Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

Biometrikus felhasználó azonosítás

Diplomaterv

Készítette
Boér Lehel

Konzulens
dr. Zainkó Csaba

2019. november 2.

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott *Boér Lehel*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2019. november 2.

Boér Lehel
hallgató

Kivonat

Jelen dokumentum egy diplomaterv sablon, amely formai keretet ad a BME Villamosmérnöki és Informatikai Karán végző hallgatók által elkészítendő szakdolgozatnak és diplomatervnek. A sablon használata opcionális. Ez a sablon LaTeX alapú, a *TeXLive* TeX-implementációval és a PDF-LaTeX fordítóval működőképes.

Abstract

This document is a LaTeX-based skeleton for BSc/MSc theses of students at the Electrical Engineering and Informatics Faculty, Budapest University of Technology and Economics. The usage of this skeleton is optional. It has been tested with the *TeXLive* TeX implementation, and it requires the PDF-LaTeX compiler.

1. fejezet

Háttérismeretek

1.1. Biometrikus azonosítás

A biometria az emberek fizikai jellemzőinek mérésével és elemzésével foglalkozik. Alkalmazását tekintve három területet különböztetünk meg:

- Felhasználó ellenőrzés: Az azonosító rendszer a biometrikus adatot egy, korábban vetthez hasonlítja. Ez alapján dönt, hogy a felhasználó hozzáférhet-e a kívánt erőforráshoz. Ilyen egy ujjlenyomat-olvasóval ellátott mobiltelefonon a képernyőzár feloldása. A felhasználó ellenőrzés arra ad választ, hogy az illető az-e akinek mondja magát.
- Felhasználó azonosítás: Az azonosító rendszer a biometrikus adatot több korábban vett mintához hasonlítja és arra ad választ, hogy ki a felhasználó; azaz beletartozike a korábban eltárolt biometrikus adatokból álló csoportba vagy nem. Ilyen lehet például egy ujjlenyomat-leolvasóval ellátott beléptetőrendszer cégek esetében.
- Duplikátum detektálás: Annak ellenőrzése, hogy egy felhasználó egynél többször szerepel-e egy adatbázisban. Csalások, például szociális támogatást többször igénylők kiszűrésére használják.

Az első biometrikus azonosítási eljárás az ujjlenyomatvételen alapuló személyiségazonosítás volt, amely a modern kriminalisztika világában terjedt el, de manapság már megtalálható okostelefonokban, biometrikus beléptetőrendszerekben is.

A biometrikus azonosítást az ún. biometrikus azonosító rendszer végzi el. A folyamat során a biometrikus azonosító rendszer mintát vesz az azonosítandó egyén egy vagy több előre meghatározott fizikai jellemzőjéről, és ezekről digitális lenyomatokat képez. Az első, regisztrációs fázisban a biometrikus minta lenyomatát a rendszer egy adatbázisban eltárolja, majd később az azonosítás során az aktuális mintát összeveti a korábban rögzítettel és dönt az egyezésről. Ahhoz, hogy az ember egy fizikai jellemzőjét biometrikus adatként használhassuk, a következő elvárásokat támasztjuk vele szemben:

- Általánosság: A biometrikus adattal minden egyénnek rendelkeznie kell.
- Egyediség: A biometrikus adatnak egyedinek kell lennie a releváns populáción belül.
- Állandóság: A biometrikus adat nem, vagy csak keveset változzon az idő eltetével.
- Mérhetőség: Az biometrikus adat az egyén részéről legyen könnyen mérhető testi adottság.

- Teljesítmény: A biometrikus azonosító rendszerek teljesítménye: gyorsaság, pontosság, technológia.
- Elfogadottság: A releváns populáción belül a mérési eljárás mennyire elfogadott (emberi méltóság megőrzése).
- Biztonság: Mennyire nehéz utánozni, hamisítani a biometrikus adatot?

A biometrikus adat lehet fiziológiai (DNS, arc, ujjlenyomat, írisz) vagy viselkedési (hang, írás, gesztusok). Mivel ezek az adatok statisztikai jellegűek, megbízhatóságuk változó. Minél több adat van egy mintában, annál egyedibb, és minél nagyobb a releváns populáció (eltárolt minták összessége), annál valószínűbb, hogy találunk két hasonló mintát. Ennek elkerülésére manapság terjednek a multimódusú biometrikus azonosító rendszerek, amelyek több biometrikus adatot felhasználva végzik ez az ellenőrzés, azonosítás és duplikátum detektálás feladatát.

1.2. Beszélőfelismerés

Az emberi kommunikáció során fontos feladat a beszélő partner felismerése. A telekommunikációs technológia fejlődése miatt elterjedt a telefonon vagy interneten történő hangalapú kommunikáció; a telefonos felhasználófelismerés mint biometrikus azonosítási módszer megjelent már banki alkalmazásokban, call centerekben és az elektronikus kereskedelemben is (mobiltelefonos vásárlás). Az elektronikus kommunikáció során sokszor csak a beszélő hangjára hagyatkozhatunk, az alapján ismerhetjük fel az illetőt. A beszélőfelismerést háromféle módon végezhetjük:

- Naiv beszélőfelismerés: Az emberi, naiv beszélőfelismerés során az ismerős hangokat meglepően nagy pontossággal ismerjük fel.
- Törvényszéki beszélőfelismerés: A törvényszéki szakértői vizsgálat eredménye.
- Automatikus beszélőfelismerés: A beszélőfelismerést számítógépes rendszer végzi.

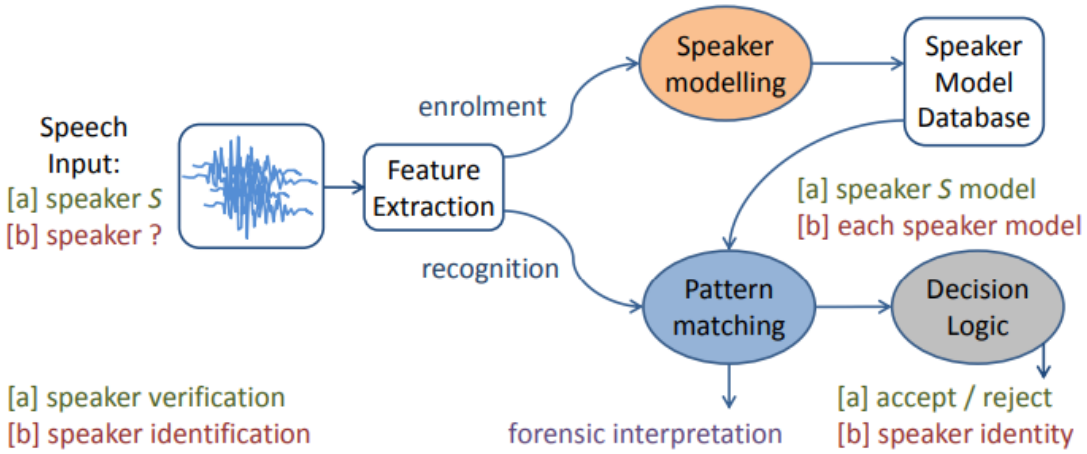
A beszélőfelismerés alatt három szűkebb fogalmat értünk. Ha a folyamat során az ismeretlen beszélőről azt ellenőrizzük, hogy az-e akinek állítja magát beszélő ellenőrzésről van szó. Beszélő szegmentáláskor a hangmintát homogén csoportokra bontjuk a beszélő személye alapján. Végül beszélő azonosításról beszélünk, ha az illető hangját rögzített hangok egy csoportjával vetjük össze és azt szeretnénk eldönteni, hogy melyikhez hasonlít a legjobban. Utóbbi felveti a kérdést, hogy mi történik ha a beszélő nem tagja a csoportnak.

Emiatt megkülönböztetjük a nyitott és zárt halmazú beszélőazonosítást. Utóbbi esetén csak olyan beszélőket ismerünk fel, akikről van hangminta az adatbázisban, míg az előbbinél ismeretlen beszélők is megjelenhetnek, így ezt is kezelni kell.

A beszélőfelismerés továbbá lehet szöveg-függő és szöveg-független attól függően, hogy a felismerő rendszer egy előre meghatározott mondatot vár, vagy bármilyen hangminta alapján működik.

1.3. Az automatikus beszélőfelismerés története

Az automatikus beszélőfelismerést egy számítógépes program végzi emberi beavatkozás nélkül. Az első automatikus beszédfelismerő rendszert a Texas Instruments fejlesztetése volt és 1977-ben publikálták. A rendszer szövegfüggő beszélőellenőrzésre volt képes és az évek során a téves elutasítási és elfogadási rátája 1% alatt maradt. A hetvenes évek óta a a



1.1. ábra. Automatikus beszélőfelismerő rendszer architektúrája.

beszélőazonosító és ellenőrző rendszerek rengeteget fejlődtek kezdve a vektor kvantálástól kezdve a GMM modelleken át a mély neurális hálókig.

Az automatikus beszélőfelismerő általános működését az alábbi (!) ábra szemlélteti. A beszélő-ellenőrzés és beszélőazonosítás során is az első lépés rögzített *jellemzők kinyerése*. Ezután az első, *tanító fázisban* referencia modelleket készítünk az egyes beszélőkhöz a jellemzők alapján, amelyeket eltároljuk a beszélő adatbázisban. Beszélő-ellenőrzés esetén ebben a fázisban egy küszöbérték is meghatározásra kerül. *Teszt fázisban* a rendszer kinyeri ugyanazokat a jellemzőket az aktuális hangmintából, majd *jellemző összehasonlítás* történik.

1. beszélő-ellenőrzés esetén megkeresi az ellenőrizendő személyhez tartozó modellt az adatbázisban és összehasonlítja az aktuális jellemzőkkel. Ha az eredmény a küszöbszint felett van, a rendszer egyezést mutat.

2. beszélőazonosítás esetén az aktuális jellemzőket az összes modellel összehasonlítja, majd a legjobb egyezés mellett dönt.

1.3.1. Jellemző kinyerés

A jellemző kinyerés célja a dimenzió csökkentése és a beszélőspecifikus információk kinyerése. Mivel a beszéd komplex jel, a beszélő azonosítása szempontjából felesleges információkat is hordoz. Ilyen például a környezet és a csatorna zaja. A kinyert jellemzőket a hangminta terjedelme alapján osztályozzuk. *Rövidtávú jellemzők* a 20-30 ms-os keretekből kinyert mel-frekvenciás és lineáris prediktív kepsztrális együtthatók (MFCC és LPCC). A *prozódikus jellemzők* kinyerése 100 ms-os terjedelemben történik és a beszéd ritmusát, a hangmagasságot és a sebességet jellemzik. A hosszútávú jellemzőket a jel akár perc hosszú kereteiből nyerjük ki. Ezek képesek reprezentálni a beszélő akcentusát illetve a szavak szemantikáját és az idiolektust.

A beszélőfelismerő rendszerek teljesítményét javította, ha a jellemzőket csak a hangminta azon részeiből nyerték ki, amikben beszéd is volt. Erre alkalmazott technika a *Voice Activation Detection* (*VAD*).

1.3.2. Jellemző normalizálás

Jellemzőkinyerés során próbáljuk kiszűrni a beszélő szempontjából értékes részéket, ugyanakkor nincs tökéletes jellemző, amely ne változna a környezet hatására. Ezt a változást segítik minimalizálni a normalizálási módszerek.

1.3.3. Beszélő modellek

Kezdetben a vektor kvantálás volt az elterjed modellezési módszer, amit később a *Gaussian Mixture Model* (*GMM*) váltott fel. A GMM egy adathalmazt több normális eloszlás keverékeként ír le és képes nem felügyelt módon klaszterezni az adatokat. Egy beszélőhöz egy valószínűségi sűrűségfüggvényt rendel, amely különböző pontokban kiértékelve (például teszt fázisban a beszélőtől kinyert jellemzők) egy valószínűséget ad a két beszélő hasonlóságára.

A GMM megközelítés főleg beszélőazonosításra alkalmas. Beszélő-ellenőrzéshez szükség volt egy másik modellre is, ami képes leírni minden más beszélőt az ellenőrizendőn kívül. Erre adott megoldást az *Universal Background Model* (*UBM*). Később jobb teljesítményt értek el, ha a teszt fázisban a beszélőkkel először UBM modelleket tanítottak és ezekből származtattak GMM-eket. Ezt nevezik GMM-UBM módszernek.

Mivel a tanító és teszt hangminták eltérő hosszúságúak lehetnek, szükség volt egy fix hosszúságú reprezentációra, ezt oldották meg a GMM szupervektorok, amelyeket az akkori megközelítés szerint szupport-vektor gépekkel vagy faktoranalízissel használtak.

Az utóbbi két módszer előnyeit kombinálva megszületett az $i-vektorok$, amelyet követve eljutunk a mai state-of-the-art módszerhez, a mély neurális hálózatokhoz (DNN).

1.4. Korábbi eredmények

Szerző (év)	**Szervezet**	**Adatbázis**	**Módszer**	**Jellemzők**	**Hang típusa**	**Pontosság**
Douglas A. Reynolds (1995)	Lincoln Laboratory	49	MFCC	rövid kifejezések	telefon	96.8 %
Rabah W. (2004)	King Abdulaziz University	20	SVD-alapú algoritmus	LPC/Cepstral	iroda	94 %
Yang Shao (2008)	Ohio State University	34	GFCCs	hallási jellemzők	telefon	~99.33 %
P. Krishnamoorthy (2011)	TIMIT	100	GMM-UBM	MFCC	labor	80 %
Alfredo Maesa (2012)	Voxforge.org	250	MFCC	spektrális jellemzők	beszéd-adatbázis	$>$ 96 %
Sharada V. Chougule (2015)	Finolex Academy of Management and Technology	97	NDSF	spektrális	labor	~98-100 %

1.1. táblázat. Korábbi eredmények szövegfüggetlen beszélőazonosítás terén.

2. fejezet

Beszélőazonosító rendszerek napjainkban

A fejezet bemutatja a tanítás során használt beszédadatbázisokat, két neurális hálózat alapú, zárt-halmazú, automatikus beszélőfelismerő rendszert; a *WaveNet classifiert* és a *SincNetet* illetve az ezekkel elért eredményeket.

2.1. Beszédadatbázisok

2.1.1. TIMIT

A TIMIT beszédkorpuszt automatikus beszédfelismerő rendszerek fejlesztéséhez tervezték. 630 beszélőtől tartalmaz mintákat amerikai angol nyelven a 8 legelterjedtebb nyelvjárásban. A TIMIT archívum tartalmaz egy TRAIN és egy TEST mappát, ezek tanításhoz és teszteléshez valók. Ezeken belül további, a dialektusok sorszámával (DR1, ..., DR8), azon belül a beszélő azonosítójával elnevezett könyvtárak találhatók. Egy beszélőhöz 10 db beszédminta tartozik 16 kHz-es NIST SPHERE fájlok formájában.

Dialektus regió (DR)	**Férfi**	**Nő**	**Összesen**
1	31 (63%)	18 (27%)	49 (8%)
2	71 (70%)	31 (30%)	102 (16%)
3	79 (67%)	23 (23%)	102 (16%)
4	69 (69%)	31 (31%)	100 (16%)
5	62 (63%)	36 (37%)	98 (16%)
6	30 (65%)	16 (35%)	46 (7%)
7	74 (74%)	26 (26%)	100 (16%)
8	22 (67%)	11 (33%)	33 (5%)

2.1. táblázat. A beszélők eloszlása dialektusok szerint.

A dialektus régiók a következők:

- DR1: New England
- DR2: Northern
- DR3: North Midland
- DR4: South Midland
- DR5: Southern
- DR6: New York City
- DR7: Western
- DR8: Army Brat (moved around)

A NIST SPHERE formátum az elején definiál egy fix hosszú fejlécet, amit a hang bináris kódolása követ. Erre figyelni kell a hangfájlok beolvasásánál, Python esetében nem minden hangfeldolgozó könyvtár támogatja. Ilyen esetben kézzel el kell távolítani a fejlécet a fájlok elejéről. A hangfájlokhoz tartozik egy szöveges dokumentum ami az elhangzott szöveget és annak a wav fájlbeli helyét tartalmazza. Továbbá egy WRD fájl írja le a szavakat és egy PHN kiterjesztésű a fonémákat, illetve azok időbeni elhelyezkedését a wav fájlban.

A hangfájlokhoz tartozó egyéb fájlok beszédfelismerés szempontjából fontosak. Mivel én a beszédkorpuszt beszélőfelismerésre használtam, csak a hangfájlokra volt szükségem. A TEST mappában - mivel a TIMIT-et alapvetően beszédfelismeréshez tervezték - teljesen különböző beszélők vannak a TRAIN mappához képest, ezért a TRAIN mappabeli beszélőket kell felosztani tanításhoz és teszteléshez.

2.1.2. CMU Arctic

A CMU Arctic beszédadatbázist beszédszintézis kutatásokhoz tervezték. 18 adathalmazt tartalmaz 18 különböző embertől más-más akcentussal, angol nyelven. Egy emberhez több száz beszédminta tartozik wav fájlok formájában.

2.1.3. Adatok előfeldolgozása

Az előfeldolgozó szkript a TIMIT adatbázis esetében a hangmintákról eltávolítja a NIST Sphere fejlécet és a mondat előtti és utáni szüneteket. Ezután normalizálja a hangmintákat. A CMU Arctic esetében a hangminták alapból normalizálva vannak, ezért csak azonos méretűre kell vágni őket az egységes bemeneti dimenziók érdekében (ahogy a TIMIT esetében is).

2.2. Mérési elrendezés

Google Colab...

2.3. WaveNet classifier

A *WaveNet classifier* egy módosított WaveNet architektúra beszélőidentifikációhoz.

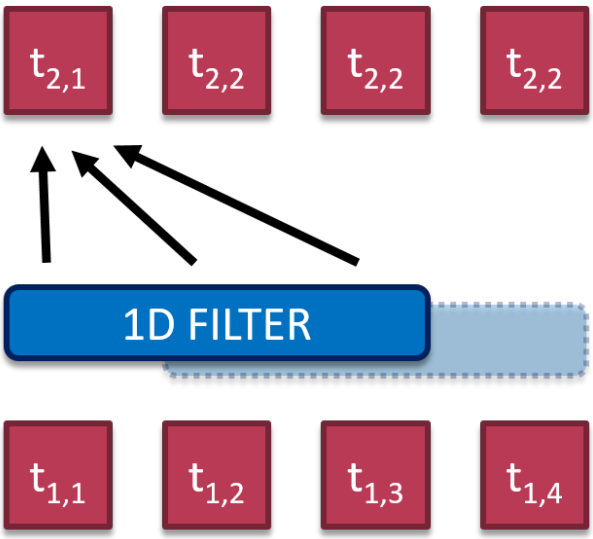
2.3.1. WaveNet

A WaveNet egy mély neurális hálózat audio hullámformák generálásához, amelyet a Google DeepMind publikált 2016-ban. Az ötletet az akkori felfedezések adták neurális autoregresszív generatív modellezés terén, amelyeket komplex eloszlások, például képek modellezésére használtak (van den Oord et al., 2016a;b). Ezt felhasználva audio hullámformák generálásában új eredményeket értek el.

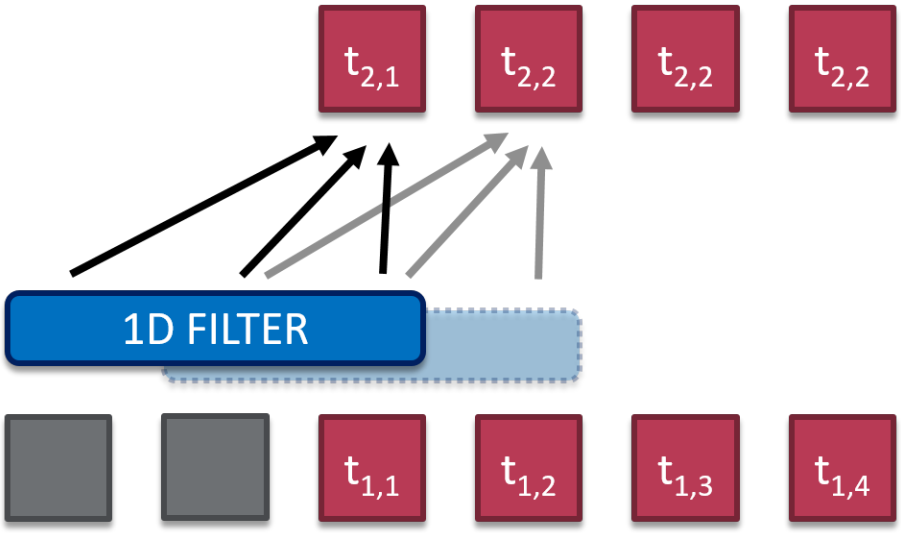
- Képes olyan természetes hangzású beszéd hullámformák generálására, amit korábban parametrikus vagy konkatenatív beszédszintézissel sosem értek el.
- Az audio hullámformák generáláshoz szükséges nagy receptív mezőt hatékonyan, nyújtott kauzális konvolúciókkal implementálja.
- Ha a modellt a beszélők identitásával tanítják, képes új hangok generálására.
- A zene generálás és a beszédfelismerés terén is ígéretesnek bizonyult.

2.3.1.1. Nyújtott kauzális konvolúció

A modell autoregresszív, vagyis a kimenete korábbi időpillanatokban felvett értékeitől függ. Ez azért fontos, mert a WaveNet egy generatív modell. A generált hullámforma t-edik időpillanatbeli értéke nem függhet jövőbeli értékektől. A kauzalitás legegyszerűbb implementációja ha legalább a *kernel-1* méretű paddinget adunk a konvolúcióhoz.



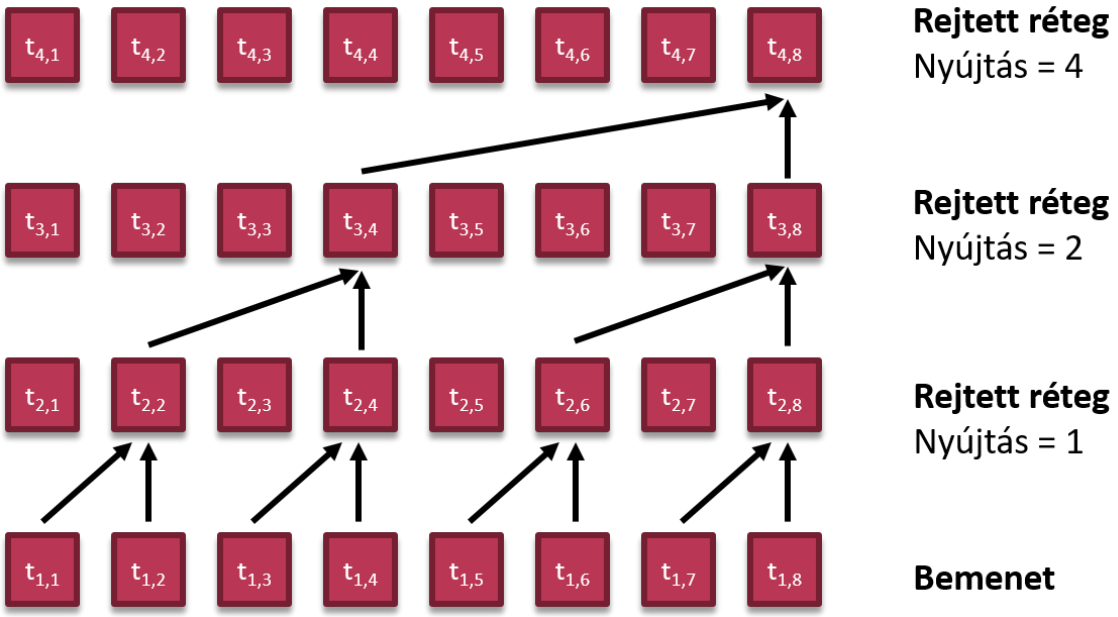
2.1. ábra. 1D nem kauzális konvolúció.



2.2. ábra. 1D kauzális konvolúció paddinggel.

A 2.1 ábra egy nem kauzális 1D konvolúciót mutat. A $t_{i,j}$ az i-edik rétegbeli j-edik neuron. Látható, hogy a $t_{2,1}$ jövőbeli időpillanatokból kap értékeket a szűrőn keresztül. Ennek megoldása a szűrő eltolása padding segítségéve. Ezt szemlélteti a **??** ábra, ahol az egyes neuronok csak korábbi időpillanatokból kapnak értékeket.

A beszéd generálásánál a t-edik időpillanatban a hullámforma értéke a korábbi adatoktól függ. Ahhoz, hogy magas frekvenciájú, pl. 16 kHz frekvencián mintavételezett hangadattal tanítsuk a hálózatot nagy receptív mezőre van szükség. A receptív mező az a szélesség, amit a szűrő lát a bemenetből. 16 kHz esetén egy másodpercnyi jelet 16000 szám reprezentál. Ahhoz, hogy a hálózat helyesen jósolja meg a következő generált értéket, a hosszú távú dependenciákat figyelembe kell vennie, tehát a receptív mező méretét elég nagyra kell megválasztani. A probléma ekkora mezők esetén, hogy sok konvolúciós réteget igényelnek (egy korábbi időpillanatbeli adat plusz egy konvolúciós réteget igényel), ami növeli a számítási komplexitást. A nyújtott konvolúciók erre adnak hatékony megoldást. A filter meghatározott távolságokkal kihagy valamennyi inputot, majd figyelembe vesz egyet. Egymás utáni rétegekben a nyújtási tényezőt exponenciálisan növelve a receptív mező is exponenciálisan fog nőni.



2.3. ábra. 1D nyújtott kauzális konvolúciós rétegek.

2.3.1.2. SoftMax eloszlás

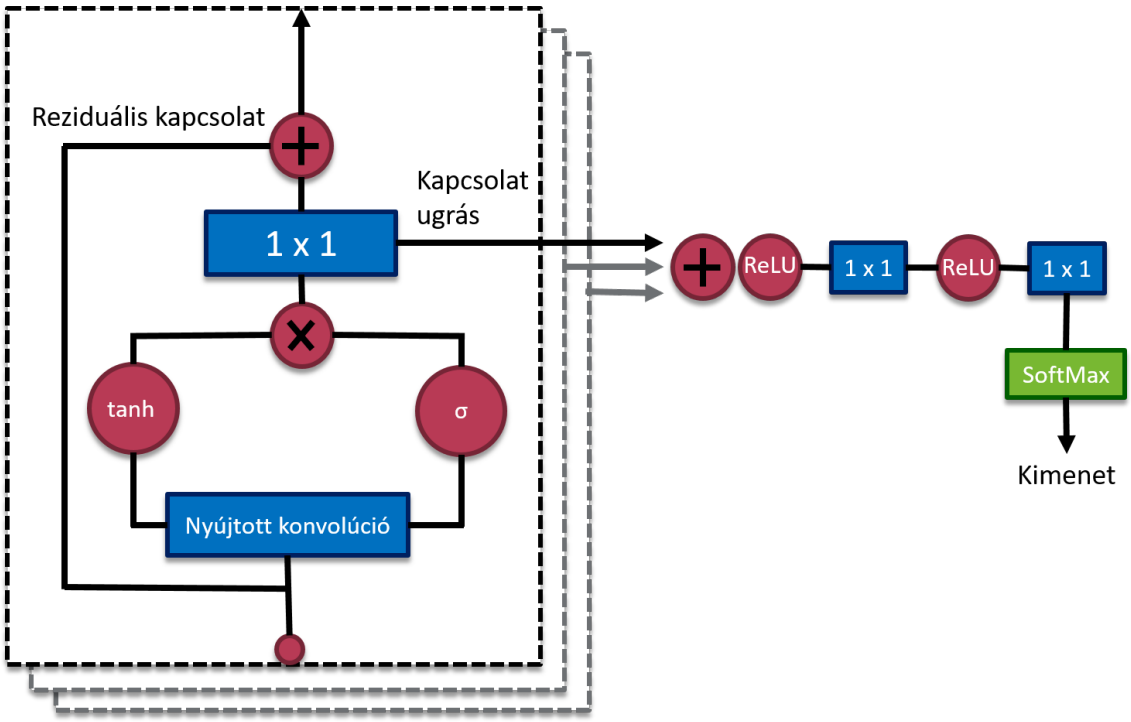
A WaveNet SoftMax réteget használ a $p(x_t|x_1, \ldots, x_t)$ feltételes valószínűségi eloszlás modellezésére. Az audio jeleket általában 16 bites egészekkel kódolják, amelyek 65536 értéket vehetnek fel. Ebben az esetben a SoftMax rétegnek 65536 valószínűséget kell kimenetként adnia, melyek összege 1. A μ-law kvantálást alkalmazva a beszédjel 256 biten kódolható és később az inverz transzformációval jó minőségben visszaállítható.

Az emberi hallás sokkal érzékenyebb alacsony amplitúdójú hangok kvantálási zajára, mint a magasabbakéra. Erre alapozva a *mu*-law kvantáló a jelet egy logaritmikus függvénnyel kvantálja úgy, hogy az alacsonyabb amplitúdójú jelek nagyobb felbontással (több bittel), a magasabbak pedig kisebbel lesznek kódolva. Ez növeli a SoftMax réteg hatékonyságát is, mert nagyobbak lesznek a valószínűségek közötti különbségek.

A tanítást a WaveNet klasszifikációs problémaként kezeli. A bemeneteket OneHot kódolással adjuk meg, a SoftMax réteg pedig az így kódolt egészekre ad valószínűségi eloszlást.

2.3.1.3. Reziduális blokkok

A WaveNet architektúra egymáshoz csatolt reziduális blokkokból és ún. kapcsolat-ugrásokból (skip-connection) épül föl. A reziduális hálózatok előnye, hogy orvosolják az elenyésző gradiens problémát, így sokkal mélyebb hálózat építhető.



2.4. ábra. WaveNet architektúra.

Egy reziduális blokk bemenete egy 2x1-es konvolúciós rétegen megy keresztül. Balra egy *tanh*, jobbra egy szigmoid aktivációs függvényen haladnak át, majd elemenkénti szorzás és 1x1 konvolúció után egyrészt átugorja a reziduális kapcsolatot, illetve azzal együtt bemenetként szolgál a következő reziduális egységnek. Az 1x1 konvolúciós rétegek a dimenzionalitás változtatására szolgál. Külön 1x1 konvolúciós szűrők skálázzák a kimenetet a következő reziduális blokk bemenetére, és a kapcsolat-ugrásokhoz.

2.3.2. Módosított WaveNet architektúra

A módosított WaveNet architektúra segítségével a WaveNetet beszélőfelismerésre használhatjuk.

```
from WaveNetClassifier import WaveNetClassifier

wnc = WaveNetClassifier((96000,), (10,), kernel_size = 2, dilation_depth = 9,
                        n_filters = 40, task = 'classification')

wnc.fit(X_train, y_train, validation_data = (X_val, y_val), epochs = 100,
        batch_size = 32, optimizer='adam', save=True, save_dir='./')
```

```
y_pred = wnc.predict(X_test)
```

A WaveNetClassifier objektum paraméterei:

- *input_shape*: Bemeneti dimenziók tuple formájában. Például ha a bemenet egy 6 s hosszú hullámforma 16 kHz-en mintavételezve, a bemeneti dimenziók (96000,)
- *output_shape*: Kimeneti dimenziók tuple formájában. Például ha 100 osztály szerint klasszifikálunk, a kimeneti dimenziókból képzett tuple (100,).
- *kernel_size*: A konvolúciós filter/kernel mérete a reziduális blokkokban.
- *dilation_depth*: A reziduális blokkok száma.
- *n_filters*: A konvolúciós filterek száma a reziduális blokkokban.
- *task*: Klasszifikáció vagy regresszió.
- *regression_range*: A regresszió céltartománya lista vagy tuple formátumban.
- *load*: Előző WaveNetClassifier betöltése (bool).
- *load_dir*: A betölteni kívánt modell könyvtára.

2.3.3. Eredmények

A WaveNet classifiert mindkét beszédadatbázison teszteltem. A TIMIT beszédkorpusszal csak kevés beszélő esetén ért el jó eredményt, több mint 20 beszélő esetén a modell nem tanult. Ennek valószínűsített oka, hogy a TIMIT adatbázis beszélőnként 10 hangmintát tartalmaz.

Beszélők száma	18
Minta/beszélő	100
Minta össz.	1800
Minta hossza	4000
Epochok száma	43
Hiba	0.0013
Pontosság	1.0

2.2. táblázat. Paraméterek CMU Arctic adatbázissal.

!!! teszthalmaz mérete !!!
Tanítás után a teszt adathalmazon a hálózat 96.799 %-os pontosságot ért el.

2.4. SincNet

2.4.1. Eredmények

3. fejezet

Meta learning

Ebben a fejezetben ismertetem a metatanítás fogalmát, a megközelítéseket. Megmutatom, hogy mit old meg a few-shot learning és ez hogyan felhasználható nyílt halmazú beszélőfelismerő rendszerekben.

> "— Miért van szüksége a neurális hálózatoknak sok tanítómintára a jó teljesítéshez? Egy két éves gyerek miután látott pár autót képes felismerni azt. — Egy ekkora gyereknek volt két éve tapasztalatokat szerezni olyan dolgokról, amik nem autók voltak. Biztos vagyok benne, hogy ez fontos szerepet játszott a dologban."
>
> (Quora)

Az emberek képesek hasznosítani a korábban szerzett tudást. Ha valaki megtanult biciklizni, utána sokkal könnyebben motorbiciklire ül. Felismerünk dolgokat úgy, hogy előtte csak néhányszor láttuk élőben vagy csak képen láttuk.

Az emberekkel szemben a neurális hálózatokkal két probléma van.

1. Nem tanulnak effektíven, sok tanítómintát igényelnek egy feladat megtanulásához.
2. Nem hasznosítják a korábban, más feladatok által megszerzett tudást.

Metatanítás alatt olyan tanító módszerek összességét értjük, amelyek felhasználják a korábban szerzett tudást és hasznosítják azt a későbbi feladatok során. A metatanuló modellek általánosítják a tapasztalataikat és könnyen adaptálódnak új feladatokhoz. A könnyű adaptáció alatt kevés tanítómintával végzett tanítást értünk, más néven finomhangolást.

3.1. Few-shot learning

A neurális hálózatok rengeteg tanítómintát igényelnek. A kevés tanítómintával tanított hálózatok túltanulnak; a tanítóhalmazon jó eredményt mutatnak, de csökken az általánosító képességük, így a teszthalmazon jelentősen rosszabb eredményt érnek el. Ugyanakkor ha sok tanítómintával tanítjuk a hálózatokat, az adott feladatokra már jól általánosítanak, de magukra a feladatokra tanítjuk túl őket. További hasonló feladatokra nem tudnak általánosítani.

A few-shot learning azzal a problémával foglalkozik, hogy hogyan építhetünk olyan modelleket, amelyek képesek új feladatokat gyorsan megtanulni. A modellt sokféle feladatra tanítjuk, de minden feladathoz kevés tanítóminta tartozik. A tanítás után új feladat esetén a modell kevés tanítómintával képes jól megtanulni azt.

A few-shot learningre való képességet az n-way k-shot feladattal szokták mérni.

1. A modell kap egy eddig nem látott osztályból egy tesztmintát.
2. Kap továbbá egy ún. segéd adathalmazt, ami az összes k eddig nem látott osztályból n mintát tartalmaz.
3. Az algoritmus el kell döntse, hogy melyik osztályba tartozik a tesztminta a segédhalmazból.

A beszélőfelismerő rendszereket tekintve nagyon fontos, hogy a neurális hálózat rugalmas legyen a tanulás szempontjából. Tegyük fel, hogy egy cég telefonos, szövegfüggetlen, nyílt halmazú beszélőfelismerést szeretne abból a célból, hogy a betelefonáló kliensek adatait rövid hangminta után a rendszer automatikusan kilistázza az operátornak. Egy nagyobb telefontársaság esetén ez több ezer beszélőt is jelenthet. Hagyományos neurális hálózatokkal minden egyes klienstől több perces hangmintára lenne szükség, hogy ne tanítsuk túl a modellt.

A másik, még nagyobb probléma, hogy új, vagy távozó kliens esetén újra kell tanítani a teljes hálózatot. A few-shot learning tökéletesen megoldja mindkét problémát. Kevés tanítómintával – regisztrációkor pár kérdés kliensenként – működik a rendszer és rugalmas is tanítás szempontjából; új kliens esetén hozzáadjuk a segédhalmazhoz a hangmintáját, távozáskor pedig eltávolítjuk.

3.2. Metrikus metatanítás

Metrikus metatanítás során a modell a tanítóminták halmazán egy távolságfüggvényt tanul meg.

$$P_\theta(y|x, S) = \sum_{(x_i,y_i)\in S} k_\theta(x, x_i)y_i \tag{3.1}$$

A valószínűsége annak, hogy a θ modellparaméterekkel az x minta S segédhalmaz mellett az y osztályba tartozik egyenlő a segédhalmazba tartozó címkék súlyozott összegével. A súlyt a k_θ ún. kernel függvény számolja ki. A kernel függvény az a távolságfüggvény, amit a modell megtanul.

A megközelítésre több implementáció is létezik, mint például a Matching Network, Prototypical Network, Relation Network és a konvolúciós sziámi hálózat. Ezek közül az utolsót fogom részletesen bemutatni.

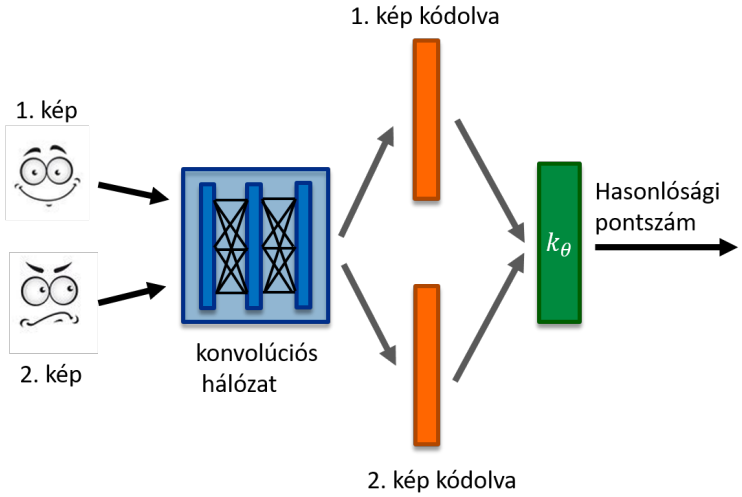
3.2.1. Konvolúciós sziámi hálózatok

A konvolúciós sziámi hálózatok két konvolúciós ikerhálózatból épülnek fel, amelyek megosztott súlyokat és rétegeket használnak. A hálózat két bementet kap, és miután a konvolúciós rétegek kiszámolták a jellemző vektorokat, ezeknek egy θ távolságfüggvénnyel méri a hasonlóságát. Ha a hasonlósági pont meghalad egy küszöbértéket, a két képet hasonlónak tekintjük, egyébként különbözőnek.

Sok ábrán a sziámi hálózatokat két azonos hálózattal reprezentálják. Valójában mivel a két ikerhálózat súlyai és rétegei megosztottak, elég egy hálózatot használni és csak a jellemző vektorokat eltárolni, így kevesebb erőforrást használunk.

Működésük közben egy mintáról nem azt tanulják meg, hogy melyik osztályba tartozik, hanem a különbséget a többi mintához képest. Tanítás közben a konvolúciós hálózatot súlyait javítják, hogy az általa képzett kódolások azonos minták esetén azonosak, különbözők esetén pedig különbözőek legyenek, így a távolságfüggvény jól fog működni.

A konvolúciós sziámi hálózatok megoldást jelentenek a few-shot learning problémára, mert kevés tanítómintával is jól működnek. Vegyük példaként egy vállalat arcfelismerő



3.1. ábra. Konvolúciós sziámi hálózat architektúrája.

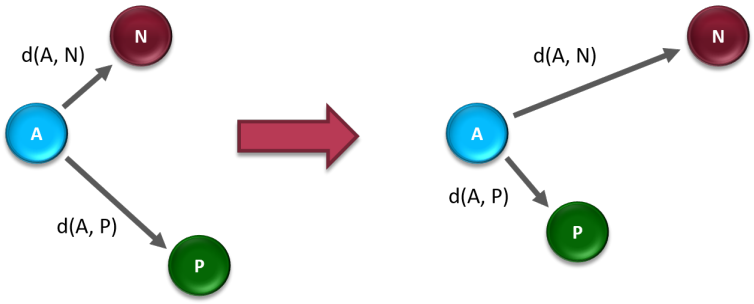
rendszerét. A sziámi hálózatnak elég egy segédhalmazban egy-egy képet eltárolni minden alkalmazottról. Amikor egy alkalmazott a rendszert használja, a képét összehasonlítja a segédhalmazban lévő képekkel és a hasonlóság alapján dönt. Ezzel két problémát old meg:

1. Egyrészt nem szükséges minden alkalmazottról több képet készíteni.
2. Másrészt új alkalmazottak érkezése, vagy egy alkalmazott távozása után nem szükséges újra tanítani a hálózatot.

A sziámi hálózatoknak a legismertebb költségfüggvénye a *triplet loss* függvény. Ezt a *gradient descent* módszerrel optimizálva tanítjuk a hálózatot. A triplet loss három bemenetet igényel, amelyek jelen példában a képekből képzett jellemző vektorok. Az egyik egy rögzített kép vektora, ez az ún. *anchor*. A másik kettő pedig egy pozitív és egy negatív minta. Az egyik ugyanabból az osztályból származik mint az *anchor* kép, a másik különbözőből.

$$\mathcal{L}_{triplet}(A, P, N) = \max(d(A, P) - d(A, N) + m, 0) \tag{3.2}$$

A d a távolságfüggvény, ami lehet például euklideszi távolság. A *triplet loss* veszi az anchor és a pozitív meg az anchor és a negatív minta távolságainak különbségét, majd ezt eltolja az m küszöbértékkel. Ha az előbbi pozitív ezt veszi eredményül, egyébként nullát.



3.2. ábra. A triplet loss függvény csökkenti a távolságot a hasonló és növeli a különböző minták között.

Akkor jó a vektorok elhelyezkedése a metrikus térben, ha a hasonlók között a távolság kicsi, a különbözők között pedig nagy. Azt szeretnénk elérni, hogy $d(A, P) \leq d(A, N)$ fenn

álljon. Átrendezve a $d(A,P) - d(A,N) \leq 0$ egyenletet kapjuk. Ezt kielégíti a $d(A,P) = 0$, $d(A,N) = 0$ megoldás. A másik triviális megoldás, ha a pozitív és negatív minta kódolása ugyanaz lenne, ekkor ugyanis $d(A,P) = d(A,N)$ így $d(A,P) - d(A,N) = 0$. Szeretnénk, ha a neurális hálózat nem nullvektorokkal vagy azonos vektorokkal kódolná az összes képet, ezért hozzáadunk egy m küszöbértéket az egyenlethez.

$$\begin{aligned} d(A,P) + m &\leq d(A,N) \\ d(A,P) - d(A,N) + m &\leq 0 \end{aligned} \tag{3.3}$$

Ideális esetben a $d(A,P) - d(A,N) + m$ negatív, ilyenkor a veszteség 0. Ha nem így van, a triplet loss ezt a veszteséget adja vissza. A költségfüggvény a tanítóhalmazban lévő tripletekre alkalmazott triplet lossok összege. Ezt minimalizálva a 3.2 ábrán látható távolságok csökkentése, növelése történik.

<triplet-ek kiválasztása>

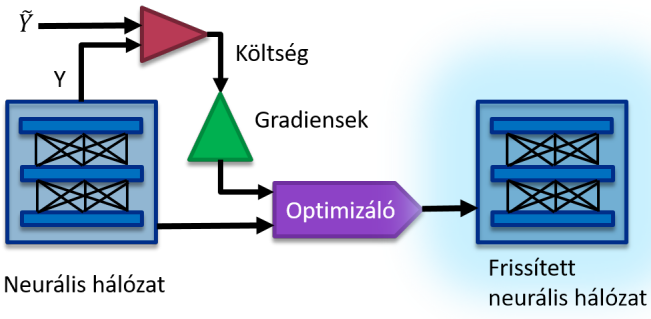
3.3. Optimizációs metatanítás

Az optimizációs algoritmusok mindig egy célfüggvényt szélsőértékét keresik. Neurális hálózatokban ez a célfüggvény a költségfüggvény felel meg és függ a modell tanulható paramétereitől. A optimizációs metatanítási megközelítések a modellek egyes paramétereit szintén modelleknek tekintik. Ezek a paraméterek tipikusan a modell kezdeti paraméterei (súlyok, eltolássúly) és az optimizációs algoritmus.

A fejezetben bemutatok egy megközelítést, ahol az optimizációs algoritmust modellezik az adott feladat tanításának felgyorsítása érdekében és két másik few-shot learning megoldást: a MAML és Reptile algoritmusokat, ahol a modell kezdeti paramétereit állítják be úgy, hogy könnyen adaptálható legyen új feladatokhoz.

3.3.1. Optimizáló algoritmus modellezése

Egy neurális hálózat tanulását mutatja a 3.3 ábra. A neurális hálózat által számolt kimenet és az elvárt kimenet közötti különbség alapján a költségfüggvénnyel számoljuk ki a hibát. Ezt jelöli a piros háromszög. A zöld háromszög a költségfüggvény gradienseit számolja ki az egyes rétegekre. Az optimizáló megkapja a gradienseket és a neurális hálózat súlyait, majd javít rajtuk.

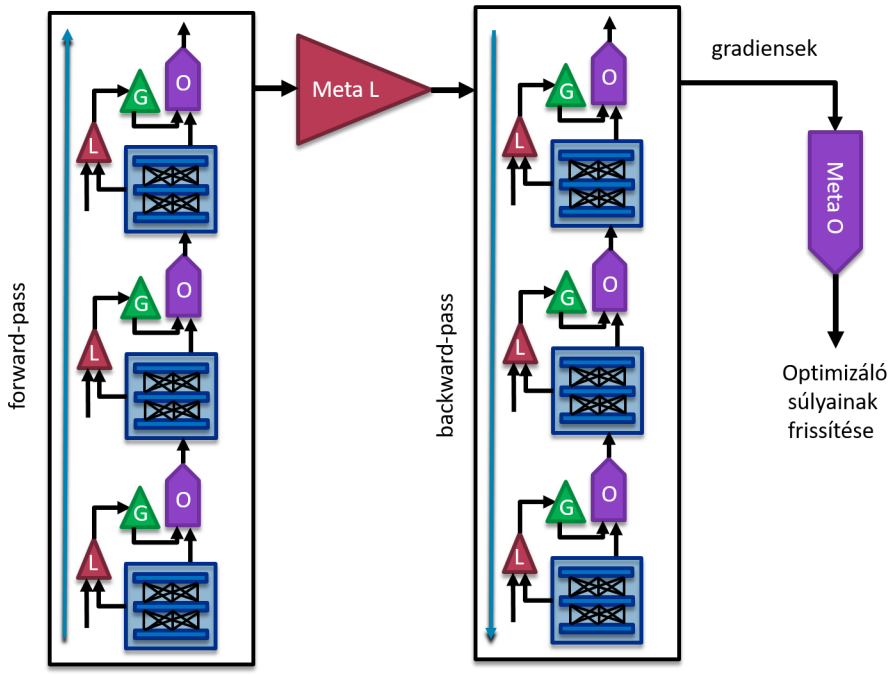


3.3. ábra. Neurális hálózat architektúra.

Tegyük fel, hogy a 3.3 ábrán látható neurális hálózatot bináris klasszifikációra használjuk és hívjuk simán modellnek. A modell kezdeti paramétereit minden iterációban az optimizáló frissíti. A metatanítás az optimizációs algoritmus szempontjából azt jelenti,

hogy az optimizáló állítható paramétereit nem mi állítjuk be kézzel, hanem a feladatot átadjuk egy másik modellnek, amit megtanítunk arra, hogy ezeket optimizálja miközben az eredeti modellünk tanul. Tehát az optimizáló az eredeti modellünk súlyait állítja, miközben a másik modellünk az optimizáló paramétereit javítja.

Ezzel absztrakciós szintet léptünk, ezt a másik modellt nevezzük metamodellnek. A metamodellnek ugyanúgy lesz meta-költségfüggvénye, meta-gradiensei és meta-optimizálója. Ugyanakkor látnunk kell, hogy ezt a metaoptimizálót is tekinthetjük modellnek és léphetünk feljebb metaszinteket, de előbb utóbb szükség lesz egy konkrét meta-optimizálóra, ami az alatta lévő optimizáló paramétereit állítja.



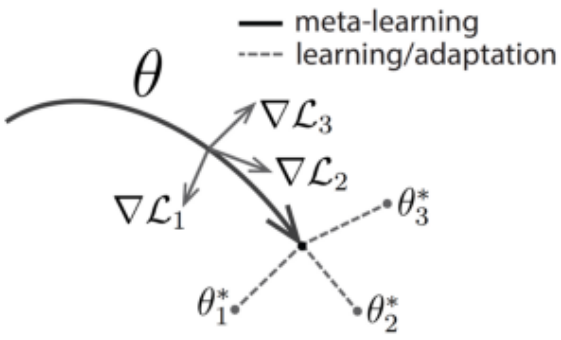
3.4. ábra. Optimizáló metatanítás.

A 3.4 ábra az optimizáló metatanítást szemlélteti. A konkrét modell szinten a fekete téglalapokban függőlegesen az eredeti bináris klasszifikációra használt modellünk tanítása történik, míg a téglalapok között vízszintesen a metamodell tanítása látszik.

A metaköltségnek vehetjük az eredeti modell költségeinek összegét egy adott iterációig. Ez jól leírja, hogy a modell tanul-e. A metaköltség-függvény alapján kiszámoljuk a meta-gradienseket, amit átadunk a metaoptimizálónak. Ez már egy konkrét optimizáló, például ADAM. Ezután a meta-optimizáló állítja az optimizáló paramétereit úgy, hogy csökkentse a meta-költséget, vagyis javítja a tanulási folyamatot.

3.3.2. Model-Agnostic Meta Learning (MAML)

A MAML a modell kezdeti paramétereit optimalizálja úgy, hogy gyorsan tanuljon. Célja, hogy új, hasonló feladatokra kevesebb - akár egy - iterációval jó eredményt érjen el az optimizációs algoritmus. A modellt több feladathoz adaptálja a kezdeti paraméterei beállításával, így az kevés gradiens frissítés után képes megtanulni új feladatokat. Ez egy megközelítés a few-shot learning problémára.



3.5. ábra. A MAML algoritmus vizualizációja. (Finn et al 2017)

Legyen a modell egy f_θ függvény, ahol θ jelöli a modell paramétereit. Amikor a modellt egy új $\mathcal{T}_i$ feladathoz adaptáljuk, a modell θ paramétereit változtatjuk θ_i'-re. Az adaptált paraméter egy gradiens frissítés esetén a következő:

$$\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta) \tag{3.4}$$

A 3.5 ábrán a θ jelöli a modell kezdeti súlyait. A szürke vonalak a $\nabla\mathcal{L}_1$, $\nabla\mathcal{L}_2$, $\nabla\mathcal{L}_3$ gradienseket mutatják, a θ_1^*, θ_2^*, θ_3^* pedig az adott feladathoz adaptált modell optimális paraméterei. A vastag vonal a metatanulási folyamatot mutatja. Látható, hogy a θ paraméterű modell jelenlegi helyzetében közel van mindhárom feladat optimális paramétereihez, tehát kevés gradiens frissítéssel finomhangolható bármelyikre.

Algorithm 1 Model-Agnostic Meta Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β step size hyperpatameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

Az algoritmus először veszi a $\mathcal{T}$ feladatok egy $p(\mathcal{T})$ eloszlását és a modell kezdeti paramétereit véletlen módon inicializálja. A feladatok közül kiválaszt párat és mindegyikre K tanítómintával tanítja a modellt. A tanítás során gradiens frissítésekkel kiszámolja az optimális θ_i modellparamétereket 3.4 szerint. A meta-célfüggvényt az adaptált θ_i paraméterekkel kiszámolt költségek összege adja a $\mathcal{T}_i$ feladatokon.

$$\min_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'}) = \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)}) \tag{3.5}$$

Mielőtt új $\mathcal{T}_\backslash$ feladatokat választ, sztochasztikus gradient descent módszerrel frissíti a modell kezdeti paramétereit a meta-célfüggvény szerint.

3.3.3. Reptile

A reptile algoritmus nagyon hasonlít a MAML-hoz. Ugyanúgy a hálózat kezdeti súlyait incicializálja úgy, hogy további hasonló feladatokra könnyen általánosítható legyen. A MAML-hoz képest előnye, hogy kevesebb számítást igényel, nincs szükség második deriváltak kiszámolására, csak SGD-t futtat a feladatokon.

Algorithm 2 Reptile

1: Initialize ϕ, the vector of initial parameters
2: **for** iteration = 1, 2, ... **do**
3: Sample task τ, corresponding to loss L_τ on weight vectors $\tilde{\phi}$
4: Compute $\tilde{\phi} = U_\tau^k(\phi)$, denoting k steps of SGD or Adam
5: Update $\phi \leftarrow \phi + \epsilon(\tilde{\phi} - \phi)$
6: **end for**

Adott a szigma kezdeti paraméter vektor. Valamennyi iteráción keresztül választunk egy véletlen feladatot és futtatjuk rajta valahányszor az SGD algoritmust, ami a ϕ paraméter vektorból a $\tilde{\phi}$-t eredményezi. Ezután javítjuk a modell kezdeti paramétereit a megadott szabály szerint.

3.4. Modell alapú metatanítás

A modell alapú metatanítás lényege az olyan modelleket tervezése, amelyek képesek kevés tanítással, gyorsan javítani a paramétereiket. Egy ismert implementáció a Memory-Augmented Neural Networks, ami neurális Turing-gépeket használ.

4. fejezet

Android alkalmazás beszélőfelismerésre

A következő fejezetben bemutatom egy általam a beszélőfelismerés szemléltetésére készített android alkalmazás fő elemeit és működését, a kapcsolódó technológiákat és implementációs részleteket. Az alkalmazásban konkrét beszélőfelismerésre készített neurális hálózatokat használok fel. A cél nem egy új beszélőfelismerő modell készítése volt, hanem az eddigiek felhasználásával egy működő alkalmazás létrehozása.

Az alkalmazás kliens-szerver architektúrájú lesz. Az android kliens segítségével a felhasználó regisztrálhat a rendszerbe névvel és hanggal, majd az azonosítás opció megnyomásával hangmintát adva az alkalmazás eldönti, hogy regisztrálva van-e, és ha igen, akkor visszaadja az illető nevét.

4.1. Modell predikció a felhőben vagy lokálisan?

A szervernek tárolnia kell a felhasználóktól származó hangmintákat és a felhasználó azonosítás folyamata közben összehasonlítani őket. Az összehasonlítást egy sziámi neurális hálózat végzi. Az alkalmazás felépítését tekintve a legfontosabb kérdés, hogy ezt az összehasonlítást központilag egy felhőben a szerver, vagy a kliens oldali alkalmazás végezze. Mivel a modell végzi a predikciót, ettől a döntéstől függően azt a szerveren vagy a kliens eszközökön kell tárolni. Mindkét architektúrának vannak előnyei és hátrányai is.

Mivel esetünkben az alkalmazás kész, előre tanított modelleket használ, nincs szükség a modellek tanítására. Ennek ellenére ha a jövőben saját modellt szeretnénk használni felmerül a kérdés, hogy hol tanítsuk azt.

- A felhőben hosztolt gépi tanulási szolgáltatások általában saját modelleket használnak. Mi a saját adatainkat átadjuk és a szolgáltatás gondoskodik a modell tanításáról. Ezután a predikciót egy API-n keresztül végezhetjük el. Figyelni kell arra, hogy ebben az esetben nem mindig miénk a modell. Ha nincs lehetőség tanítás után a modell letöltésére, akkor a predikció mindenképp a felhőben marad. Másrészről a magas szintű szolgáltatások kezelése könnyebb, nem kell érteni a neurális hálózatok tanításához, de nem elég flexibilisek. Általában nem változtathatunk a modellen és az API-n sem.
- A felhőben taníthatjuk a modellünket az erre készített szolgáltatások nélkül is nagyobb hozzáértést feltételezve. Ekkor megválaszthatjuk a modell típusát és a technológiákat (Tensorflow, Pytorch, stb.) és teljes mértékben miénk a modell és az irányítás. A lokális tanítással szemben további előny, hogy az erőforrásokat rugalmasan fel-le skálázhatjuk.

- Lokálisan is taníthatjuk a modellt. Ez főleg akkor éri meg, ha van elég erőforrásunk hozzá vagy a modell mérete nem indokolja több erőforrás használatát. Nagyobb modellek esetében a tanítási idő hosszú. Az árakat és az időt mérlegelve dönteni kell, hogy a lehetőségek közül melyiket indokolt választani.

Ha már rendelkezünk egy működő modellel, akkor el kell döntenünk, hogy a predikciót a felhőben vagy lokálisan az eszközön végezzük.

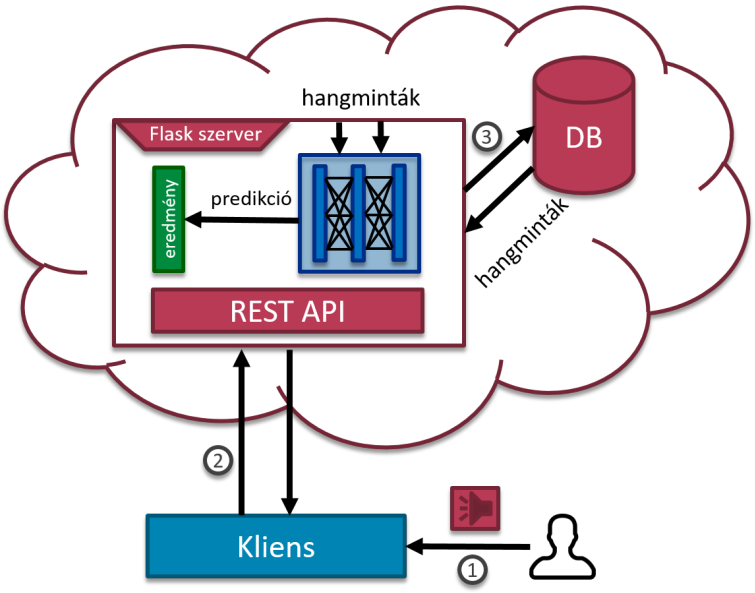
- A hálózati kapcsolatot tekintve ha a predikciót lokálisan végezzük, nincs szükség internetkapcsolatra, predikció a készüléken történik. Ellenkező esetben a kliens interneten keresztül kérést küldd a szervernek, ami elvégzi a predikciót és visszaküldi a választ.
- A modellt felhőben a szerveren tárolni biztonságosabb. Lokális predikció esetén a modellt a készüléken tároljuk. Ilyenkor gondoskodni kell a biztonságáról, hogy ne lehessen visszafejteni azt (reverse engineering).
- A predikció sebessége is meghatározó szempont. Ha lokálisan, a mobiltelefon hajtja végre, a felhőhöz képest kisebb erőforrással gazdálkodunk, ami miatt lassabb lesz a számítás. Viszont ha a felhőben végezzük, a kommunikációs overhead hozzáadódik a válaszhoz. Le kell mérni, hogy a kliens-szerver kommunikáció késleltetése mekkora az erőforráskülönbségből származó számítási időkhöz képest.
- Az alkalmazás architektúráját tekintve ha a modell és a predikció a szerveren történik, bármikor frissíthetjük, finomhangolhatjuk a modellt és ehhez nem kell a felhasználóknak frissítéseket letölteniük. Nincs szükség szinkronizációra. Az alkalmazás elosztott, elkülönült backend és frontend részekből áll, amelyek egy interfészen keresztül kommunikálnak, így ezek a komponensek lecserélhetők, csak az interfészt kell implementálniuk (API).
- Felhőbeli erőforrásokat használva az árat is figyelembe kell venni. Minél több felhasználó használ egy alkalmazást és amennyiben a számítás központilag a felhőben történik, annál több erőforrásra van szükség. Nagy előnye a készüléken végzett predikciónak, hogy jól skálázódik. Több felhasználó esetén nincs szükség több erőforrásra.

Jelen alkalmazás egy beszélőfelismerő szoftvert szemléltet, amelyet a valóságban vállalatok üzemeltetnek beléptető kapuval biometrikus azonosításra használva. Központi szerverre szükség van a regisztrált felhasználók adatainak biztonságos tárolása és a könnyű szinkronizáció miatt. Ha feltesszük, hogy a beléptető rendszert nem több mint tízezer ember azonosítására használják és a belépés szekvenciálisan történik, óriási erőforrásokat sem kell használni, mert a szerver egy időben legfeljebb a beléptető kapuk száma szerinti számítást kell végezzen, ami korlátos. Ezeket figyelembe véve úgy döntöttem, hogy a modell tárolása és a predikció szerveren fog történni.

4.2. Alkalmazás architektúra és általános működés

Mivel az alkalmazás tárolja a regisztrált felhasználók hangmintáit és a modellt; a biztonság, a könnyű központi adminisztráció és a tény, hogy nincs szükség szinkronizációra mind megerősíti a kliens-szerver architektúra szükségességét és előnyeit, így peer-to-peer megoldás fel sem merült.

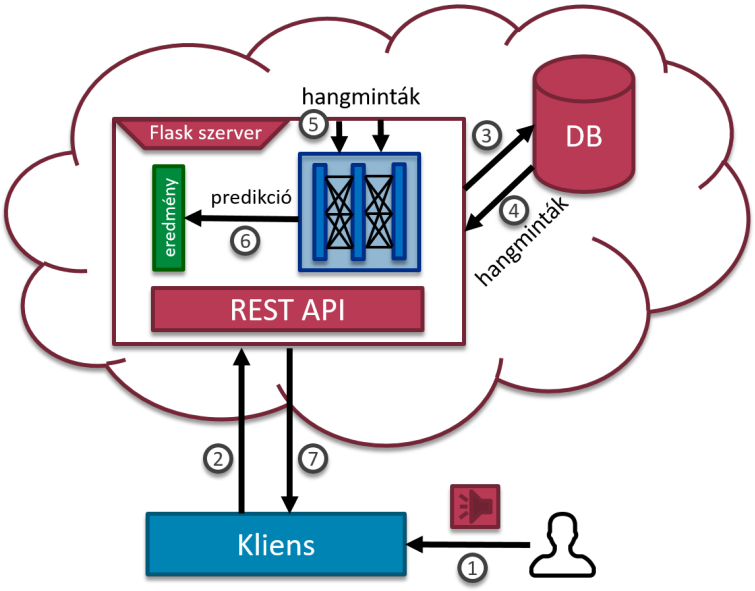
A 4.1 ábrán látható kliens a felhőben futó szerverrel egy REST API-n keresztül kommunikál. A kliens segítségével a felhasználó regisztrál a rendszerbe, ezután azonosíthatja magát. A regisztrációs fázis működése a következő:



4.1. ábra. Beszélőfelismerő alkalmazás: Regisztrációs fázis.

1. A kliens alkalmazás rögzíti a felhasználó hangját és nevét.
2. A kliens a hangmintát és a nevet elküldi a szervernek a REST API-n keresztül.
3. A szerver előfeldolgozza a hangfájlt, a modell segítségével hangvektort készít belőle és menti az adatbázisba.

Miután a felhasználók regisztráltak, az alkalmazás a hangjuk alapján azonosítani tudja őket. Az azonosítási fázis lépéseit a 4.2 ábra mutatja.



4.2. ábra. Beszélőfelismerő alkalmazás: Regisztrációs fázis.

1. A kliens rögzíti az azonosítani kívánt felhasználó hangját.
2. A hangmintát továbbítja a szervernek.

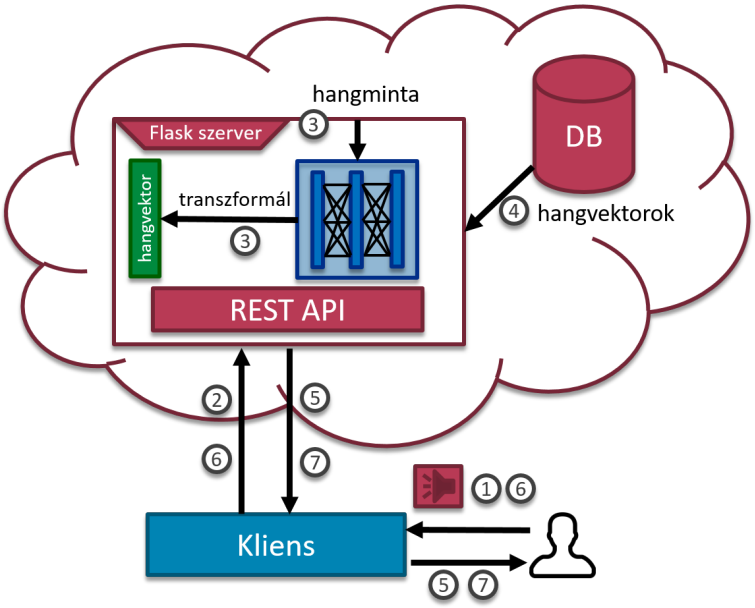
3. A szerveren futó alkalmazás a hangfájlt előfeldolgozza és hangvektort készít belőle.

4. A szerver kiszámolja a hangminták közötti hasonlósági pontokat. A minimálishoz tartozó felhasználó nevét elküldi a kliensnek.

4.2.1. Biztonsági funkciók

Azt is figyelembe kell venni, hogy azonosításkor a szerver a minimális vektortávolsághoz tartozó névvel tér vissza. Ez azt jelenti, hogy ha egy regisztrált felhasználó van a rendszerben, bárki képes lenne belépni a regisztrált felhasználó nevében, hiszen csak egy távolság van, ami minimális. Ennek elkerülésére további biztonsági funkciókkal kell kiegészíteni az alkalmazást;

Meg kell határozni egy biztonsági küszöböt a vektorok távolságára - azaz ha nem elég hasonlóak, egyéb autentikációs mechanizmusra lesz szükség. Ennek megfelelően a szervert kiegészítettem egy biztonsági küszöbbel, amit ha azonosításkor a vektortávolság nem ér el, jelszó alapú azonosítást kér a felhasználótól. A biztonsági funkcióval kiegészített működés abban az esetben, ha a biztonsági küszöböt nem éri el az alkalmazás a 4.3 ábrán látható. A regisztrációs fázis annyiban tér el az előzőtől, hogy a felhasználó egy jelszót is megad a neve mellett, amit a szerver eltárol.



4.3. ábra. Beszélőfelismerő alkalmazás: Azonosítás jelszó alapú autentikációval.

4.2.2. Vektorok átlagolása

A szerver oldali alkalmazás minden sikeres azonosítás után átlagolja a tárolt vektort az aktuális azonosítottal. Azonosításkor ha a hasonlósági pontszám a biztonsági küszöb alatt van, az átlagolás minden további nélkül megtörténik. Amennyiben a minimális vektortávolság nem üti meg a küszöbszintet az alkalmazás az aktuális vektort ideiglenesen eltárolja. Ezután ha a felhasználó sikeresen autentikál jelszóval, a vektort átlagolja a korábbival majd törli, egyébként csak törli azt.

4.2.3. Konfiguráció

A szerver oldali alkalmazás képes többféle hasonlósági pontot számolni a vektorok között. A jelenlegi implementáció a következő hasonlósági pontokat támogatja $v_1 = (x_1, x_2, \dots)$ és $v_2 = (y_1, y_2, \dots)$.

- Euklideszi-távolság:

$$d_{euc}(v_1, v_2) = \left(\sum_{i=1}^{n} x_i - y_i \right)^{\frac{1}{2}}$$

- Koszinusz-távolság:

$$d_{cos}(v_1, v_2) = \frac{v_1 \cdot v_2}{\|v_1\| \cdot \|v_2\|} = \frac{\sum_{i=1}^{n} x_i \cdot y_i}{\sqrt{\sum_{i=1}^{n} x_i^2} \cdot \sqrt{\sum_{i=1}^{n} y_i^2}}$$

Konfigurálható továbbá a biztonsági küszöbszám is. Ez azon értéke, melyet ha a hasonlósági pontszám nem ér el, a szerver jelszó alapú autentikációs kérést küld a kliens fele.

1. A kliens rögzíti az azonosítani kívánt felhasználó hangját.
2. A hangmintát továbbítja a szervernek.
3. A szerveren futó alkalmazás a hangfájlt előfeldolgozza és hangvektort készít belőle.
4. A szerver összehasonlítja a hangvektort a korábban eltároltakkal és kiszámol egy hasonlósági pontot vagy vektortávolságot.
5. A pontszám nem éri el a biztonsági küszöböt, ezért kliensnek felszólítást küld, hogy jelszó alapú autentikációt kér.
6. A kliens megadja a saját jelszavát, amivel korábban regisztrált és elküldi a szervernek.
7. A szerver ellenőrzi, hogy a megadott jelszó valóban az azonosítani kívánt felhasználóhoz tartozik-e. Amennyiben igen, visszaküldi az azonosított felhasználó nevét, egyébként jelzi, hogy az autentikáció sikertelen volt.

4.3. Felhasznált modellek

4.3.1. Voicemap

A *voicemap* egy beszélőfelismerő GitHub repository, ami tartalmazza a modell kódját, a tanítást, különböző teszteket és egy optimalizált előre tanított modellt. Az implementációhoz *Kerast* használ.

4.3.1.1. Az implementált modellek

Két modellt vizsgál; egy sziámi neurális hálózatot és egy sima konvolúciósat klasszifikációval. A sziámi hálózat alapja egy konvolúciós enkóder, amely a nyers hangmintákból kinyeri a jellemzőket és egy hangvektorokat állít elő belőlük. Az enkóder hálózat a következő konvolúciós blokkokból áll:

- Conv1D 32-es méretű filterekkel, ahol a filterek száma szorzódik a blokk számával. Az aktivációs függvény *ReLu*.
- BatchNormalization
- SpatialDropout1D
- MaxPool1D

Négy ilyen konvolúciós blokk követi egymást, majd egy *GlobalMaxPool1D* és egy *Dense* réteg a hangvektor méretével.

A sziámi hálózat alapja két enkóder hálózat, amelynek súlyai megosztottak, tehát úgy is tekinthetünk rá, hogy egy enkóder hálózatra két bemenetet adhatunk. A hangminták a hálózaton áthaladva jellemző vektorokká alakulnak. Ezt a részt a konvolúciós enkóder végzi. Ezután a két vektor közötti távolságot a sziámi hálózat kiszámolja.

Az implementált távolságmetrikák a következők:

- *weighted_l1*: Az eredeti one-shot cikk szerinti távolságmetrika.
- *uniform_euclidean*: Euklideszi távolság két vektor között.
- *cosine_distance*: Koszinusz távolság, azaz a két vektor által bezárt szög koszinuszát méri.

A kiszámolt távolság ezután minden esetben áthalad egy szigmoid aktivációs függvényű *Dense* rétegen, ami 0 és 1 közé nyomja az eredményt.

4.3.1.2. A k-way n-shot feladat

A repository tartalmaz egy utility szkriptet a hangminták preprocesszálásához és few-shot kiértékeléshez, amit később a saját alkalmazásomban is felhasználtam. Ez tartalmazza többek között az *n_shot_task_evaluation* függvényt is, ami *k-way n-shot* feladatokkal teszteli a modellt. Az alábbi argumentumokat adhatjuk meg:

- *model*: A tesztelendő modell.
- *dataset*: A tesztmintákat tartalmazó adathalmaz objektum.
- *preprocessor*: Előfeldolgozó függvény a minták csökkentett mintavételezésére és standardizálására.
- *num_task*: A feladatok száma.
- *n*: Hány hangminta tartozik egy beszélőhöz a segédhalmazban.
- *k*: Ennyi beszélő, azaz osztály van a segédhalmazban.
- *network_type*: Sziámi vagy klasszifikációs.
- *distance_metric*: Sziámi hálózat esetén a hangvektorok közötti távolságmetrika.

4.3.1.3. Beszédadatbázisok és generátorok

A voicemap tanításhoz és teszteléshez a *LibriSpeech* adatbázis *dev-clean*, *train-clean-100* és *train-clean-360* adathalmazokat használja, amelyek sorban 40, 251 és 921 különböző beszélőtől tartalmaznak nyers hangfájlokat. Egy adathalmazt egy adatgenerátor osztály reprezentál ami a keras.util.Sequence osztályból származik. Az adatgenerátorok nagy adathalmazok esetén hasznosak, amikor az egész adathalmaz nem fér bele a memóriába. Ilyen esetekben egyesével generálnak adatokat.

A *keras.util.Sequence* osztályból való leszármaztatás miatt az adatgenerátorban implementálni kell a *___len___* és *___get_item___* függvényeket. Előbbi az adathalmaz méretét adja vissza, utóbbi annak indexelését teszi lehetővé. Továbbá biztosítja, hogy egy epochon belül egy mintával csak egyszer tanítjuk a modellt.

A *LibriSpeechDataset* osztály ezen kívül képes még egy beszélőtől és különböző beszélőktől származó hangminta párok előállítására, tanításhoz hangminta pár kötegek építésére, amelyekben az azonos és különböző beszélőktől származó párok esélye 50%. Az objektum létrehozásakor választhatunk sima és sztochasztikus mód között. Míg az előbbi a minta elejétől számolva, utóbbi véletlenszerűen választja ki a megadott hosszúságú szakaszt egy mintán belül. A módtól eltekintve ha egy hangmintát kiegészíthetünk 0-ákból álló *paddinggel* is a megfelelő méretre. Fontos függvénye még a *build_n_shot_task*, ami validációhoz és kiértékeléshez *k-way n-shot* feladatokat hoz létre. A függvény visszaad egy hangmintát az azonosítandó beszélőtől és egy generált segédhalmazt, ami tartalmaz mintát az azonosítandó és egyéb beszélőktől is az n és k paraméterek alapján.

4.3.1.4. Tanítás

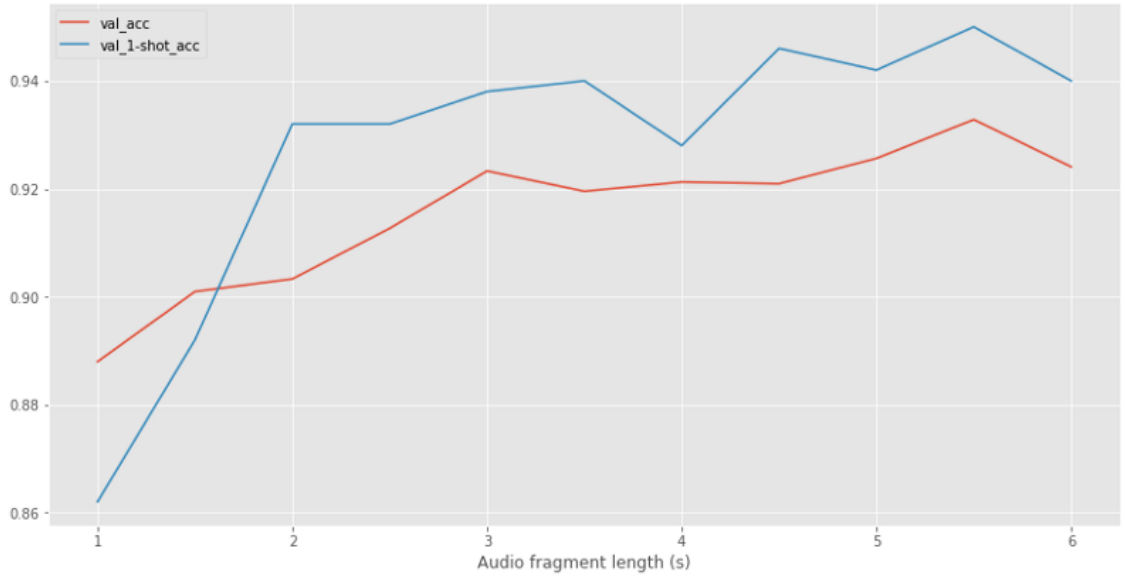
A sziámi és a sima klasszifikációs hálózat nagyrészt közös paraméterekkel rendelkeznek. A fő különbség, hogy míg a sziámi veszteségfüggvénye bináris keresztentrópia, és a veszteség az alapján dől el, hogy a hangminta pár egyazon vagy más beszélőktől származik, a klasszifikációs hálózat kategórikus keresztentrópiát használ, tehát a hangmintát megpróbálja besorolni k osztály valamelyikébe k beszélő esetén. A közös paraméterek:

- hangminta hossza: 3 sec
- batchsize: 64
- filterek száma: 128
- jellemző vektor dimenzió: 64
- dropout: 0
- steps_per_epoch: 500
- kiértékelési feladatok száma: 500
- n_shot_klasszifikáció: 1
- k_way_klasszifikáció: 5

Tehát mindkét modell optimizált hiperparamétereket használ, *Adam* optimizálót és veszteségfüggvényként keresztentrópiát. Egy epoch 500 batch iterációból áll és egy batch méret 64. Az epochok végén három callback fut le. Minden epoch végén kiértékelés történik: 500 *1-shot 5-way* feladat átlagos eredménye jelzi a pontosságot. Amennyiben ez növekszik a modellről *checkpoint* készül (*ModelCheckpoint*). Továbbá ha a kiértékelés során a pontosság nem nő, a *ReduceLROnPlateau* csökkenti a tanulási rátát.

4.3.1.5. Kísérletek és optimalizálás

A repositoryban számos kísérlet található a legjobb teljesítmény elérésére. A *wide_vs_tall* szkript leméri, hogy a modell hogyan teljesít különböző hosszúságú hangmintákkal tanítva. A mérés alatt *1-shot 5-way* feladatokkal, azaz 5 különböző beszélőtől 1-1 beszédmintával validálta a modellt.



4.4. ábra. Voicemap: Regisztrációs fázis.

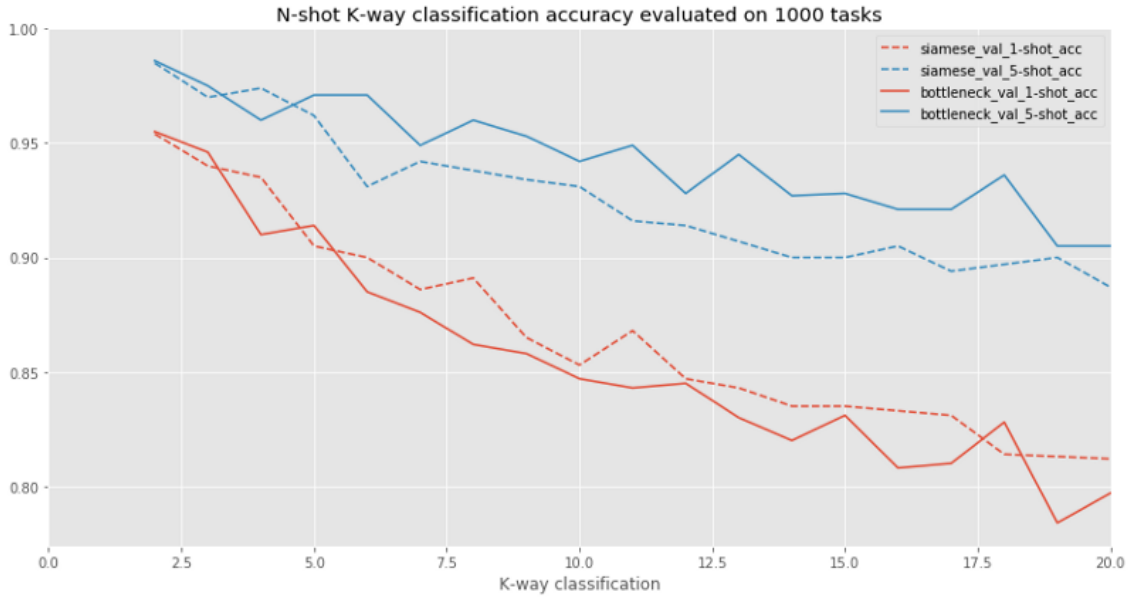
A 4.4 ábrán látható, hogy a hangminta méretét növelve a modell valóban jobban tanul, de a több adat miatt megnövekedik a tanítási idő és több memóriára van szükség. A grafikon mutatja, hogy 3 másodperc után a validáció stagnálni kezd, ezért az erőforrásokat és a tanítási időt figyelembe véve ez tűnik a legjobb választásnak.

A *grid_search_siamese_network* szkript hiperparaméter optimizációt végez a sziámi hálózaton a filterek számát, a hangmintákból képzett vektorok hosszát és a dropoutot vizsgálva. A talált legjobb paraméterek:

- filterek száma: 128
- jellemző vektor dimenzió: 64
- dropout: nincs

A *k_way_accuracy* kísérlet a hiperparaméter optimalizált modellt teljesítményét méri le különböző *n-shot k-way* feladatokkal. A pontosságot a következő ábra mutatja:

1-shot k-way feladatok esetén a sziámi hálózat átlagosan jobban teljesít sima klasszifikációsnál, de *5-shot k-way* feladatok esetén már az utóbbi kerül fölénybe. Ez valószínűleg annak köszönhető, hogy 5 hangminta - beszélő párból a sima osztályozó hálózat már jobban meg tudta tanulni a beszélőket a sziáminál.



4.5. ábra. Voicemap: n-shot k-way pontosság.

4.3.1.6. Saját kísérletek

4.4. Implementáció

4.4.1. Kliens alkalmazás

A kliens oldali android alkalmazást Android Studio segítségével készítettem el. Az alkalmazás felhasználói felületeket tartalmaz regisztrációhoz és azonosításhoz, képes hang rögzítésére wav formátumban és ennek http protokollon keresztüli továbbítására a szerverhez. A szerver válaszát az azonosításról jelzi a felhasználónak.

4.4.1.1. Projekt felépítése

Az Android Studios projekt főbb elemei a következők:

- *AndroidManifest.xml*: Engedélyek és activityk deklarálása.
- *java*: Activitykhez tartozó és egyéb java osztályok.
- *res/layout*: A felhasználói felülethez tartozó xml leírók.
- *build.gradle*: Gradle modul konfiguráció és dependenciák módosítására.

4.4.1.2. Engedélyek

Bizonyos műveletek elvégzéséhez az Android operációs rendszeren engedélyek szükségesek. Az engedélyek a művelet típusától függően *normál* vagy *veszélyes* osztályba tartoznak. Ha az alkalmazás olyan műveletet akar végrehajtani, amihez engedély szükséges, azt deklarálni kell az android projekt manifest fájljában.

A *normál* engedélyeket az operációs rendszer ezután automatikusan biztosítja, míg a *veszélyesekhez* a felhasználó engedélyét kéri. Korábbi android verziókban az engedélyek megadási az alkalmazás telepítésekor történt, de később ez megváltozott. A minimum Android 6.0-át (23-as API szint) használó készülékeken a felhasználó már nem telepítéskor,

hanem futási időben kell megadja az engedélyeket. Egy felugró ablak listázza, hogy az alkalmazás milyen engedélyeket kér, ezután választhat, hogy melyeket adja meg neki.

A beszélőfelismerő alkalmazás a következő engedélyeket használja:

- RECORD_AUDIO: A mikrofon használata hangfelvételhez.
- READ_EXTERNAL_STORAGE: A külső tárhely olvasásához.
- WRITE_EXTERNAL_STORAGE: A külső tárhely írásához.
- INTERNET: Internetelérés hálózati kommunikációhoz.

4.4.1.3. Tárhely

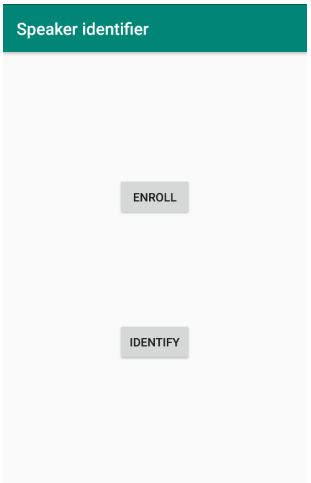
Az alkalmazás a felhasználótól vett hangmintát *wav* formátumban tárolja el. A *wav* egy tömörítetlen audioformátum és mivel később hangmintából jellemzőkinyerés történik, így nem veszíthetünk információt a jellemzőkről a tömörítés miatt. Később bemutatom, hogy a tömörített audioállományok milyen hatással vannak a modell teljesítményére.

Az Android beépített *MediaRecorder* osztálya a *wav* formátumot nem támogatja és nincs erre készült más beépített Java osztály sem, ezért egy erre készített kódot használtam fel Selvaline blogjáról. Selvaline *WavRecorder* osztálya az általunk beállított paraméterekkel (csatorna, mintavételi frekvencia, stb.) felveszi a hangot és egy *wav* fájlban tárolja azt.

Az Android operációs rendszeren használhatunk külső és belső tárhelyet. A belső tárhely előnye, hogy a fájlokhoz csak az alkalmazás fér hozzá és annak eltávolításakor a hozzá kapcsolódó fájlok is törlődnek. A külső tárhely előnye esetünkben, hogy könnyű mountolni. Ha a készüléket USB-vel számítógéphez csatlakoztatjuk, a külső tárhelyen lévő fájlokat böngészhetjük. Ez a hangfájl tesztelése miatt célszerű választás volt.

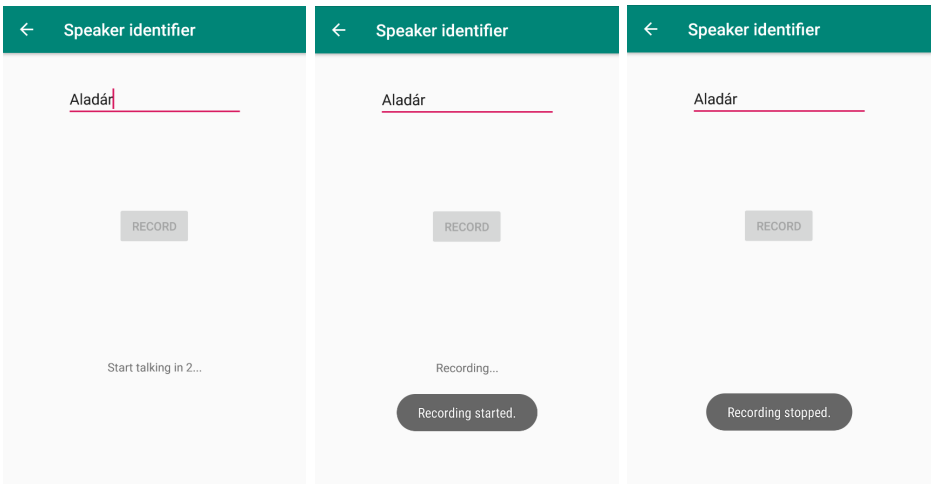
4.4.1.4. Felhasználói felület

A felhasználói felület három részből áll és minden felülethez egy-egy *Activity* osztály tartozik. A fő felületen választhat a felhasználó a regisztráció és az azonosítás között. A fő felületen az *Enroll* gombot megnyomva a regisztrációs, illetve ehhez hasonlóan az *Identify* opcióval az azonosító felületre ugorhatunk.



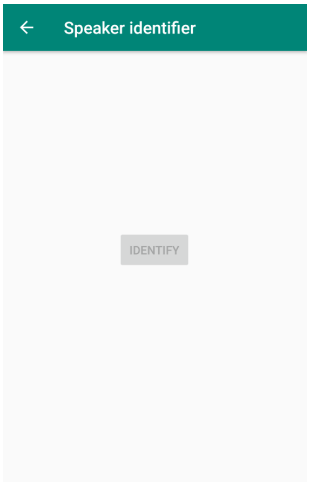
4.6. ábra. Menü.

A regisztrációs felületen megadjuk a nevet, majd a *Record* gombra kattintva három másodperc múlva elindul a hangfelvétel, ami négy másodpercen keresztül tart. A felületen megjelenő információ folyamatosan tájékoztatja a felhasználót a hangfelvétel folyamatáról.



4.7. ábra. Felhasználói felület regisztrációhoz.

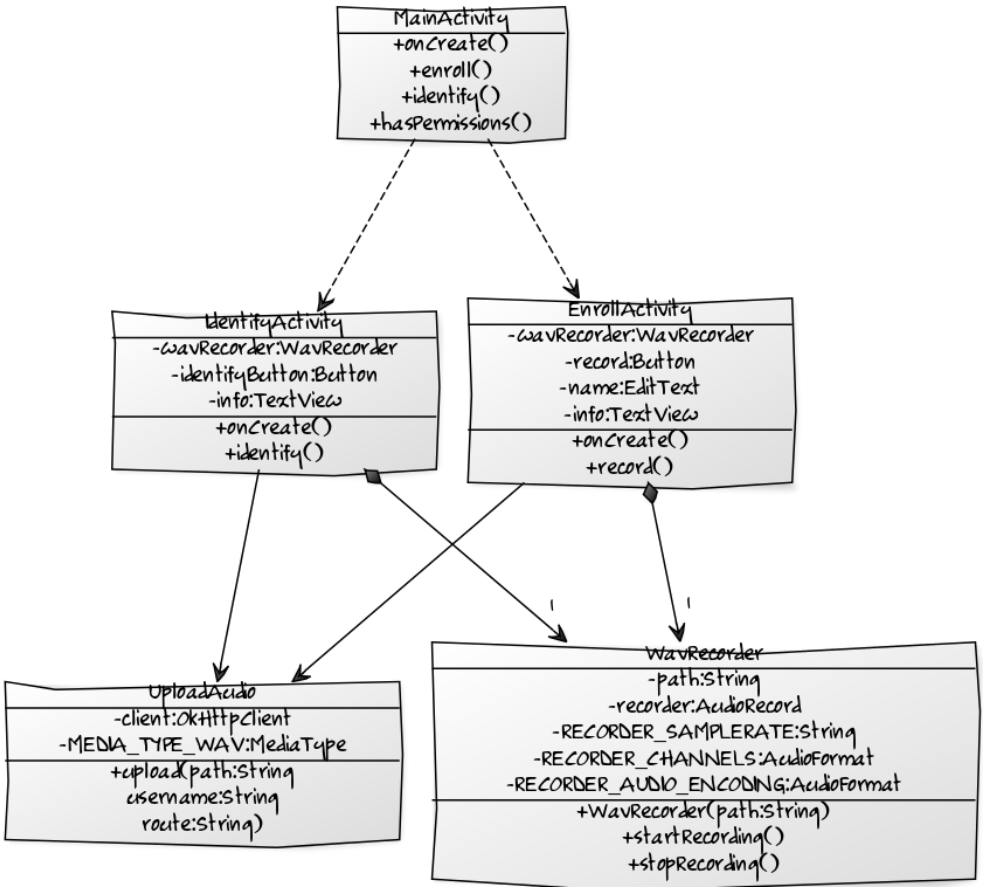
A zöld fejlécen a vissza nyílra kattintva visszajutunk a főmenübe. Ezt az opciót a manifest fájlban a *parentActivityName* tag megadásával érhetjük el.



4.8. ábra. Felhasználói felület azonosításhoz.

A főmenüből az azonosító felületre lépve az *Identify* gombra kattintva szintén visszaszámlálásra elindul a hangrögzítés, ami négy másodperig tart. Ezután megkapjuk a választ; az azonosított felhasználó nevét, vagy a nem sikerült azonosítani üzenetet.

4.4.1.5. Osztálydiagram és részletes működés



4.9. ábra. Beszélőfelismerő alkalmazás: Osztálydiagram.

A *MainActivity* osztály tartozik a főmenühöz. Amikor a főmenübe lépünk, az *onCreate* metódus hívódik meg, amely ellenőrzi, hogy az alkalmazásnak megvannak-e a szükséges engedélyei a *hasPermission* segédmetódus által. Ha igen akkor nem tesz semmit, egyébként a felhasználótól az összes engedély megadását kéri.

Az *enroll* metódus a felületen az *Enroll* gombhoz, az *identify* az *Identify* gombhoz tartozik. Mindkét függvény létrehoz egy-egy *Intent*-et ami a hivatalos definíció szerint egy elvégzendő művelet absztrakt leírása. Az *Intent* valójában arra jó, hogy egy felületről egy másik felületre lépjünk. A kiinduló felületen egy gombra kattintva (pl. *Enroll*), a gombhoz tartozó listener metódus (*enroll()*) hívódik meg, ami létrehoz egy *Intentet* paraméterként átadva az elérni kívánt felülethez tartozó *Activity* osztályt (*EnrollActivity*). Ezután az *Intentet* elindítva átlépünk a felületre.

A regisztrációs felületre lépve az *onCreate* metódus meghívásakor az *EnrollActivity* példányosít egy *WavRecorder* objektumot a fájl elérési útjával paraméterezve. A *record* metódus visszaszámlál, kiírja a felhasználónak az időzített üzeneteket, és elindítja majd megállítja a *WavRecordert*.

Az időzítést az ajánlás szerint a *Handler* osztály segítségével végeztem. Ennek előnye, hogy az időzített feladatokat egy háttérszálon futtatja, így nem blokkol. Így nem történhet

meg, hogy egy hosszabb feladat elvégzése miatt a felületen nem kattinthatunk amíg az nincs kész.

Az *IdentifyActivity* osztály nagyon hasonlóan működik. A különbség, hogy a *EnrollActivity*-hez képest nem küld nevet és más címre küldi a hangfájlt.

Mindkét osztály a *UploadAudio* objektumot használ a fájl szerverhez való elküldéséhez. Az *UploadAudio* egy *OkHttpClient* http klienst használ és *multipart/form* Http POST kérést küld a szerver felé a hangfájllal ill. regisztrációkor a névvel. Ezután egy callback metódussal várja a szerver válaszát, amit az azonosító felületre továbbít.

4.4.2. Szerver

Szerver oldalon egy Flask webalkalmazás fut felhőben és REST API-val rendelkezik. Fogadja a klienstől érkező hangfájlokat, előfeldolgozza és tárolja őket. Azonosításkor a modellen prediktál, majd visszaküldi az eredményt a kliensnek.

4.4.2.1. Flask

A Flask egy Python nyelvhez készült web mikrokeretrendszer. Mikrokeretrendszernek nevezzük a minimális webalkalmazás keretrendszereket, amelyekből általában hiányoznak a autentikációs, autorizációs könyvtárak, az objektum-relációs leképzés stb.

A Flask két fő BSD-licencelt komponensre épül:

- Werkzeug: A Werkzeug egy Python toolkit WSGI alkalmazásokhoz. Keretrendszereket lehet építeni rá és többféle Python verziót támogat. A WSGI egy hívási konvenció Pythonban írt webalkalmazásokhoz történő kérésekre.
- Jinja: Web template engine Python alkalmazásokhoz.

A Flask előnye, hogy nagyon gyorsan és könnyedén lehet vele REST API-t készíteni és elindítani egy webalkalamzást. Nincs szükség más könyvtárakra, toolokra hozzá.

Képes egyszerre több kérést is kezelni. Külön sessionökből érkező kérések külön szálakon futnak, így a keretrendszer a szálkezelést is megoldja helyettünk. A következő fontosabb funkciókat biztosítja:

- Webszerver fejlesztésre és debugger
- Unit tesztek integrált támogatása
- REST támogatása
- Jinja2 template weboldalakhoz
- WSGI 1.0 szabvány
- Jól dokumentált

4.4.2.2. Projekt felépítése

- *app*: A webalkalmazást indító és a logikát tartalmazó (előfeldolgozás, predikció, fájlok kezelése) illetve konfigurációt kezelő Python szkriptek könyvtára (app.py, preprocess.py).
  - app/app.py
  - app/model_loader.py

  - app/preprocess.py
  - app/config.py
  - app/paths.py

- *data*: Tartalmazza a klienstől regisztrációs fázisban kiszámolt vektorokat, az átlagolásra váró vektorokat és felhasználói adatokat.

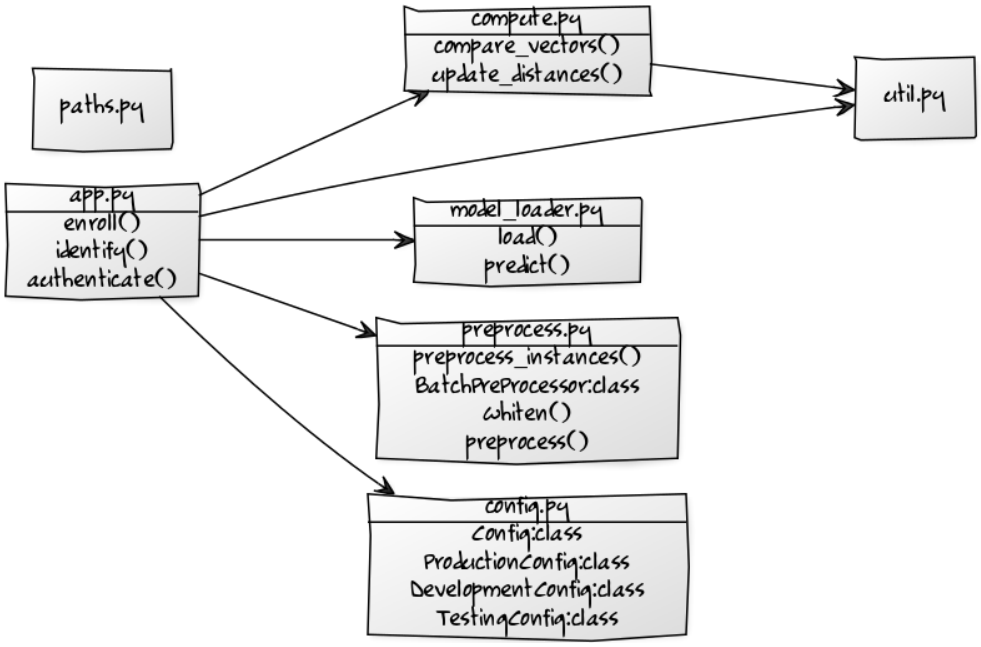
  - data/vectors: Az előfeldolgozott hangfájlok.
  - data/vectors_to_average: Az átlagolásra váró, ideiglenesen tárolt vektorok helye.
  - user_dict: Felhasználó UID - felhasználónév összerendeléseket tartalmazó dictionary.
  - passwords: Felhasználó UID - jelszó párokat tartalmazó dictionary.

- *model*: A beszélőfelismerésre használt modelleket tároló mappa.

4.4.2.3. Részletes működés

Az alkalmazás a következő REST végpontokat biztosítja a kliens számára:

- /enroll: A regisztráció végpontja.
- /identify: Az azonosítási végpont.
- /authenticate: A jelszó alapú autentikáció végpontja.



4.10. ábra. Flask szerver: Osztálydiagram.

- *app.py*: Az *app.py* implementálja a REST végpontokat. Az *app.py* először létrehoz egy Flask objektumot, ezután betölti a megfelelő konfigurációt a *config.py* szkriptben deklarált valamelyik objektummal. Betölti a modellt, majd elindítja az alkalmazást, ami a */enroll*, */identify* és */authenticate* útvonalakon *HTTP POST* kérésekre vár.

Mivel az alkalmazás egy prototípus és nincs szükség felhasználóspecifikus adatok tárolására néven és jelszón kívül, adatbázis helyett *dictionary* tárolja az egyedi azonosító - felhasználónév illetve egyedi azonosító - jelszó párokat egy fájlba szerializálva, a hangvektorok pedig egyedi azonosítónévvel vannak eltárolva.

  - *enroll()*: Regisztrációkor megkapja a klienstől a hangmintát, a nevet és a jelszót. Amennyiben a hangminta nem támogatott formátumú, HTTP 400 (Bad Request) státuszkóddal válaszol.

    Egyébként a felhasználónak létrehoz egy egyedi azonosítót és elmenti a jelszavát az *passwords* dictionarybe. Preprocesszálja a hangmintát, a modell segítségével hangvektort készít belőle majd eltárolja.

    A hangvektorok bináris formátumban kerülnek mentésre egyedi azonosító néven, így könnyen visszakereshetők.

    Végül frissíti a *user_dict* dictionaryt az aktuális felhasználó nevével és azonosítójával.

  - *identify()*: A kliens ide küldi a hangmintát azonosításra. Amennyiben a hangminta nem támogatott formátumú, HTTP 400 (Bad Request) státuszkóddal válaszol. Ha a hangminta támogatott preprocesszálja, majd hangvektort készít belőle. Összehasonlítja a korábban regisztrált felhasználók hangvektorjaival, és megkeresi a minimális hasonlósági pontszámot.

    Ha a minimális távolság meghaladja a biztonsági küszöböt, az aktuális vektort menti későbbi lehetséges átlagolás céljából. Ezután jelszó alapú autentikációs kérést küldd a kliensnek HTTP 403 (Forbidden) státuszkóddal. Mivel a HTTP protokoll állapotmentes, a kérés fejlécében elküldi a minimális távolsághoz tartozó felhasználó azonosítóját is az egyszerűség kedvéért (session kezelés helyett).

    Ha a minimális távolság a biztonsági küszöbön belül van, azaz az azonosítás sikeres volt, átlagolja az aktuális hangvektort az azonosítottal és HTTP 202 (Accepted) státuskóddal illetve a fejlécben az azonosított felhasználó nevével és egyedi azonosítójával válaszol.

  - *authenticate()*: A */authenticate* végpontra küldött jelszó alapú autentikáció esetén hívódik meg. A klienstől érkező kérés tartalmazza az azonosítani kívánt felhasználó UID-t és a jelszót. Ellenőrzi, hogy a küldött jelszó megegyezik-e a korábban eltárolttal.

    Amennyiben igen, az autentikáció elején eltárolt hangvektort átlagolja a korábbival, majd HTTP 202 (Accepted) státuskóddal illetve a fejlécben az azonosított felhasználó nevével és egyedi azonosítójával válaszol.

    Ha a jelszavas autentikáció nem sikerült, törli az átlagolásra váró hangvektort és HTTP 403 (Forbidden) státuszkóddal válaszol a kliensnek.

- *model_loader.py*: A fejlesztés kezdetén a modell betöltése a *app.py*-ban volt implementálva. Mivel a Flask több szálat használ, problémát okozott, hogy a modellt

valamelyik függvény hamarabb használta minthogy azt valóban betöltötte volna.

Erre többféle megoldási lehetőség van. Például kötelezhetjük a Tensorflowt globális default gráf használatára. Mivel ez a megoldás csak részben működött, más javaslat alapján külön osztályba, a *ModelLoader*be szerveztem. Ez az osztály felelős a modell betöltéséért illetve ez végzi el a predikciót.

- load(file_name): Betölti az paraméterként kapott útvonalon található modellt.
- predict(model_input, model): A bemenet és a választott modell alapján visszaadja a prediktált kimenetet.

A *ModelLoader* objektum inicializálásakor betölti a sziámi és a sima konvolúciós modellt is. Később a predict() függvényben paramétereiben állíthatjuk be, hogy melyik modellen szeretnénk a predikciót végrehajtani.

- *preprocess.py*:
  - *whiten(batch, rms)*: A *whiten* standardizálja a hangmintákat. Egy hangmintát egész számok sorozata reprezentál (PCM kódolás 4 vagy 16 biten). Először kivonja az átlagot minden számból, így a hangminta átlaga 0 lesz. Ezután újraskálázza a számokat, hogy ne legyenek kiugró amplitúdó értékek.
  - *preprocess_instances(downsampling, whitening)*: A *preprocess* függvénynek két fő paramétere van: a hangminta hossza és a *downsampling*, vagyis a csökkentett mintavételezési ráta. Ez a függvény végzi el a csökkentett mintavételezést és a standardizálást egy hangmintán.

    A csökkentett mintavételezés az eltérő mintavételi frekvencia miatt szükséges. Az Androidot futtató készülékek 8 és 16 bites PCM mellett 8, 16 és 44.1 kHz-es mintavételi frekvenciákat támogatnak. Esetünkben a modellt 4 kHz-en tanították a saját hangminták pedig 8 kHz-esek ezért 2-es csökkentett mintavételezés szükséges.
  - *BatchPreProcessor(mode, instance_preprocessor, target_preprocessor)*: Ez az osztály wrapper funckiót lát el. Ez segít abban, hogy egységesen tudjuk kezelni az (inputs, outputs) klasszifikációs batcheket és a ([inputs_1, inputs_2], outputs) sziámi típusúakat is. A *mode* paraméterrel megadjuk, hogy milyen típusú batchek előfeldolgozását szeretnénk. Az *instance_preprocessor* a bemenetek, a *target_preprocessor* a kimeneteket preprocesszáló függvény. Például *BatchPreProcessor('classifier', preprocess_instances(downsampling))*.
  - *preprocess(audio_file, sample_length, downsampling)*: Ez a függvény hatja végre a preprocesszálást. Beolvassa a paraméterként megadott hangmintát. A hangminta közepén *sample_length* hosszúságú darabot vág ki belőle. Létrehoz egy *BatchPreProcessor* objektumot, előfeldolgozza a batchet, majd visszatér a feldolgozott hangmintával.

    Mivel a jelenleg használt modell 3 másodperces és 4 kHz-en mintavételezett hangmintákat vár a bemenetére, a saját 4 másodperces hangmintákból 3 másodperces darabokat vág ki a közepétől számolva $1,5$-$1,5$ másodperces

távolságra. Erre azért van szükség, hogy ha a felhasználó kicsit később kezd el beszélni vagy hamarabb hagyja abba, a felvétel elején vagy végén lévő szünet ne kerüljön bele a mintába.

4.4.3. Fejlesztési környezet

Ebben a fejezetben bemutatom a fejlesztéshez használt technológiai stacket. Szó lesz a szerver oldali és a kliens alkalmazáshoz használt toolokról, környezetekről.

4.4.4. Git

A Git egy nyílt forráskódú verziókezelő, ami a fejlesztésben rendkívül fontos. A verziókezelés nem csak akkor hasznos, ha többen dolgoznak egy projekten. Egyéni kódbázist refaktorálás után ezáltal visszaállíthatunk, törölt fájlokat, kódrészleteket követhetünk vissza. A saját repositorymat GitHubon tároltam.

4.4.4.1. Amazon EC2

A webszerver a 4.1 fejezet alapján Amazon EC2-es virtuális gépen fut. Az Amazon Elastic Compute Cloud szolgáltatás biztonságos és skálázható számítási kapacitást nyújt felhőben. Virtuális gépek bérelhetők rajta, amelyek utána testreszabhatók tárhely és erőforrások; processzor, memória szempontjából.

A felhasználók létrehozhatnak saját virtuális gépeket, amelyeken saját alkalmazásokat futtathatnak. Többféle fizetési modell közül választhatunk: Létezik óra vagy másodperc alapú, de saját dedikált gépet is bérelhetünk. Az AWS Educate segítségével diákok korlátozottan de ingyenesen használhatják az Amazon EC2 szolgáltatásokat egy megadott kreditszintig.

A virtuális gépen Ubuntu 16.04 operációs rendszer fut és ssh-n keresztül férhetünk hozzá. Ehhez a gép létrehozásakor lehet új kulcspárt generálni vagy meglévőket használni. A gép bootolásakor a publikus kulcs bekerül a */.ssh/authorized_keys* fájlba. Később ssh-nál a saját privát kulcsunkat megadva tudunk autentikálin. A virtuális menedzselése az AWS Console-on keresztül történik. A gépen futó Flask szerver portján engedélyezni kellett a be és kimenő forgalmat. Ehhez létre kell hozni egy új *Security Groupot*, majd hozzáadni a megfelelő *Rule*-t.

Az alkalmazás szempontjából nagy előny, hogy a felhőbeli gépet könnyen elérjük, és fejlesztés közben sem a saját gépünk erőforrásait használja. Hátránya, hogy a kényelmes remote fejlesztés több konfigurációt igényel.

4.4.4.2. cmder

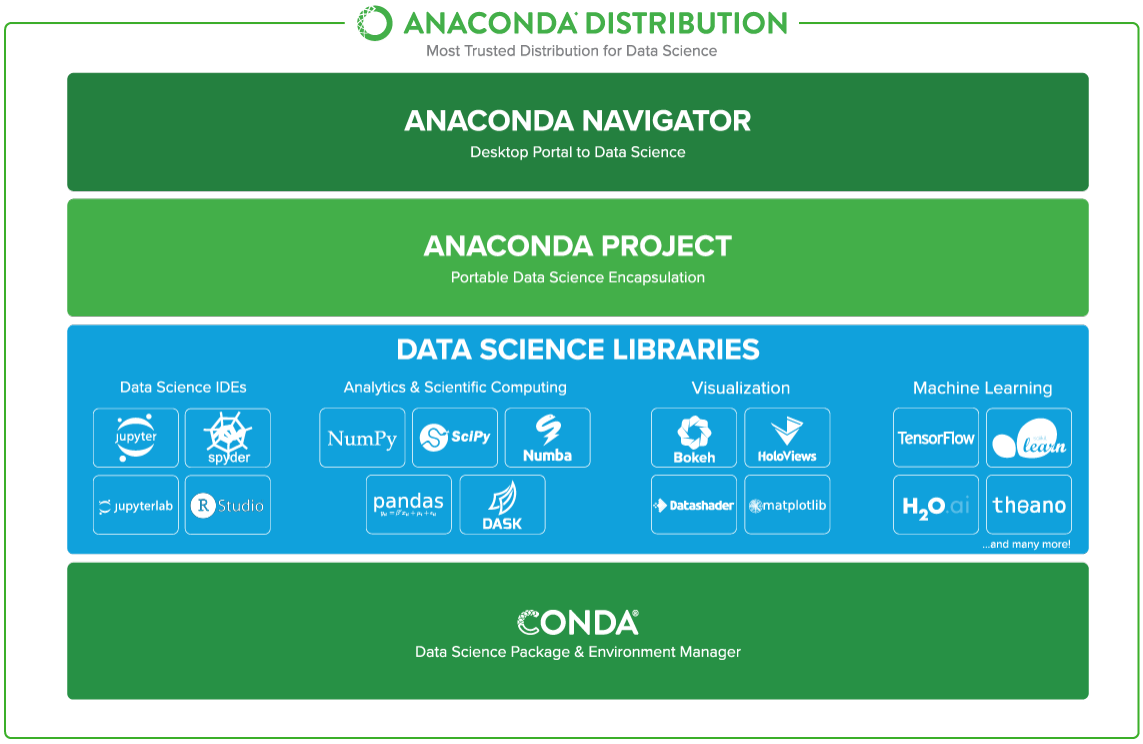
A cmder egy Windowshoz készült konzol emulátor. Beépített Gittel és ssh-klienssel rendelkezik és több Linuxos parancsot is támogat. Használata sokkal kényelmesebb a Puttynál, könnyedén csatlakozhatunk vele a Linuxos géphez.

Az ssh klienst paraméterezni kell a privát kulcsunkkal és a timeout elkerülése érdekében saját ssh konfigurációs fájllal. Eleinte probléma volt az ssh kapcsolat lejárási ideje, de ez kliens és szerveroldalon is könnyen konfigurálható. Linuxon a */etc/ssh/sshd_config*-ban a *ClientAliveInterval <seconds>* és *ClientAliveCountMax <seconds>*, Windowson

(jelen esetben kliens oldalon) a paraméterben megadott konfigurációs fájlban *ServerAliveInterval <seconds>* és *ServerAliveCountMax <seconds>* beállításokkal küszöbölhető ki.

4.4.4.3. Anaconda

Az Anaconda egy nyílt forráskódú Python és R disztribúció, amely több mint 1500 programcsomaggal, saját csomagkezelővel, grafikus felülettel és virtuális környezetkezelővel rendelkezik.



4.11. ábra. Flask szerver: Osztálydiagram.

Támogatja a gépi tanulást, tartalmazza a fontos gépi tanulási csomagokat (Keras, Tensorflow, Pytorch stb.), jupyter nootebookot. A virtuális környezetkezelővel a projekteknek külön virtuális környezetet hozhatunk létre. Ez fontos a verziók ütközésének elkerülése végett; Ha egy adott projektet Python 2.7-et használ, egy másik pedig Python 3.6-ot vagy ugyanolyan de eltérő verziójú programcsomagokat, akkor érdemes külön környezetet létrehozni nekik. A környezetek között a parancssorban navigálhatunk, csomagokat telepíthetünk rajtuk.

Az Anaconda Cloud további csomagokat, környezeteket, publikus és privát notebookokat tartalmaz. Ide feltölthetünk sajátot, illetve kereshetünk, telepíthetünk közülük csomagokat, környezeteket a saját gépünkre, környezetünkre.

4.4.4.4. JetBrains PyCharm

A JetBrain PyCharm egy Python programozási nyelvhez készült IDE. Azért esett erre a választás, mert van benne automatikus kódkiegészítés, intelligens kód navigáció - definícióra vagy használatra ugrás - és kód refaktorálás. Utóbbi a kód formázását és az importált csomagok rendezését jelenti.

Új projekt létrehozásakor saját virtuális környezetet hoz létre, de megadhatunk neki anaconda által létrehozottat, azt is képes kezelni. Jelen esetben a legfontosabb funkció a remote fejlesztés integrált támogatása volt.

A projekt beállításokban új Deployment létrehozásakor megadható a virtuális gép címe és a privát kulcs ssh-hoz. Megadhatók a lokális a remote mappák közötti mappingek és részletes szinkronizációs beállítások. A projekt interpreterének a remote gépen lévő Anaconda virtuális környezete által használt Python interpretert kell megadni, ez alapján a PyCharm feloldja a hozzá tartozó környezetet, látni fogja a programcsomagokat.

Szintén hasznos funckió a remote debugger, aminek segítségével a távoli virtuális gépen futó alkalmazást tudjuk debuggolni lokálisan.

4.4.5. Android Studio

4.4.6. Genymotion

Köszönetnyilvánítás

Ez nem kötelező, akár törölhető is. Ha a szerző szükségét érzi, itt lehet köszönetet nyilvánítani azoknak, akik hozzájárultak munkájukkal ahhoz, hogy a hallgató a szakdolgozatban vagy diplomamunkában leírt feladatokat sikeresen elvégezze. A konzulensnek való köszönetnyilvánítás sem kötelező, a konzulensnek hivatalosan is dolga, hogy a hallgatót konzultálja.

Irodalomjegyzék

[1] Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar: Diplomaterv portál (2011. február 26.). http://diplomaterv.vik.bme.hu/.

[2] James C. Candy: Decimation for sigma delta modulation. 34. évf. (1986. 01) 1. sz., 72–76. p. DOI: 10.1109/TCOM.1986.1096432.

[3] Gábor Jeney: Hogyan néz ki egy igényes dokumentum? Néhány szóban az alapvető tipográfiai szabályokról, 2014. http://www.mcl.hu/~jeneyg/kinezet.pdf.

[4] Peter Kiss: Adaptive digital compensation of analog circuit imperfections for cascaded delta-sigma analog-to-digital converters, 2000. 04.

[5] Wai L. Lee – Charles G. Sodini: A topology for higher order interpolative coders. In *Proc. of the IEEE International Symposium on Circuits and Systems* (konferenciaanyag). 1987. 4-7 05., 459–462. p.

[6] Alexey Mkrtychev: Models for the logic of proofs. In Sergei Adian – Anil Nerode (szerk.): *Logical Foundations of Computer Science*. Lecture Notes in Computer Science sorozat, 1234. köt. 1997, Springer Berlin Heidelberg, 266–275. p. ISBN 978-3-540-63045-6. URL http://dx.doi.org/10.1007/3-540-63045-7_27.

[7] Richard Schreier: *The Delta-Sigma Toolbox v5.2*. Oregon State University, 2000. 01. http://www.mathworks.com/matlabcentral/fileexchange/.

[8] Ferenc Wettl – Gyula Mayer – Péter Szabó: *LaTeX kézikönyv*. 2004, Panem Könyvkiadó.