum használata mellett. A mérések természetesen adatbázisspecifikusak lennének, így jelen esetben a megoldás adatbázisát használtam fel mérési alapnak. Az MSSQL keretein belül többféle képpen is lehet képeket tárolni, ezek közül az adatbázis által natívan támogatott *image* formátum kifejezetten erre a célra lett kialakítva, emellett szövegként (*nvarchar*) tároltam el az implementált megoldás 131 pontos, elméleti 32 pontos változatát, illetve a kép *Base64* kódolással szöveggé alakított változatát. Az implementált 131 pontos változat esetében a lementett adatot használtam fel, míg a 32 pontos változatnál - implementáció hiányában - véletlenszerűen generált integer koordinátákat alakítottam karakteres formátummá. Az image típus tesztelésére a konkrét képet olvastam be, a base64 kódoláshoz pedig az eredeti képet alakítottam át és az így kapott karaktereket mentettem le. Az összehasonlításhoz az MSSQL *datalength* parancsát használtam az adott rekord bizonyos oszlopára mely byte-ban visszaadja a megmért adat méretét.

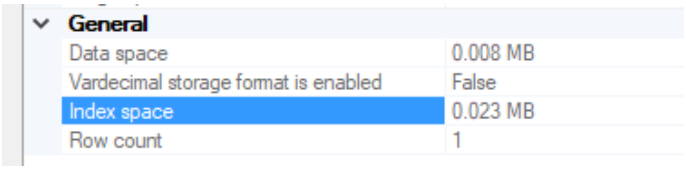
Az állományrendszer esetében [61] kutatásait vettem alapul a méréseket helyet-

Implementáció	Méret (bájtban)
Eredeti (jpg)	5120
Base64	12104
MSSQL *image*	4537
131 pontos landmark	5120
32 pontos landmark	458
DCT (becsült)	2778
DWT (becsült)	487

2. táblázat. Az adattárolás összehasonlítása egyéb gyakori megoldásokkal

tesítő számoláshoz: egy 41 kB méretű JPG formátumú képet tömörít össze előbb diszkrét koszinusz transzformációs módszerrel (25 kB), majd a saját maga által javasolt, *discrete wavelet transform* nevezetű módszerrel (4kB). Ezekből az adatokból egyszerű arányszámokkal következtettem a saját 5 kB-os referencia képem tömörített méreteire. A konkrét számadatokat a 2. táblázat foglalja össze.

Eme adatok mellett érdemes egy extrém példát is megvizsgálni az összehasolítás kedvéért, ahol direkt a minél alacsonyabb méret volt a fő szempont. [1] egy olyan módszert dolgozott ki, melynek segítségével nem csak tömörítési eljárásnak vetik alá a képeket, de mindezt úgy teszik, hogy közben a lehető legkisebb, de még arcfelismerésre használható képek keletkezzenek. Kutatásuk során olyan vastagkliensű megoldást akartak létrehozni, melyben a mobil kliensen kerül eltárolásra a teljes képadatbázis, így szükséges volt egy gyorsan továbbítható, kevés helyet foglaló formátum, melyet lehetséges arcfelismerő műveleteket végezni. Ehhez több módszert is kidolgoztak, de a leghatékonyabb egy a diszkrét koszinusz transzformáción alapuló volt, mely a pixelekből számított együtthatókkal dolgozik. A tesztek során egy 400 mintából álló, nagyjából 3.7 MB-os adatbázist vettek alapul, melyet a fenti módszerrel sikerült kimagasló 14 KB-os méretűre tömöríteniük, azaz 1 transzformált kép nagyjából 36 byte méretű lett, és még használható volt arcfelismerésre. Látható tehát, hogy a jelen implementáció elmarad ugyan a fenti extrém példától, azonban a képtárolási megolsásokhoz képest jól teljesít, mi több, megvan az az előnye, hogy nem kell rajta utómunkát végezni felismerés során. Az előbi esetében a felismerést megelőző kitömörítés átlagosan 12.8 másodperc volt, melyhez a felismerés 0.25 másodpercet adott hozzá. Ez pedig támpontot ad a sebesség kritériumának vizsgálatára is.



20. ábra. 1 rekord mérete az adatbázisban

8.2. Adatkeresés és továbbítás

A sebességet a minták összehasonlításának gyorsasága, illetve az esetleges előmunkálatok sebessége határozza meg. Ez a folyamat lényegében két lépésől áll: először ki kell olvasni az adatbázisból egy rekordot, majd azon elvégezni az összehasonlítást, ehhez számítódik hozzá az előfeldolgozás ideje. [62] a fent említett technikát színhisztogrammokra alkalmazva készített méréseket átalgosan 500 kép felhasználásával: a szakirodalomban alkamazott módszerek mérései szerint nagyjából 3 másodperc alatt találták meg a kívánt képet, ehhez képest [56]-nek átlagosan 2 másodperces időt sikerült elérni, míg sajátfejlesztésű megoldásuknak ezt sikerült 1 másosperc körüli értékre javítani.

8.2.1. Az adatkeresés mérése

A fent említett adatokkal összehasonlítandó, a jelen implementációról is szükségesek mérési eredmények. Azonban, a teljes rendszer pontos „end-to-end" mérése az osztott kialakítás miatt meglehetősen nehézkes, így a különálló komponensekről készültek mérések. Így például a gépi tanuló környezet által előállított eredményről inferencia mérés készült a Tensorflow által kínált tesztkörnyezet felhasználásával, mely a már betanított modellen hajt végre a tesztminta nagyságával megegyző összehasonlítást, mellyel szimulálható a lehetséges legrosszab keresés, azaz mikor a keresett elem megtalálásához az egész mintasoron szükséges végighaladni. Az I. háló esetében 1215 teszmintán mérve átlagosan 115 mikroszekundum/minta sebességgel a teljes soron kevesebb mint 1 másodperc alatt ment végbe a teszt, míg a II. esetében 8221 mintán 64 mikroszekundum/minta sebességgel ért el egy hasonló 1 másodperc körüli értéket. A Tensorflow kimenetét a 19. ábra mutatja be.

Ehhez az eredményhez még hozzáadódik az adatbáziselérés ideje, mely adatbázis, terheltség és rendszerfüggő, de összehasonlítás képpen a saját fejlesztői rendszeremen csináltam egy mérést a Microsoft githubon közzétett[86] benchmarking rendszerével. Ehhez az előbbi méretet demonstráló méréssel ellentétben nem csak 1 oszlop, hanem 1 teljes rekord méretét mértem az adatbázisban (20. ábra), így a keresésben többet számító index területből kiindulva 1000 rekord mérete nagyjából 24 MB. A segédprogramot 120 másodpercig futtattam, végeredményben pedig átlagosan 453,756 milliszekundum olvasási időt kaptam, mely keresés során hozzáadódna a nagyjából 1 másodperchez.

Végezetül, a folyamat része még a kliens oldali felismerés is, mely voltaképp elindítja a folyamatot. Ennek teszteléséhez 100, különböző adathalmazokból összeállított képen futtattam le képfelismerést az adatgeneráláshoz is használt alkalmazás módosított változatát használva. A legrosszabb eredményt egy 11 MB-os képnél

[86] https://github.com/microsoft/diskspd

Implementáció	Adatelérés ideje (ms)
Irodalmi átlag [62]	3000
[56]	1775 (3 személy) közül
[62] saját megoldása	1000
Összesített landmark	1000 + 453 + 600 = 2053

3. táblázat. Az adatelérés sebességének összehasonlítása egyéb megoldásokkal

értem el, mely nagyjából 5 másodperc volt, de az átlag 0.6 másodperc körül volt átlagosan 5.3 MB nagyságú képek mellett. A 3. táblázaton látható, hogy a felismerés, és ezzel együtt az 1 rekordra jutó keresés várható sebessége - habár nem rekordtartó - az átlag alatt van.

A fent leírtak természetesen nem csak az adattároláshoz, de az adattovábbításhoz is kapcsolódnak, hiszen az adattömörítés ott is fontos szempont. Mi több, a közeljövőben az IoT eszközök elterjedése végett ennek jelentősége még inkább felértékelődik: a CISCO-nál 2019 februárjában 2022-re a teljes globábális hálózati forgalom 50%-át (ez becslésük szerint nagyjából 14.6 Mrd) a jelen implementációhoz hasonló okos eszközök fognak majd lebonyolítani [63].

9. Összegzés és továbbfejlesztési javaslatok

A jelen munka a gépi tanuló módszerek keretein belül egy arcfelismerést lehetővé tévő rendszer bemutatásáról szól, mely a szoftvertechnológiában kliens-szerver architektúrának nevezett kompozíciót használja. Mivel a vonatkozó irodalomban nem találni példát ilyen rendszerre, így a már meglévő ismeretek alapján felvázoltam egy ilyen rendszer elvi felépítését és annak kritériumait, majd ezek nyomán prototípus jelleggel implementáltam egy, a követelményeknek megfelelő szoftverrendszert bizonyítva az elgondolás létjogosultságát. Ezután a rendszert összevetve a szakirodalommal megállapítható, hogy - noha előfordulnak tőle bizonyos területen hatékonyabb megoldások - az elvi és gyakorlati előnyöket összegezve valós alternatívája lehet a jelenleg regnáló módszereknek, mi több, az általam közölt megoldás bizonyos aspektusai meg is haladják azokat. Az általam kidolgozott elosztott rendszer újszerűsége az, hogy itt a kliens oldal végzi el a különböző előfeldolgozási folyamatokat, és már csak ezek eredményét küldi el a szerverhez, ami az azonosítást végzi. Mint azt kifejtettem, ez számos előnnyel jár:

- Jelentősen csökken a hálózaton átküldendő adatmennyiség, hiszen nem teljes képeket, hanem csak a landmark pontokat kell átküldeni. Az értékelés során bemutattam, hogy ez valóban egy drasztikus csökkentést jelent, ami mobil eszközök esetében különösen fontos lehet.

- Mivel egy betanított hálózat esetében a predikció már meglehetősen gyors, így a kisebb hálózati terhelés miatt a sebesség is számottevően javulhat.
- Szintén előnyös, hogy a tényleges képek nem jutnak el a szerverre, így egyszerűsödik az érzékeny adatok védelme. A szerver nem tárol képeket (és a hálózaton sem küldjük ezeket át), a kliens pedig nem tárolja az azonosításhoz szükséges adatokat (ezeket nem kell szinkronizálni sem, például).
- A rendszer moduláris felépítésének köszönhetően bármikor cserélhetőek az egyes komponensek, erre szintén mutattam példát a dolgozatban.
- A módszer elkerüli a képek hálózati átvitele miatti túlzott tömörítéséből adódó problémákat is, hiszen a kliens még a tömörítetlen képeken tudja elvégezni az előfeldolgozást.

Megjegyzendő azonban, hogy a jelen keretek között kialakított rendszer valóban csupán egy prototípus, így érdemes lenne egy olyan tesztrendszer kiépítése, melyben valós üzleti keretek között lehetne vizsgálni a módszer működését. Mivel a rendszer egyesíti a központosított és a vastagklienses módszerek előnyeit, így egy olyan „tömegmegoldás" implementálása javasolt, melynek bizonyos elemeit gazdaságtalan lenne a többegységből álló eszközflotta egységeibe építeni, mások hatékony működése szempontjából viszont szükséges egy központi motor. Ennek megfelelően, az általam javasolt tesztrendszer egy intelligens kamerarendszer, mely felhasználói és szerverrel történő kommunikáció nélkül képes emberi arcok detektálására, de önmagán túlmutató funckiókat is be tud tölteni a szerverrel kapcsolatot tartva. Olyan érzékelőkkel egybekötve, mint például infravörös mozgásérzékelő egy olyan rendszer lenne építhető, mely csak mozgásra kapcsol be és képes megállapítani, hogy egy személy elhaladását érzékelte-e és ha igen, akkor azt tovább tudja küldeni elemzésre a szerver felé. Így tesztelhető lenne például, hogy mennyi erőforrás takarítható meg a nem személyek által okozott fals elemzések elmaradásából, vagy hogy alacsonyabb rendszerszintű energiaigénnyel rendelkezik-e egy ilyen megoldás az adattárolást is igénylő megoldásokkal szemben [1]. Noha az implementáció által alkalmazott Android operációs rendszert használó kamerák nem elterjedtek a piacon, a [43] által kidolgozott Raspberry PI 3 alapú intelligens kamera konfiguráció felhasználható erre a célra, mivel az eszköz natív Android támogatással rendelkezik[87]. Ezen implementációban érdemes lenne még a rögzített arcadatokkal elképzelhető legkisebb helyet foglaló megoldásokat is tesztelni, valamint a rendszer teljes „end-to-end" benchmarkolása is egy jövőbeli implementáció feladata lehetne.

Eredményeimet egy angol nyelvű konferencia cikkben foglaltam össze, amelyet az International Conference on Image Processing and Vision Engineering (IMPRO-

[87]https://developer.android.com/things/hardware/raspberrypi

VE2021) nyújtottam be. Végezetül szeretném megköszönni az Óbudai Egyetem GPGPU Programozás Kutatócsoportjának a támogatását

Hivatkozások

[1] S. Mukherjee, Z. Chen, A. Gangopadhyay, and S. Russell, „A secure face recognition system for mobile-devices without the need of decryption," in *Workshop on secure knowledge management*, 2008.

[2] M. Day, „Exploiting facial landmarks for emotion recognition in the wild," *arXiv preprint arXiv:1603.09129*, 2016.

[3] F. Khan, „Facial expression recognition using facial landmark detection and feature extraction via neural networks," *arXiv preprint arXiv:1812.04510*, 2018.

[4] G. K. Chavali, S. K. N. Bhavaraju, T. Adusumilli, and V. Puripanda, „Micro-expression extraction for lie detection using eulerian video (motion and color) magnication," 2014.

[5] P. N. Belhumeur, J. P. Hespanha, and D. J. Kriegman, „Eigenfaces vs. fisherfaces: Recognition using class specific linear projection," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 19, no. 7, pp. 711–720, 1997.

[6] R. Brunelli and T. Poggio, „Face recognition: Features versus templates," *IEEE transactions on pattern analysis and machine intelligence*, vol. 15, no. 10, pp. 1042–1052, 1993.

[7] M. Lades, J. C. Vorbruggen, J. Buhmann, J. Lange, C. Von Der Malsburg, R. P. Wurtz, and W. Konen, „Distortion invariant object recognition in the dynamic link architecture," *IEEE Transactions on computers*, vol. 42, no. 3, pp. 300–311, 1993.

[8] M. Turk and A. Pentland, „Eigenfaces for recognition," *Journal of cognitive neuroscience*, vol. 3, no. 1, pp. 71–86, 1991.

[9] Y. Rodriguez and S. Marcel, „Face authentication using adapted local binary pattern histograms," in *European Conference on Computer Vision*. Springer, 2006, pp. 321–332.

[10] Y. Sun, D. Liang, X. Wang, and X. Tang, „Deepid3: Face recognition with very deep neural networks," *arXiv preprint arXiv:1502.00873*, 2015.

[11] A. K. Jain, J. Mao, and K. Mohiuddin, „Artificial neural networks: A tutorial," *Computer*, no. 3, pp. 31–44, 1996.

[12] R. Iyer, V. Menon, M. Buice, C. Koch, and S. Mihalas, „The influence of synaptic weight distribution on neuronal population dynamics," *PLoS computational biology*, vol. 9, no. 10, p. e1003248, 2013.

[13] K. Tamás, „A mesterséges neurális hálók a jövőkutatás szolgálatában," 2002.

[14] W. S. McCulloch and W. Pitts, „A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943.

[15] K. Hinkelmann. Neural networks. URL: http://didattica.cs.unicam.it/lib/exe/fetch.php?media=didattica:magistrale:kebi:ay_1718:ke-11_neural_networks.pdf (elérés dátuma: 2019. 05. 13.). [Online]. Available: http://didattica.cs.unicam.it/lib/exe/fetch.php?media=didattica:magistrale:kebi:ay_1718:ke-11_neural_networks.pdf

[16] Z. Tang and P. A. Fishwick, „Feedforward neural nets as models for time series forecasting," *ORSA journal on computing*, vol. 5, no. 4, pp. 374–385, 1993.

[17] A. Eleyan and H. Demirel, „Pca and lda based face recognition using feedforward neural network classifier," in *International Workshop on Multimedia Content Representation, Classification and Security*. Springer, 2006, pp. 199–206.

[18] ——, „Face recognition system based on pca and feedforward neural networks," in *International Work-Conference on Artificial Neural Networks*. Springer, 2005, pp. 935–942.

[19] S. Ebrahimi Kahou, V. Michalski, K. Konda, R. Memisevic, and C. Pal, „Recurrent neural networks for emotion recognition in video," in *Proceedings of the 2015 ACM on International Conference on Multimodal Interaction*, 2015, pp. 467–474.

[20] Y. Chen, J. Yang, and J. Qian, „Recurrent neural network for facial landmark detection," *Neurocomputing*, vol. 219, pp. 26–38, 2017.

[21] C. Sammut and G. I. Webb, *Encyclopedia of machine learning and data mining*. Springer Publishing Company, Incorporated, 2017.

[22] M. Liang and X. Hu, „Recurrent convolutional neural network for object recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 3367–3375.

[23] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, „Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[24] F. H. C. Tivive and A. Bouzerdoum, „A new class of convolutional neural networks (siconnets) and their application of face detection," in *Proceedings of the International Joint Conference on Neural Networks, 2003.*, vol. 3. IEEE, 2003, pp. 2157–2162.

[25] A. Ahmed, K. Yu, W. Xu, Y. Gong, and E. Xing, „Training hierarchical feed-forward visual recognition models using transfer learning from pseudo-tasks," in *European Conference on Computer Vision*. Springer, 2008, pp. 69–82.

[26] Y. Sun, X. Wang, and X. Tang, „Hybrid deep learning for face verification," in *Proceedings of the IEEE international conference on computer vision*, 2013, pp. 1489–1496.

[27] Y. Sun, Y. Chen, X. Wang, and X. Tang, „Deep learning face representation by joint identification-verification," in *Advances in neural information processing systems*, 2014, pp. 1988–1996.

[28] P. Corcoran and C. Iancu, „Hidden markov models in automatic face recognition-a review," in *Reviews, Refinements and New Ideas in Face Recognition*. IntechOpen, 2011.

[29] A. V. Nefian, „Embedded bayesian networks for face recognition," in *Proceedings. IEEE International Conference on Multimedia and Expo*, vol. 2. IEEE, 2002, pp. 133–136.

[30] A. Dhall, R. Goecke, J. Joshi, M. Wagner, and T. Gedeon, „Emotion recognition in the wild challenge 2013," in *Proceedings of the 15th ACM on International conference on multimodal interaction*, 2013, pp. 509–516.

[31] S. R. Khanal, J. Barroso, N. Lopes, J. Sampaio, and V. Filipe, „Performance analysis of microsoft's and google's emotion recognition api using pose-invariant faces," in *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*, 2018, pp. 172–178.

[32] H. Hosseini, B. Xiao, and R. Poovendran, „Google's cloud vision api is not robust to noise," in *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2017, pp. 101–105.

[33] K. Bhuvaneshwari, A. Abirami, and N. Sripriya, „Face recognition using pca," *International Journal of Engineering and Computer Science*, vol. 6, no. 4, 2017.

[34] D. E. King, „Dlib-ml: A machine learning toolkit," *Journal of Machine Learning Research*, vol. 10, pp. 1755–1758, 2009.

[35] S. Viraktamath, M. Katti, A. Khatawkar, and P. Kulkarni, „Face detection and tracking using opencv," *The SIJ Transactions on Computer Networks & Communication Engineering (CNCE)*, vol. 1, no. 3, pp. 45–50, 2013.

[36] P. Viola and M. Jones, „Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, vol. 1. IEEE, 2001, pp. I–I.

[37] F. Schroff, D. Kalenichenko, and J. Philbin, „Facenet: A unified embedding for face recognition and clustering," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 815–823.

[38] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, „Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[39] R. Collobert, S. Bengio, and J. Mariéthoz, „Torch: a modular machine learning software library," Idiap, Tech. Rep., 2002.

[40] D. Gannon, „Cntk revisited. a new deep learning toolkit release from microsoft."

[41] S. M. Lewandowski, „Frameworks for component-based client/server computing," *ACM Computing Surveys (CSUR)*, vol. 30, no. 1, pp. 3–27, 1998.

[42] M. Zur Muehlen, J. V. Nickerson, and K. D. Swenson, „Developing web services choreography standards—the case of rest vs. soap," *Decision Support Systems*, vol. 40, no. 1, pp. 9–29, 2005.

[43] I. Aydin and N. A. Othman, „A new iot combined face detection of people by using computer vision for security application," in *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*. IEEE, 2017, pp. 1–6.

[44] S. Karnila, S. Irianto, and R. Kurniawan, „Face recognition using content based image retrieval for intelligent security," *International Journal of Advanced Engineering Research and Science*, vol. 6, no. 1, 2019.

[45] I. Sommerville, „Software engineering 9th edition," *ISBN-10*, vol. 137035152, p. 18, 2011.

[46] Y. Wu, T. Hassner, K. Kim, G. Medioni, and P. Natarajan, „Facial landmark detection with tweaked convolutional neural networks," *IEEE transactions on pattern analysis and machine intelligence*, vol. 40, no. 12, pp. 3067–3074, 2017.

[47] R. Padilla, C. Costa Filho, and M. Costa, „Evaluation of haar cascade classifiers designed for face detection," *World Academy of Science, Engineering and Technology*, vol. 64, pp. 362–365, 2012.

[48] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, „Efficient backprop," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48.

[49] G. Klambauer, T. Unterthiner, A. Mayr, and S. Hochreiter, „Self-normalizing neural networks," in *Advances in neural information processing systems*, 2017, pp. 971–980.

[50] L. Bottou, „Stochastic gradient descent tricks," in *Neural networks: Tricks of the trade*. Springer, 2012, pp. 421–436.

[51] T. Dozat, „Incorporating nesterov momentum into adam," 2016.

[52] T. Upadhyay and D. Roy, „Lie detection based on human facial gestures."

[53] J. Zhao, M. Mathieu, and Y. LeCun, „Energy-based generative adversarial network," *arXiv preprint arXiv:1609.03126*, 2016.

[54] X. Yi, E. Walia, and P. Babyn, „Generative adversarial network in medical imaging: A review," *Medical image analysis*, vol. 58, p. 101552, 2019.

[55] S. A. Zahorian, P. Silsbee, and X. Wang, „Phone classification with segmental features and a binary-pair partitioned neural network classifier," in *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, vol. 2. IEEE, 1997, pp. 1011–1014.

[56] E. Weinstein, P. Ho, B. Heisele, T. Poggio, K. Steele, and A. Agarwal, „Handheld face identification technology in a pervasive computing environment," in *Short Paper Proceedings, Pervasive 2002*, 2002, pp. 48–54.

[57] T. J. Hazen, E. Weinstein, and A. Park, „Towards robust person recognition on handheld devices using face and speaker identification technologies," in *Proceedings of the 5th international conference on Multimodal interfaces*, 2003, pp. 289–292.

[58] I. Aydin and N. A. Othman, „A new iot combined face detection of people by using computer vision for security application," in *2017 International Artificial Intelligence and Data Processing Symposium (IDAP)*. IEEE, 2017, pp. 1–6.

[59] Y. Liu, D. Zhang, G. Lu, and W.-Y. Ma, „A survey of content-based image retrieval with high-level semantics," *Pattern recognition*, vol. 40, no. 1, pp. 262–282, 2007.

[60] A. W. Smeulders, M. Worring, S. Santini, A. Gupta, and R. Jain, „Content-based image retrieval at the end of the early years," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 22, no. 12, pp. 1349–1380, 2000.

[61] A. J. I. Barbhuiya, T. A. Laskar, and K. Hemachandran, „An approach for color image compression of jpeg and png images using dct and dwt," in *2014 International Conference on Computational Intelligence and Communication Networks*. IEEE, 2014, pp. 129–133.

[62] A. R. Kumar and D. Saravanan, „Content based image retrieval using color histogram," *International journal of computer science and information technologies*, vol. 4, no. 2, pp. 242–245, 2013.

[63] G. Forecast, „Cisco visual networking index: global mobile data traffic forecast update, 2017–2022," *Update*, vol. 2017, p. 2022, 2019.