Automatsko prepoznavanje znakovnog jezika

Kurtin, Luka

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1510

AUTOMATSKO PREPOZNAVANJE ZNAKOVNOG JEZIKA

Luka Kurtin

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1510

AUTOMATSKO PREPOZNAVANJE ZNAKOVNOG JEZIKA

Luka Kurtin

Zagreb, lipanj 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 4. ožujka 2024.

ZAVRŠNI ZADATAK br. 1510

Pristupnik:	**Luka Kurtin (0036534475)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	prof. dr. sc. Hrvoje Mlinarić

Zadatak: **Automatsko prepoznavanje znakovnog jezika**

Opis zadatka:

Proučite postojeća rješenja za detekciju znakovnog jezika. Izvršiti klasifikaciju slova znakovnog jezika. Prilikom postupka klasifikacije potrebno je definirati koordinate svakog slova znakovnog jezika te ih pohranjivati u bazu. Napisati program za izradu baze te provjeriti dobivene rezultat. Koordinate slova znakovnog jezika porebno je dobiti korištenjem radnog okvira Mediapipe Hands. Za provjeru rezultat napravite aplikaciju za mobilne uređaje koji će omogućiti prepoznavanje pojedinih slova znakovnog jezika.

Rok za predaju rada: 14. lipnja 2024.

Sadržaj

Uvod

U današnjem svijetu, komunikacija omogućava razmjenu informacija i ideja te predstavlja jednu od glavnih komponenti povezivanja različitih zajednica i kultura. Međutim, glavni izazov s kojim se ljudi suočavaju osiguravanje je učinkovite komunikacije za sve članove društva, bez obzira na razlike u kulturi, jeziku i jezičnim sposobnostima. Osobe s oštećenjem sluha koje koriste znakovni jezik nailaze na svakakve prepreke u svakodnevnoj komunikaciji, a uz to i dalje se suočavaju s manjkom tehnoloških rješenja koja su prilagođena njihovim potrebama i koja bi im olakšala komunikaciju. Veliki problem u njihovoj komunikaciji predstavlja i relativno malen broj ljudi koji su barem u nekoj mjeri upoznati s korištenjem znakovnog jezika za komunikaciju. Cilj ovog rada stvaranje je mobilne aplikacije koja bi olakšala korisnicima komunikaciju s korisnicima znakovnog jezika. Osnova aplikacije korištenje je umjetne inteligencije za prepoznavanje američkog znakovnog jezika (ASL). Razvoj ovog rješenja bazira se na modelu TensorFlow Lite umjetne inteligencije izrađenom pomoću programskog jezika Python. Model koristi koordinate dlana dobivene pomoću radnog okvira Mediapipe Hands za treniranje. Izrađen i istreniran model integriran je u mobilnu aplikaciju razvijenu u programu Android Studio koristeći programski jezik Kotlin i sučelje Jetpack Compose. Sama aplikacija omogućava prepoznavanje znakova američke znakovne abecede u stvarnom vremenu, nudeći brzo i precizno sučelje za komunikaciju na američkom znakovnom jeziku. Osim toga, aplikacija sadrži detaljne opise i slike koji korisnicima pružaju jasne upute o tome kako izvesti određeno slovo abecede. Kako bi se osiguralo spremanje podataka prethodno klasificiranih znakova, korištena je dodatna biblioteka Room za stvaranje baze podataka.

1. Korišteni alati i programi

U ovom poglavlju bit će objašnjeni alati i programi korišteni u razvoju mobilne aplikacije za prepoznavanje znakovnog jezika pomoću umjetne inteligencije.

1.1. Android Studio

Android Studio [1] integrirana je razvojna okolina stvorena od strane JetBrains-a i Google-a, a moguće ju je pokrenuti na računalima s operativnim sustavom Windows, macOs ili Linux. Programerima omogućava jednostavan i učinkovit razvoj mobilnih aplikacija korištenjem raznih alata dostupnih unutar same razvojne okoline, a podržava programske jezike Java, C++ i Kotlin. Android Studio prilagođen je razvoju aplikacija za sustav Android, a korisniku pruža mogućnost analiziranja i otkrivanja pogrešaka u aplikaciji te pokretanje aplikacije na vlastitom fizičkom uređaju ili na emuliranom uređaju unutar same razvojne okoline.

1.1.1. Kotlin

Od sredine 2019. godine, programski jezik Kotlin zamijenio je programski jezik Java kao Google-ov preferirani programski jezik za razvoj Android aplikacija. Programski jezik Kotlin [2] ima mnoge prednosti u odnosu na programski jezik Javu, kao što su kraća sintaksa i olakšavanje pronalaska grešaka prilikom rada s podacima različitog tipa tijekom programiranja, što ga čini prikladnijim za razvoj mobilnih aplikacija.

1.1.2. Jetpack Compose

Programski paket Jetpack Compose [3] alat je koji radi isključivo s programskim jezikom Kotlin,. Služi za brže stvaranje korisničkih sučelja Android aplikacije u odnosu na korištenje XML datoteka korištenih u programskom jeziku Java koje služe za dizajniranje različitih zaslona aplikacije. Ovakav pristup izgradnje korisničkih sučelja omogućava lakše održavanje te bolju čitljivost i razumljivost koda. Razlike između izgradnje korisničkog sučelja pomoću XML datoteka i alata Jetpack Compose prikazane su tablicom (Tablica 1.1).

Tablica 1.1 Razlike između XML datoteka i alata Jetpack Compose

	XML	Jetpack Compose
Syntax	Verbose, declarative and too much text	Declarative, programmatic and concise
Language	Java or Kotlin	Kotlin only
Code structure	Each layout has a separate XML file. Sometimes complex and hard to maintain	Inline UI Kotlin code and easy to maintain
Reactivity	Non-reactive	Reactive
Learning curve	Easy to learn	Easier to learn than XML
Customization	Customizable	Higly customizable than XML
Adoption	Widely used in old/earlier projects	Widely used in recent projects
Community	Extensive	Rapidly growing
Era	Old	Modern
Owner	W3C	Google Inc.

Alat koristi Composable metode [4], označene notacijom `@Composable`, koje se mogu kombinirati s drugim Composable ili običnim metodama te tvoriti složenije funkcije na način koji pruža jednostavnu prilagodbu koda. Definiranje Composable metode prikazano je kodom (Programski kod 1.1)

```
@Composable
fun Greeting(name: String, modifier: Modifier = Modifier) {
    Text(
            text = "Hello $name!",
            modifier = modifier
        )
}
```

Programski kod 1.1 Definiranje Composable metode

Alat također omogućava programerima pregled Composable metoda direktno u razvojnom okruženju, bez potrebe za pokretanjem aplikacije, ako se metoda anotira s `@Preview`. Primjer korištenja prikazan je kodom (Programski kod 1.2).

```
// A basic composable function
@Composable
fun Greeting(name: String) {
    Text(text = "Hello $name!")
}

// Previewing the composable function in Android Studio
@Preview
@Composable
fun PreviewGreeting() {
    Greeting("Android")
}
```

Programski kod 1.2 Definiranje Preview metode

1.2. TensorFlow

Biblioteka TensorFlow [5] je biblioteka otvorenog koda (eng. open-source) za razvoj umjetne inteligencije i duboko učenje koja omogućava brzu i učinkovitu implementaciju modela umjetne inteligencije na raznim uređajima i platformama, od web-stranica, preglednika i servera do mobilnih uređaja, mikrokontrolera i sklopova FPGA. Ključna značajka biblioteke TensorFlow korištenje je tenzora, struktura sličnih višedimenzionalnim poljima, što omogućava učinkovitu obradu podataka i optimizaciju resursa pri izradi modela umjetne inteligencije. Biblioteka podržava mnoge arhitekture modela, poput konvolucijskih neuronskih mreža (eng. Convolutional Neural Network, CNN) ili rekurentnih neuronskih mreža (eng. Recurent Neural Network, RNN).

Biblioteka TensorFlow podijeljena je u više različitih biblioteka od kojih je svaka prilagođena različitim potrebama i uređajima:

- TensorFlow Core
  - osnovna verzija za širok spektar aplikacija
  - obično na snažnim računalima i serverima zbog zahtjevnih operacija tokom treniranja modela
- TensorFlow Extended (TFX) [6]
  - Pruža alate za upravljanje životnim ciklusom modela umjetne inteligencije
  - na serverima i u oblaku

- TensorFlow.js [7]
  - za izvođenje izravno u web preglednicima (računala, tableti i pametni telefoni)
- TensorFlow Quantum
  - za razvoj kvantnih modela strojnog učenja
- TensorFlow Lite [8]
  - za implementaciju modela umjetne inteligencije na mobilnim uređajima, u mobilnim aplikacijama
- TensorFlow Lite for Microcontrollers
  - verzija prilagođena uređajima s mikrokontrolerima (senzori i ugradbeni sustavi)

U sljedećem poglavlju 1.2.1 bit će rečeno malo više o TensorFlow Lite verziji biblioteke koja je korištena za izradu ove aplikacije.

1.2.1. TensorFlow Lite

Biblioteka TensorFlow Lite (TFLite) verzija je biblioteke TensorFlow koja je optimizirana za upotrebu na mobilnim uređajima. Ova biblioteka omogućava brzu i laganu implementaciju modela umjetne inteligencije na pametnim telefonima i tabletima koji uglavnom imaju manju procesorsku snagu i memoriju, tj. ograničenije resurse od računala. Biblioteka TensorFlow Lite optimizirana je za izvođenje izravno na uređaju, bez potrebe za stalnom internetskom vezom. Postupak prilagodbe modela TensorFlow Lite za mobilne uređaje naziva se kvantizacija. Ovim postupkom smanjuje se broj bitova korištenih za prikaz težina u modelu, čime se ujedno smanjuje potrebna memorija te povećava brzina izvođenja. Kvantizacijom se postiže optimalna ravnoteža između performansi modela i korištenih resursa.

1.3. MediaPipe

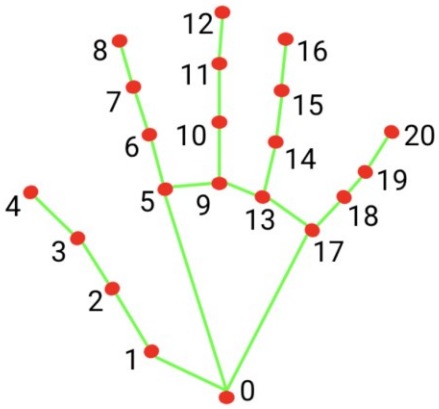
MediaPipe [9] je radni okvir (eng. framework) otvorenog koda kojeg je razvio Google, a omogućuje izradu prilagođenih modela strojnog učenja. Ovi modeli olakšavaju razvoj aplikacija za analizu i obradu podataka kao što su detekcija objekata, lica, ruku i poza,

praćenje pokreta i prepoznavanje gesti, analiza lica i slično. Radni okvir MediaPipe može se koristiti na računalima i mobilnim uređajima, u područjima računalnog vida, obrade slike i videa, virtualne i proširene stvarnosti, robotike i sl. Koristi cjevovod (eng. pipeline) koji predstavlja tok podataka kroz više različitih koraka obrade. Podržano je više vrsta ulaznih podataka (npr. video, slika, audio zapis, itd.) koji se obrađuju korištenjem niza modula za obradu podataka (eng. calculators) od kojih se sastoji cjevovod. Svaki od modula na određeni način (npr. detekcija objekata, prepoznavanje gesti, itd.) obrađuje određene podatke koje dobije kao ulaz te nakon obrade generira rezultat koji služi kao ulaz nekom drugom modulu. Osim modula i cjevovoda, radni okvir MediaPipe sadrži i predefinirane modele strojnog učenja trenirane na velikim skupovima podataka. Ove modele koriste moduli pri obradi podataka. Nakon prolaska podataka kroz cjevovod i sve module za obradu, generiraju se izlazni rezultati koje se dalje može koristiti u druge svrhe.

1.3.1. MediaPipe Hands

Radni okvir MediaPipe Hands [10] dio je radnog okvira MediaPipe specijaliziran za detekciju, praćenje i analizu pokreta dlanova te detekciju koordinata dlanova u stvarnom vremenu na temelju videa ili slike. Radni okvir sastoji se od dva međusobno povezana modela neuronskih mreža Prvi model je Palm Detection Model za prepoznavanje dlana neovisno o položaju dlana na ekranu, a drugi je Hand Landmark Model određuje 21 točku dlana određene paarametrima prikazanim na (Slika 1.1). Svaka točka dlana sadrži x, y i z koordinate, gdje su x i y koordinate izražene u pikselima, dok je z koordinata definirana udaljenošću dlana od ekrana. Tijekom korištenja modela moguće je podesiti neke od varijabli modela, kao što su MAX_NUM_HANDS, MIN_DETECTION_CONFIDENCE, MIN_TRACKING_CONFIDENCE i sl. Rezultat dobiven korištenjem radnog okvira MediaPipe Hands sastoji se od tri kolekcije:

- MULTI_HAND_LANDMARKS – kolekcija koja sadrži koordinate dobivene pomoću radnog okvira MediaPipe Hands
- MULTI_HAND_WORLD_LANDMARKS – kolekcija s istim koordinatama kao i prethodna, izraženim u metrima
- MULTI_HANDEDNESS – kolekcija u kojoj su spremljeni podatci o tome je li detektiran dlan lijeve ili desne ruke (sa sve detektirane dlanove)



0. WRIST
1. THUMB_CMC
2. THUMB_MCP
3. THUMB_IP
4. THUMB_TIP
5. INDEX_FINGER_MCP
6. INDEX_FINGER_PIP
7. INDEX_FINGER_DIP
8. INDEX_FINGER_TIP
9. MIDDLE_FINGER_MCP
10. MIDDLE_FINGER_PIP
11. MIDDLE_FINGER_DIP
12. MIDDLE_FINGER_TIP
13. RING_FINGER_MCP
14. RING_FINGER_PIP
15. RING_FINGER_DIP
16. RING_FINGER_TIP
17. PINKY_MCP
18. PINKY_PIP
19. PINKY_DIP
20. PINKY_TIP

Slika 1.1 Specifične točke dlana radnog okvira MediaPipe Hands

2. Model TensorFlow Lite za detekciju znakova

Program za izradu modela TensorFlow Lite za prepoznavanje znakovnog jezika rađen je u programskom jeziku Python. Kod je podijeljen u dva programa, program makeDatabase.py za pripremu podataka koji služe kao ulaz za model, te program train.py kojim se izrađuje model TensorFlow Lite. Za pisanje programa korištena je verzija 3.10.0 programskog jezika Python, a za izradu modela verzija 2.15.0 biblioteke Tensorflow.

2.1. Program za pripremu podataka

Program za pripremu podataka 'makeDatabase.py' omogućava korisniku detekciju koordinata dlanova korištenjem radnog okvira Mediapipe Hands. Koordinate je moguće detektirati iz skupa prethodno pripremljenih fotografija, raspodijeljenih u mape. Svaka mapa sadrži fotografije koje prikazuju jedno od slova abecede, isključujući slova J i Z, pošto je za njihovo prikazivanje potrebno izvođenje pokreta zbog čega ih nije moguće prikazati pomoću fotografije. Osim ovog načina, koordinate se mogu dobiti i korištenjem web-kamere pomoću koje korisnik može uslikati novu fotografiju iz koje se zatim na isti način kao i sa prije pripremljene fotografije dobivaju koordinate dlana. Za pravilan rad programa prethodno su izrađene još dvije CSV datoteke, jedna koja u svakom retku sadrži jedno slovo koje će model moći prepoznati (npr. slovo 'a' u prvom redu, slovo 'b' u drugom, itd.). Nakon što su koordinate dobivene, one se pohranjuju u CSV datoteku 'keypoints.csv'. Svaki redak te datoteke prvo sadrži indeks po kojem se određuje koje slovo prikazuju koordinate, a nakon indeksa dolaze svih 21 točka dlana, svaka sa x i y koordinatom. To znači da svaki redak CSV datoteke sadrži 43 vrijednosti, tj. indeks i 42 koordinate.

Pozicioniranjem u mapu u kojoj je program spremljen korištenjem naredbenog retka (eng. Command Prompt) te pokretanjem naredbe `python makeDatabase.py` započinje se izvođenje programa. Pokretanjem programa poziva se metoda `odaberi_vrstu_ulaza` prikazana kodom (Programski kod 2.1). Korištenjem standardne biblioteke tkinter [11] programskog jezika Python za grafičko korisničko sučelje korisniku se prikazuje dijaloški okvir koji od korisnika traži odabir jedne od triju opcija: 'Da' za korištenje već postojećih fotografija kao ulaznih podataka, 'Ne' za korištenje web-kamere, ili 'Cancel' za odustajanje

i završetak rada programa. Ovisno o odabiru korisnika, pozivaju se metode `odabir_slika` ili `odabir_kamera`. Metoda `odabir_slika` (Programski kod 2.2), koristeći biblioteku tkinter, prikazuje prozor za odabir mape s podmapama u koje su razvrstane fotografije. Odabirom putanje do mape s fotografijama poziva se metoda `ucitaj_slike` (Programski kod 2.3) koja provjerava postoji li mapa s nazivom slova koje je trenutno iščitano iz datoteke s pohranjenim slovima, te ako postoji, poziva metodu za detekciju točaka dlana i njihovih koordinata. Nakon što ta metoda završi s radom i vrati detektirane koordinate, one se zajedno s indeksom slova zapisuju u izlaznu CSV datoteku koju program stvori ako još ne postoji.

```python
# Odabir između slike i kamere
def odaberi_vrstu_ulaza(ulaz, izlaz):
    root = tk.Tk()
    root.withdraw()
    odabir = messagebox.askyesnocancel("Odaberi vrstu ulaza:", "Odaberite 'Yes' za slike,
    'No' za kameru, 'Cancel' za izlaz")
    if odabir is None:  # Cancel
        exit()
    elif odabir:        # Yes
        odabir_slika(ulaz, izlaz)
    else:               # No
        odabir_kamera(ulaz, izlaz)
```

Programski kod 2.1 Metoda odaberi_vrstu_ulaza

```python
# Odabir mape sa slikama
def odabir_slika(ulaz, izlaz):
    root = tk.Tk()
    root.withdraw()
    ulaz = filedialog.askdirectory(title = "Specificirajte put do mape sa slikama!")
    if ulaz:
        open(izlaz, "w").close()
        slova = "slova.csv"
        ucitaj_slike(ulaz, slova, izlaz)
```

Programski kod 2.2 Metoda odabir_slika

```python
# Ucitavanje i prikaz slike
def ucitaj_slike(ulaz, indeksi, izlaz):
    with open(indeksi, "r") as csv_indeksi, open(izlaz, "a", newline = "") as csv_izlaz:
        reader = csv.reader(csv_indeksi)
        writer = csv.writer(csv_izlaz)
        for ind, red in enumerate(reader):
            slovo = red[0]
            mapa = os.path.join(ulaz, slovo)
            if not os.listdir(mapa):
                continue

            for datoteka in os.listdir(mapa):
                img = os.path.join(mapa, datoteka)
                img_za_prikaz = cv2.imread(img)
                koordinate, img_koord, ind_slovo = extractKeypoints(img_za_prikaz, ind)
                if koordinate is not None:
                    writer.writerow([ind_slovo] + koordinate)
```

Programski kod 2.3 Metoda ucitaj_slike

Metoda `odabir_kamera` (Programski kod 2.4) prvo prikazuje dijaloški okvir s mogućnošću unosa teksta. Upisivanjem jednog od slova engleske abecede poziva se pomoćna metoda `slikaj` (Programski kod 2.5) koja koristeći biblioteku za računalni vid OpenCV [12] otvara web-kameru i omogućava korisniku uslikati fotografije pomoću nje. Fotografija se dobiva klikom na tipku 'space' kada je kamera otvorena te se privremeno pohranjuje kao .jpg datoteka u mapi iz koje je program pokrenut, a klikom na tipku 'Esc' zatvara se web-kamera i završava se rad programa. Nakon što je fotografija spremljena, povratkom u metodu `odabir_kamera` iz fotografije se, kao i kod već pripremljenih fotografija, detektiraju točke dlana i njihove koordinate korištenjem metode `extractKeypoints`. Nakon što ova metoda vrati listu koordinata točaka dlana, korištenjem biblioteke OpenCV korisniku se prikazuje uslikana fotografija, zajedno s detektiranim točkama dlana prikazanim na njoj. Pritiskom tipke '0' zatvara se prikaz točaka dlana, i program nastavlja s radom, spremajući koordinate dlana, zajedno s indeksom, kao jedan redak csv datoteke 'keypoints.csv'. U slučaju da nisu detektirane točke dlana na uslikanoj fotografiji, prikazuje se dijaloški okvir s tekstom 'Nisu detektirane točke dlana'.

```
def odabir_kamera(ulaz, izlaz):
    while True:
        slovo = tk.simpledialog.askstring("Odabir znaka", "Odaberite znak (slova engleske
        abecede (bez slova J i Z). Pritisnite ESC za izlaz")
        indeks = 0
        if slovo is None or slovo.lower() == "esc":
            break

        with open("slova.csv", "r") as svaSlova:
            reader = csv.reader(svaSlova)
            for ind, red in enumerate(reader):
                if red[0] == slovo.lower():
                    indeks = ind
                    break
        slikaj()
        frame = cv2.imread("tempImg.jpg")
        koordinate, img_koord, ind_slovo = extractKeypoints(frame, indeks)

        if koordinate is not None:
            cv2.imshow("Webcam Slika s koordinatama", img_koord)
            cv2.waitKey(0)
            cv2.destroyAllWindows()
            with open(izlaz, "a", newline = "") as csv_izlaz:
                writer = csv.writer(csv_izlaz)
                writer.writerow([ind_slovo] + koordinate)
        else:
            noKey = messagebox.showwarning("Detekcija koordinata ruku", "Nisu detektirane
            točke ruku")
```

Programski kod 2.4 Metoda odabir_kamera

```
# Odabrana kamera
def slikaj():
    kamera = cv2.VideoCapture(0)
    while True:
        val, frame = kamera.read()
        cv2.imshow("Webcam", frame)
        tipka = cv2.waitKey(1)

        # Pritisnuti tipku space za slikanje, Esc za izlaz
        if tipka == ord(' '):
            cv2.imwrite("tempImg.jpg", frame)
            kamera.release()
            cv2.destroyAllWindows()
            break
        elif tipka == 27:
            kamera.release()
            cv2.destroyAllWindows()
            exit()
```

Programski kod 2.5 Metoda slikaj

Metoda `extractKeypoints` (Programski kod 2.7) procesira ulaznu fotografiju, detektira točke dlana i njihove koordinate na fotografiji. Kao parametri metode predaju se

fotografija s koje se trebaju detektirati koordinate, te indeks koji predstavlja redak datoteke 'slova.csv' u kojoj je u svakom retku jedno slovo engleske abecede. Predani indeks određuje koje slovo je prikazano na fotografiji. Prije samog procesa detektiranja točaka dlana fotografija se pretvara iz formata BGR u format RGB te se horizontalno zrcali. Nakon pripreme fotografije, inicijalizira se radni okvir MediaPipe Hands za detekciju točaka dlana. Radni okvir MediaPipe Hands zahtjeva nekoliko parametara:

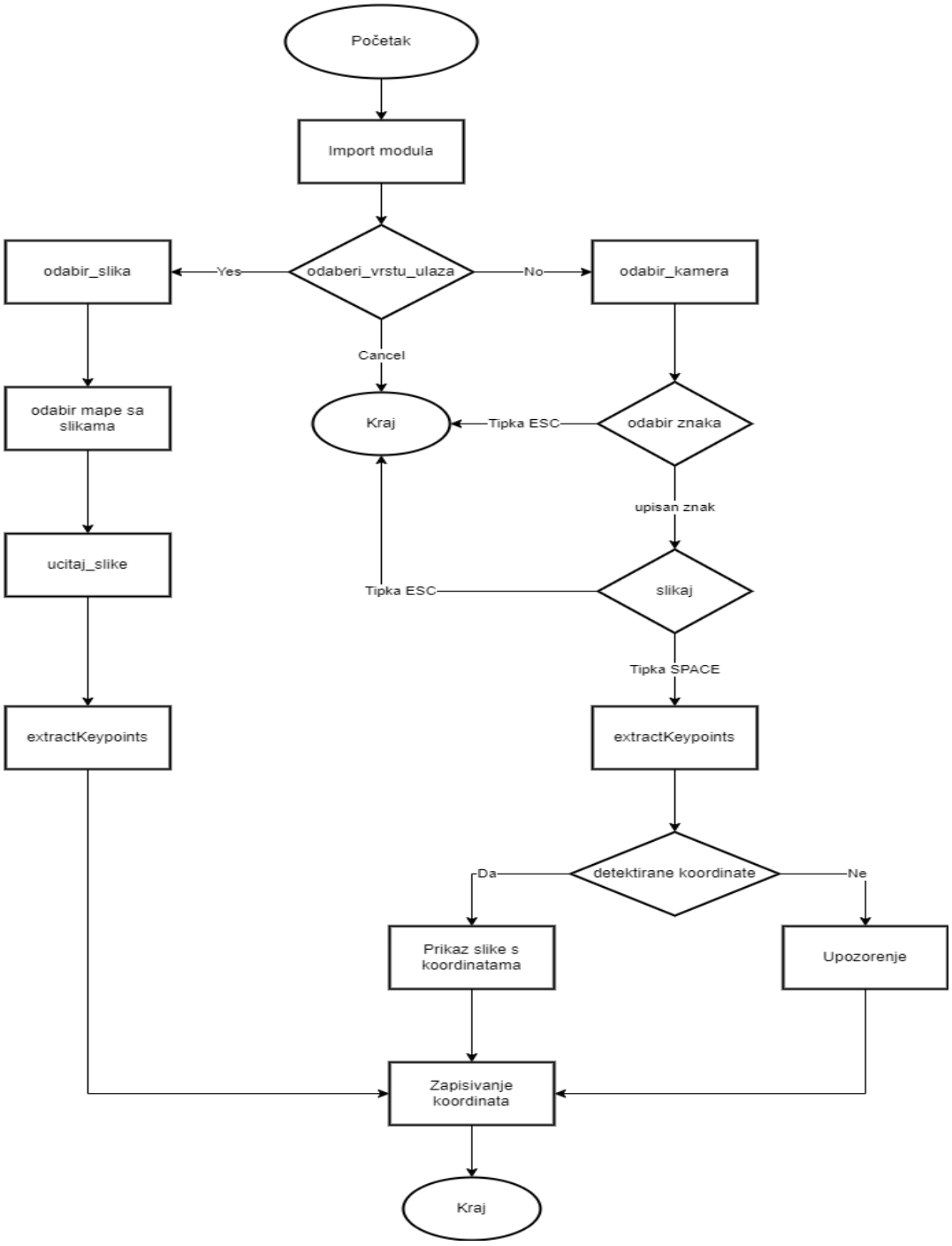
- `static_image_mode` (postavljanje na vrijednost True znači da se točke detektiraju sa fotografija)
- `max_num_hands` (maksimalan broj dlanova koje je moguće detektirati odjednom)
- `min_detection_confidence` (minimalna potrebna točnost detekcije točaka dlana, postavljena na 0.5, tj. 50% sigurnosti)

U slučaju da su sa fotografije uspješno detektirane točke dlana, prepravljanjem rezultata odbacuju se nepotrebni dijelovi, čime ostaju samo x, y i z koordinate svih 21 točaka dlana, pohranjene u jednu listu. Koordinate točaka pretvaraju se iz tipa podataka `string` u tip podataka `float` te se same koordinate korištenjem biblioteke OpenCV crtaju na fotografiju i prikazuju korisniku, zajedno s podatkom o kojoj ruci je riječ, lijevoj ili desnoj. Metoda `extractKeypoints` zatim koristi pomoćnu metodu `transformiraj` (Programski kod 2.6) koja kao parametre prima NumPy [13] listu koordinata. Pomoću ove funkcije x i y koordinate normaliziraju se tako da se prvo odrede minimalna i maksimalna vrijednost x, odnosno y koordinate te udaljenosti između minimuma i maksimuma. Zatim se koordinate normaliziraju na sljedeći način:

- od svake x koordinate oduzme se minimalna x koordinata, te se zatim rezultat podijeli širinom slike.
- od svake y koordinate oduzme se minimalna y koordinata, te se zatim rezultat podijeli visinom slike.

Normalizirane koordinate zatim se vraćaju kao rezultat metodi `extractKeypoints` koja koordinate ponovno pretvara u listu. Tu listu, zajedno sa slikom na kojoj su nacrtane koordinate te indeksom metoda `extractKeypoints` vraća metodi `ucitaj_slike`, odnosno metodi `odabir_kamera` za daljnje korištenje. Ako na fotografiji nisu detektirane točke dlana, na fotografiju se korištenjem biblioteke OpenCV iscrtava poruka

'No keypoints detected' te se vraćaju isti parametri kao i u slučaju kada su točke dlana detektirane, s razlikom u tome što se umjesto liste koordinata vraća vrijednost `None`. Tok programa 'makeDatabase.py' prikazan je na (Slika 2.1).



Slika 2.1 Tok programa makeDatabase.py

```python
def transformiraj (koord):
    minX, maxX = min(koord[::3]), max(koord[::3])
    minY, maxY = min(koord[1:3]), max(koord[1:3])
    width, height = maxX - minX, maxY - minY
    newX = (koord[::3] - minX) / width
    newY = (koord[1::3] - minY) / height
    return np.concatenate([newX, newY])
```

Programski kod 2.6 Metoda transformiraj

```python
def extractKeypoints(img, ind):
    img_rgb = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    img_rgb = cv2.flip(img_rgb, 1)
    mp_hands = mp.solutions.hands
    hands = mp_hands.Hands(
        static_image_mode = True,
        max_num_hands = 1,
        min_detection_confidence = 0.5
    )
    rezultat = hands.process(img_rgb)
    hands.close()
    img_with_keypoints = img_rgb.copy()

    try:
        if rezultat.multi_hand_landmarks:
            data = rezultat.multi_hand_landmarks[0]
            mp_drawing = mp.solutions.drawing_utils
            mp_drawing.draw_landmarks(
                img_with_keypoints,
                rezultat.multi_hand_landmarks[0],
                mp_hands.HAND_CONNECTIONS
            )
            handedness = rezultat.multi_handedness[0].classification[0].label
            cv2.putText(
                img_with_keypoints,
                handedness,
                (20, 50),
                cv2.FONT_HERSHEY_SIMPLEX,
                1,
                (0, 0, 255),
                2,
                cv2.LINE_AA
            )
            koordinate = []
            for landmark in data.landmark:
                koordinate.extend([landmark.x, landmark.y, landmark.z])
            koordinate_np = np.array(koordinate)
            transformirane = transformiraj(koordinate_np)
            return transformirane.tolist(), img_with_keypoints, ind
        else:
            cv2.putText(
                img_with_keypoints,
                "No keypoints detected",
                (20, 50),
                cv2.FONT_HERSHEY_SIMPLEX,
                1,
                (0, 0, 255),
                2,
                cv2.LINE_AA
            )
            array = [0.0] * 63
            return array, img_with_keypoints, ind
    except:
        return None, img_with_keypoints, ind
```

Programski kod 2.7 Metoda extractKeypoints

2.2. Program za izradu i treniranje modela

Model umjetne inteligencije korišten za prepoznavanje znakovnog jezika izrađen je u programskom jeziku Python programom 'train.py'. Za izradu su korištene sljedeće biblioteke:

- Tensorflow (verzija 2.15.0) – funkcionalnosti za izradu i treniranje modela umjetne inteligencije
- Pandas [14] (verzija 2.2.1) – rad s ulaznim podatcima, čitanje ulaznih podataka iz csv datoteke u DataFrame format
- NumPy (verzija 1.26.4) – efikasne matematičke operacije nad nizovima i matricama
- random [15] – slučajna podjela ulaznih podataka na dio za treniranje i dio za testiranje
- scikit-learn [16] (verzija 1.4.1) – posebno sklearn.metrics za izradu matrice konfuzije

2.2.1. Učitavanje i priprema podataka iz CSV datoteke

Program 'train.py' pokreće se pozicioniranjem u mapu u kojoj je program spremljen korištenjem naredbenog retka te upisivanjem naredbe `python train.py`. Odmah nakon pokretanja programa 'train.py', korištenjem biblioteke Pandas podatci spremljeni u datoteci 'keypoints.csv' učitavaju se u format podataka DataFrame koji koristi biblioteka Pandas. Podatci učitani u DataFrame [17] u prethodnom koraku predaju se kao parametar metodi `create_tf_data_dict` (Programski kod 2.8). Kako svaki učitani redak CSV datoteke sadrži indeks na početku, a tek nakon njega x i y koordinate točaka dlana, korištenjem indeksa kao ključa u rječniku, koordinate točaka dlana spremaju se kao vrijednost vezana za određeni ključ. Na ovaj način dobijemo rječnik s podatcima u kojem je par ključa i vrijednosti na koje se on odnosi sljedeći:

- ključ je indeks, tj. prvi element svakog retka CSV datoteke

- vrijednost je lista koja se sastoji od više unutarnjih listi, od kojih svaka sadrži x i y koordinate točaka dlana sadržane u jednom retku CSV datoteke

Odmah po završetku rada metode za stvaranje rječnika podataka `create_tf_data_dict` poziva se metoda `split_data_by_labels` (Programski kod 2.9) koja dobiveni rječnik dijeli na dva nova rječnika, jedan koji će poslužiti za treniranje modela, dok drugi služi za testiranje točnosti modela. Podjela podataka se obavlja korištenjem modula random programskog jezika Python kojim se za svaki ključ rječnika nasumično odabire određeni broj vrijednosti koji će se koristiti za testiranje. U ovoj implementaciji rječnik za treniranje sadržava 80% vrijednosti, dok se ostalih 20% nalazi u rječniku za testiranje točnosti. Nakon što su podatci podijeljeni na set za treniranje i set za testiranje točnosti, za svaki od njih poziva se metoda `extractValLabel` (Programski kod 2.10) kojoj se predaje rječnik kao parametar. Podatci koje predani rječnik sadržava dijele se u dvije liste, X_features i mapped_labels. U prvu se spremaju sve vrijednosti iz rječnika koje se prije spremanja pretvaraju u tip podataka `float`, dok se u drugu za svaku vrijednost sprema ključ rječnika kojem ta vrijednost odgovara. Nakon što su svi podatci spremljeni, obje liste se korištenjem biblioteke NumPy transformiraju u NumPy liste. Isti se postupak ponavlja i nad rječnikom za testiranje.

```
def create_tf_data_dict(data):
    tf_data_dict = {}
    for index, row in data.iterrows():
        key = row.iloc[0]
        values = row.iloc[1:].tolist()

        if key not in tf_data_dict:
            tf_data_dict[key] = []

        tf_data_dict[key].append(values)

    return tf_data_dict
```

Programski kod 2.8 Metoda create_tf_data_dict

```python
def split_data_by_label(data_dict, test_ratio=0.2):
    train_dict = {}
    test_dict = {}

    for label, data_list in data_dict.items():
        num_test_samples = int(len(data_list) * test_ratio)
        test_samples = random.sample(data_list, num_test_samples)
        train_data = [item for item in data_list if item not in test_samples]
        train_dict[label] = train_data
        test_dict[label] = test_samples

    return train_dict, test_dict
```

Programski kod 2.9 Metoda split_data_by_label

```python
def extractValLabel(dictionary):
    X_features = []
    mapped_labels = []

    for key, values_list in tf_data_dict.items():
        for values in values_list:
            float_values = [float(val) for val in values]
            X_features.append(values)
            mapped_labels.append(key) # (label_mapping[key])

    X_features = np.array(X_features, dtype =np.float32)
    mapped_labels = np.array(mapped_labels, dtype =np.float32)
    return X_features, mapped_labels
```

Programski kod 2.10 Metoda extractValLabels

2.2.2. Izrada modela

Model umjetne inteligencije koji se koristi za prepoznavanje znakovnog jezika izrađen je korištenjem umjetnih neuronskih mreža [18]. Neuronske mreže sastoje se od slojeva međusobno povezanih elemenata koje zovemo neuroni, odakle dolazi i samo ime neuronske mreže. Svaki neuron u sloju prima jedan ili više ulaza, obavlja neku jednostavnu operaciju nad njima i prosljeđuje rezultat sljedećem sloju neurona. Neuronska mreža ima jedan ulazni sloj neurona na koji se šalju ulazni podatci i jedan izlazni sloj neurona na kojem dobivamo izlazne rezultate koji mogu biti rezultati klasifikacije, regresije i sl. Osim ovih vidljivih slojeva, neuronske mreže imaju barem jedan skriveni sloj neurona u kojima se korištenjem težina obrađuju ulazni podatci. Neuronske mreže koriste se za razne zadatke poput optimizacijskih problema, problema klasifikacije i predviđanja, raspoznavanje

uzoraka, prepoznavanje i obrada slike itd. U nastavku je objašnjen dio programa kojim se kreira i trenira model umjetne inteligencije koristeći prethodno obrađene koordinate točaka dlana.

Kreiranje i treniranje modela umjetne inteligencije izvršava se programom 'train.py', a započinje odmah nakon završetka pripreme ulaznih podataka. Model se kreira korištenjem biblioteke `tf.keras.models` i metode `Sequential` [19] pomoću koje definiramo strukturu neuronske mreže koja će se koristiti. Kako se ulazni podatci sastoje od 42 koordinate, ulazni sloj neuronske mreže sastojat će se od 42 neurona, po jedan za svaku koordinatu. Prije predaje koordinata ulaznom sloju neurona, koristi se metoda `tf.keras.layers.Flatten(input_shape=(42,))`, kojom se oni pretvaraju iz dvodimenzionalnog niza u jednodimenzionalni. Drugi sloj neuronske mreže je potpuno povezan sloj koji se sastoje od 64 neurona. U ovom sloju koristi se aktivacijska funkcija Rectified Linear Unit (relu) [20] kojom se uvode nelinearnosti u model. Uvođenje nelinearnosti pomaže modelu u učenju složenih obrazaca. Za definiranje drugog sloja neuronske mreže koristi se metoda `tf.keras.layers.Dense(128, activation='relu')`. Izlazni sloj neuronske mreže definira se pomoću metode `tf.keras.layers.Dense(len(set(mapped_labels)),activation='softmax')`. Broj neurona ovog sloja jednak je broju različitih oznaka (tj. znakova engleske abecede) koje model treba moći prepoznati. Kao aktivacijska funkcija koristi se aktivacijska funkcija softmax [21] kojom se izlazi normaliziraju na raspon [0, 1], s uvjetom da zbroj svih izlaza bude jednak jedinici. Ovakvi izlazi dobri su za probleme klasifikacije gdje izlazi modela označavaju koliko je model siguran da ulazni podatci predstavljaju točno tu oznaku. Kreirani model sprema se u varijablu model i nad tom varijablom poziva se metoda `compile`. Ovom metodom model se konfigurira za treniranje definiranjem optimizacijskog algoritma, funkcije gubitka i mjerila točnosti. Kao optimizacijski model koristi se optimizacijski algoritam Adaptive Moment Estimation (Adam) [22]. Optimizacijski algoritam Adam koristi tzv. momente gradijenta (srednje vrijednosti gradijenta i kvadratnih odstupanja gradijenta) kako bi prilagodio stopu učenja. Također koristi prilagodljive stope učenja za svaku težinu čime se omogućava brža i stabilnija konvergencija. Kao funkcija gubitka odabrana je funkcija sparse_categorical_entropy koja je korisna kada su izlazi modela cijeli brojevi, kao što je slučaj u ovom programu gdje su to indeksi spremljeni u varijabli mapped_labels. Točnost kreiranog modela mjeri se pomoću vjerojatnosti točne klasifikacije ulaznih podataka. Učenje modela pokreće se metodom

`fit` koja se poziva nad varijablom u kojoj je spremljen model. Kao parametre funkciji se prosljeđuju uzorci za učenje, tj. jedan niz koordinata, oznaka vezana za taj niz koordinata, broj epoha koji označava koliko puta se cijeli skup podataka za treniranje propušta kroz neuronsku mrežu, te broj uzoraka koji se propuštaju kroz neuronsku mrežu prije ažuriranja modela. (Programski kod 2.11) prikazuje kreiranje modela i njegovu konfiguraciju te treniranje.

```
input_shape = (42,)
model = tf.keras.models.Sequential([
    tf.keras.layers.Flatten(input_shape=input_shape),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(len(set(mapped_labels)), activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
earlyStop = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=25,
restore_best_weights=True, verbose=1)
model.fit(X_features, mapped_labels, epochs=100, batch_size=32, callbacks=[earlyStop])
```

Programski kod 2.11 Kreiranje, konfiguracija i treniranje modela

Za potrebe korištenja modela unutar Android aplikacije, model je potrebno pretvoriti u oblik TensorFlow Lite. Još jedna prednost pretvaranja modela u ovaj oblik je smanjenje veličine modela. Pretvaranje modela u oblik TensorFlow Lite obavlja se metodama `tf,lite.TFLiteConverter.from_keras_model` kojoj se kao parametar prosljeđuje istrenirani model, a koja stvara konverter, te `converter.convert()` kojom se model konvertira u željeni oblik. Konvertirani model zatim se sprema u datoteku 'model.tflite'. (Programski kod 2.12) prikazuje konverziju modela u oblik TensorFlow Lite.

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
tflite_model = converter.convert()

with open('model.tflite', 'wb') as f:
    f.write(tflite_model)
```

Programski kod 2.12 Konverzija modela

2.2.3. Testiranje modela

Testiranje modela obavlja se u istom programu u kojem je model izrađen, programu 'train.py', odmah nakon spremanja modela u formatu TensorFlow Lite, a nakon testiranja prikazuje se matrica konfuzije kojom se prikazuje točnost predviđanja modela. Prvo je potrebno ponovno učitati model (ako već nije učitan u neku varijablu) te mu alocirati memoriju za tenzore. Zatim se korištenjem metoda `get_input_details()` i `get_output_details()` dobiva format ulaza i izlaza modela. Indeksi rezultata

dobiveni zajedno s ulaznim podacima za treniranje korištenjem metode `extractValLabel` spremaju se u skup koji će sadržavati samo jednu instancu svake oznake. Ovaj skup zatim se korištenjem posebnog rječnika, u kojem svaki indeks određuje jedan znak, mapira u skup oznaka znakova. Rječnik za mapiranje prikazan je odsječkom (Programski kod 2.13).

```
label_mapping = {
    0: 'a', 1: 'b', 2: 'c', 3: 'd', 4: 'e', 5: 'f', 6: 'g', 7: 'h', 8: 'i', 9: 'k', 10: 'l',
    11: 'm', 12: 'n', 13: 'nothing', 14: 'o', 15: 'p', 16: 'q', 17: 'r', 18: 's', 19: 't',
    20: 'u', 21: 'v', 22: 'w', 23: 'x', 24: 'y'
}
```

Programski kod 2.13 Rječnik za mapiranje

Prije samog testiranja modela pripremaju se dvije prazne liste, lista 'true_labels' i lista 'predicted_labels', koje će se koristiti za izradu matrice konfuzije. Prolaskom po listi koja sadrži sve uzorke za treniranje modelom se klasificiraju uzorci i dobiva se izlazna lista koja sadrži vjerojatnosti da određeni predviđeni rezultat odgovara znaku prikazanim dlanom od kojeg smo dobili ulazne podatke. Iz ove liste rezultata izdvajaju se dva rezultata s najvećom točnosti predikcije, zajedno s indeksom na kojem se oni nalaze u listi rezultata te se za svaki ulazni uzorak te vjerojatnosti ispisuju. U listu 'true_labels' dodaju se oznake željenog rezultata, dok se u listu 'predicted_labels' sprema oznaka predviđenog rezultata s najvećom točnosti. Testiranje modela prikazano je odsječkom koda (Programski kod 2.14).

```
interpreter = tf.lite.Interpreter(model_path = 'model.tflite')
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

unique_test_labels = list(set(test_labels))
unique_test_labels_mapped = [label_mapping[label] for label in unique_test_labels]

true_labels = []
predicted_labels = []

for test_input, true_label in zip(test_features, test_labels):
    test_input = np.expand_dims(test_input, axis = 0)
    interpreter.set_tensor(input_details[0]['index'], test_input)
    interpreter.invoke()

    output = interpreter.get_tensor(output_details[0]['index'])[0]
    top_indices = output.argsort()[-2:][::-1]

    best_score_idx = top_indices[0]
    best_score = output[best_score_idx]
    best_label = unique_test_labels_mapped[best_score_idx]

    second_best_score_idx = top_indices[1]
    second_best_score = output[second_best_score_idx]
    second_best_label = unique_test_labels_mapped[second_best_score_idx]

    print(f"Best Score: {best_score}, Label: {best_label}")
    print(f"Second Best Score: {second_best_score}, Label: {second_best_label}")
    print()

    true_labels.append(true_label)
    predicted_labels.append(unique_test_labels[best_score_idx])
```

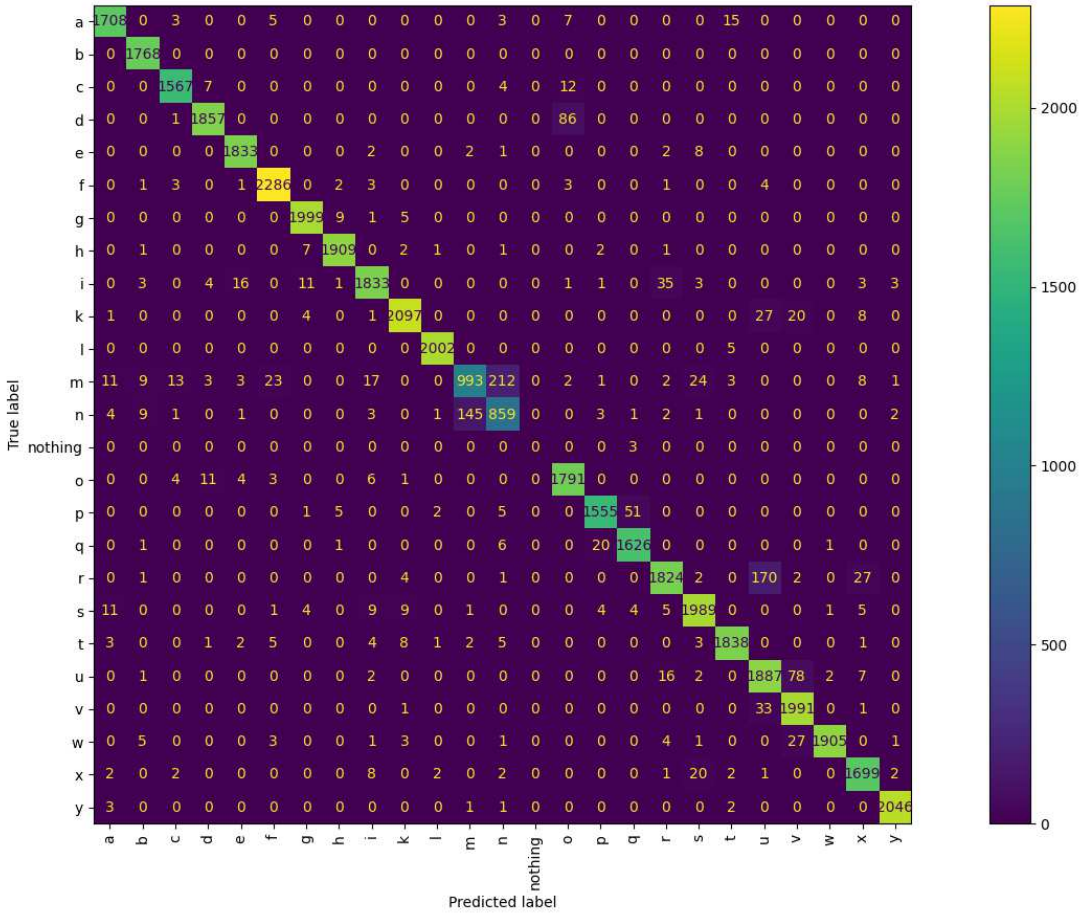
Programski kod 2.14 Testiranje modela

Izrada matrice konfuzije izvršava se korištenjem metode `confusion_matrix` biblioteke scikit-learn, no prije toga prave i predviđene oznake mapiraju se u znakove korištenjem istog rječnika koji je prikazan kodom (Programski kod 2.13). Izrađenu matricu konfuzije prikazuje se korištenjem metode `ConfusionMatrixDisplay`, također iz biblioteke scikit-learn Kod za izradu prikazan je odsječkom (Programski kod 2.15), a matrica konfuzije prikazana je na (Slika 2.2). Tok cijelog programa 'train.py' prikazan je na (Slika 2.3).

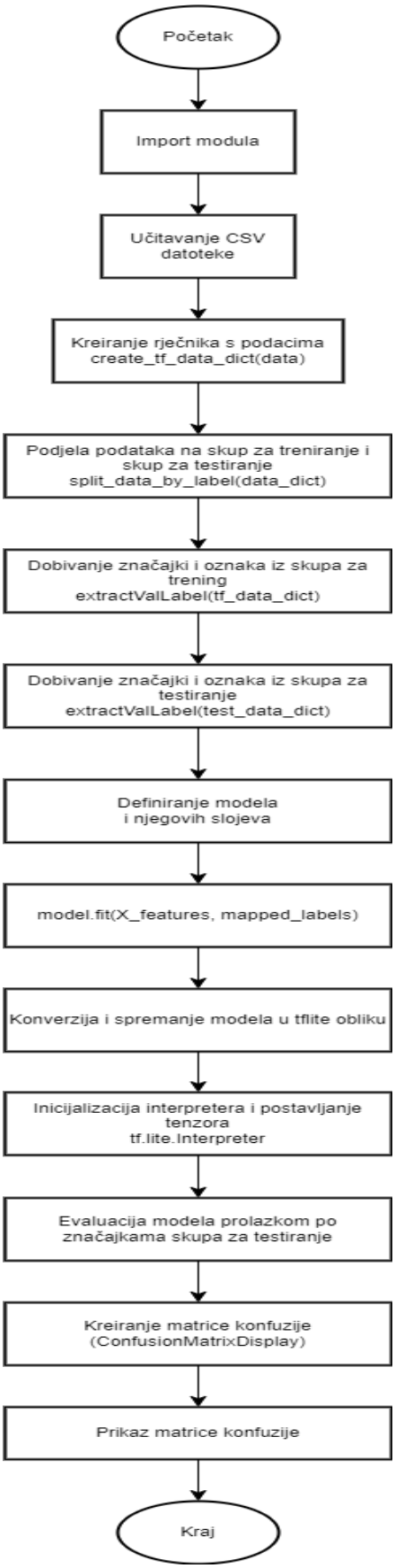
```
true_labels_mapped = [label_mapping[label] for label in true_labels]
predicted_labels_mapped = [label_mapping[label] for label in predicted_labels]
conf_matrix = confusion_matrix(true_labels_mapped, predicted_labels_mapped, labels=list
(label_mapping.values()))

disp = ConfusionMatrixDisplay(confusion_matrix=conf_matrix, display_labels=list(label_mapping.
values()))
disp.plot(include_values=True, cmap='viridis', ax=None, xticks_rotation='vertical')
plt.show()
```

Programski kod 2.15 Izrada matrice konfuzije



Slika 2.2 Matrica konfuzije modela



Slika 2.3 Tok programa train.py

3. Struktura aplikacije

U ovom poglavlju bit će detaljno opisane struktura i funkcionalnosti izrađene aplikacije za prepoznavanje znakovnog jezika. Glavni dijelovi objašnjeni u ovom poglavlju su:

- navigacija unutar aplikacije korištenjem Jetpack Compose Navigation
- prikaz liste s uputama za pravilno izvođenje pojedinog znaka engleske abecede s mogućnošću prikaza teksta na engleskom ili hrvatskom jeziku
- prepoznavanje slova engleske znakovne abecede na temelju koordinata točaka dlana dobivenih korištenjem radnog okvira MediaPipe Hands
- baza podataka Room sa spremljenim detektiranim slovom i brojem detektiranja tog slova

3.1. Početni zaslon i navigacija

Pokretanjem aplikacije prikazuje se početni zaslon koji je definiran pomoću Composable metoda Android alata za izgradnju korisničkih sučelja u jeziku Kotlin. Composable metode glavni su način kreiranja i dizajniranja različitih dijelova Android aplikacija u alatu Jetpack Compose. Ove metode moguće je kreirati unutar klasa, no uglavnom se definiraju izravno u datotekama, izvan klasa. Programski kod kojim se definira izgled početnog zaslona napisan je unutar Composable metode `MyApp`, čija je deklaracija prikazana na (**Error! Reference source not found.**). Poglavlje 3.1.1 opisuje izgled početnog zaslona prikazanog slikom (Programski kod 3.1Slika 3.1).

Slika 3.1 Prikaz početnog zaslona aplikacije

3.1.1. Izgled početnog zaslona aplikacije

Osnovna komponenta metode `MyApp`, kojom se omogućava definiranje rasporeda aplikacije je komponenta Scaffold. Njome se omogućava definiranje gornje alatne trake (eng. TopAppBar), donje navigacijske trake (eng. BottomAppBar), navigacijskog izbornika za strane (eng. Navigation Drawer), plutajućih akcijskih gumbova (eng. Floating Action Button) te središnjeg dijela za prikaz sadržaja. Od navedenih komponenti, u izrađenoj aplikaciji, uz podrazumijevani središnji dio za prikaz sadržaja, dodatno su korištene donja navigacijska traka i plutajući akcijski gumb. Donja navigacijska traka, kako joj i ime kaže, smještena je na dnu zaslona i sadrži dva gumba s ikonama (eng. IconButton) te jedan plutajući akcijski gumb u sredini između njih. Komponenta IconButton interaktivni je element sličan gumbovima, uz bitnu razliku da umjesto

klasičnog izgleda gumba umjesto teksta sadrži ikonu definiranu korištenjem tipa podatka ImageVectora. Ovisno o tome koji sadržaj je trenutno prikazan u središnjem dijelu, ikona koja odgovara tom sadržaju bijele je boje, dok su ostale ikone tamnosive. Osim ikone, gumbu je moguće definirati akciju koja se izvršava klikom na njega, što se definira parametrom onClick. Komponenta Floating Action Button gumb je koji je vizualno istaknut te također sadrži ikonu i mogućnost definiranja akcije koju je potrebno izvršiti. Za razliku od komponente IconButton, klikom na koje se mijenja sadržaj prikazan u središnjem dijelu, što je omogućeno korištenjem alata Jetpack Compose Navigation, Floating Action Button zadužen je za otvaranje kamere i provjeru potrebnih dopuštenja prije samog otvaranja kamere. (Programski kod 3.2) prikazuje definiciju komponente Floating Action Button, dok odsječak (Programski kod 3.3) prikazuje definiciju jednog IconButton-a.

```
@Composable
fun MyApp(
    dataStore: DataStore<DataAndSettings>,
    controller: LifecycleCameraController,
    classifications: List<Classification>,
    state: ListItemState,
    listItemViewModel: ListItemViewModel,
    dao: ListItemDao
) {...}
```

Programski kod 3.1 Deklaracija Composable metode MyApp

```
FloatingActionButton(
    onClick = {
        selected.value = Icons.Default.CameraAlt
        cameraPermissionResultLauncher.launch(
            Manifest.permission.CAMERA
        )
    }
) {
    Icon(
        imageVector = Icons.Default.CameraAlt,
        contentDescription = "Camera",
        tint = GreenJC
    )
}
```

Programski kod 3.2 Prikaz plutajućeg akcijskog gumba

```
IconButton(
    onClick = {
        selected.value = Icons.Default.Abc
        navController.navigate(Screen.Letters.route) { this: NavOptionsBuilder
            popUpTo( id: 0)
        }
    },
    modifier = Modifier.weight(1f)
) {
    Icon(
        imageVector = Icons.Default.Abc,
        contentDescription = "Alphabet",
        modifier = Modifier.size(26.dp),
        tint = if (selected.value == Icons.Default.Abc) {
            Color.White
        } else
            Color.DarkGray
    )
}
```

Programski kod 3.3 Prikaz komponente IconButton

3.1.2. Navigacija između zaslona u aplikaciji

Navigacija između različitih zaslona aplikacije implementirana je korištenjem alata Jetpack Compose Navigation [23]. Ovaj poseban dio alata Jetpack Compose omogućava jednostavno upravljanje navigacijom u aplikacijama koje koriste Jetpack Compose. Osnovni dijelovi potrebni za ostvarenje navigacije su NavHost, NavController i composable metode unutar NavHost-a. NavController koristi se za upravljanje navigacijom, tj. pokretanje navigacijskih operacija. To je objekt i nad njim se pozivaju metode, posebice metoda navigate kojom se prebacuje na zaslon definiran objektom koji predajemo funkciji kao parametar. Objekt NavController kreira se pri pokretanju aplikacije i predaje se metodi `MyApp` kao parametar. Komponenta NavHost dio je alata Jetpack Compose Navigation koja služi kao kontejner u kojem se definiraju odredišta (eng. route) te Composable funkcije čiji kod se izvršava navigacijom na određenu rutu korištenjem funkcije composable(). (Programski kod 3.5) prikazuje kod kojim se definira navigacija između različitih ekrana. Odredišta koja se koriste unutar NavHost-a definirana su u zasebnoj sealed klasi `Screen`. Ova klasa ima jedan parametar u konstruktoru, a to je parametar route tipa String kojim se identificira određeno odredište navigacije. Klasa `Screen` ima četiri podatkovna objekta koji predstavljaju specifično odredište. (Programski kod 3.4) prikazuje sealed klasu `Screen`.

```
sealed class Screen(
    val route: String
) {
    data object Letters : Screen( route: "letters")
    data object Camera : Screen( route: "camera")
    data object NoCamera : Screen( route: "nocamera")
    data object List : Screen( route: "list")
}
```

Programski kod 3.4 Sealed klasa Screen

```
NavHost(
    navController = navController,
    startDestination = Screen.Letters.route,
    modifier = Modifier.padding(paddingValues)
) { this: NavGraphBuilder
    composable(Screen.Letters.route) { LetterScreen(context, dataStore) }
    composable(Screen.Camera.route) { this: AnimatedContentScope  it: NavBackStackEntry
        CameraScreen(
            controller,
            classifications,
            listItemViewModel
        )
    }
    composable(Screen.NoCamera.route) { this: AnimatedContentScope  it: NavBackStackEntry
        NoCameraPermissionScreen(
            onNavigateToSettings = {
                activity.openAppSettings()
            }
        )
    }
    composable(Screen.List.route) { this: AnimatedContentScope  it: NavBackStackEntry
        ListScreen(context, state, listItemViewModel::onEvent)
    }
}
```

Programski kod 3.5 Implementacija komponente NavHost

3.2. Lista s uputama za izvedbu znakova

Sadržaj koji se prikazuje na ekranu pri pokretanju aplikacije lista je kartica sa slikom znakova engleske abecede i kratkim opisom načina izvođenja tog znaka. Kod odgovoran za izradu liste sadržan je u Composable metodi `LetterScreen`, čija deklaracija je prikazana odsječkom (Programski kod 3.7), a kod koji je odgovoran za izgled kartica u kojima su prikazani detalji svakog elementa liste definiran je Composable metodom `LetterCard`, s deklaracijom prikazanom odsječkom (Programski kod 3.6). Prije izrade

liste, u zasebnoj Kotlin datoteci definirana je podatkovna klasa `Letter` s parametrima koji se prikazuju u karticama liste. Ti parametri su resurs za sliku, naziv te opis na engleskom i hrvatskom jeziku. Osim klase `Letter`, u istoj datoteci definiran je objekt `LetterList` koji sadrži listu elemenata tipa `Letter`, koji predstavljaju znakove engleske abecede. (Programski kod 3.8) prikazuje klasu `Letter` i objekt `LetterList`, s jednim prikazanim elementom.

Lista za prikaz elemenata implementirana je korištenjem komponente LazyColumn [24] alata Jetpack Compose, a na vrhu zaslona prije liste, nalaze se dva gumba kojima se mijenja jezik opisa elemenata liste. Korištenjem komponente LazyColumn elementi liste prikazuju se vertikalno, a za svaki element izrađuje se kartica u kojoj se prikazuju podaci klase `Letter`. Komponenta LazyColumn renderira samo elemente trenutno vidljive na zaslonu, čime se poboljšavaju performanse, osobito za liste s puno elemenata, poput liste znakova engleske abecede kod koje ne stanu svi elementi na zaslon odjednom. LazyColumn definiran je kao Composable metoda, kao i većina predefiniranih komponenti alata Jetpack Compose. Kao parametar predajemo lambda izraz kojim se opisuje kako prikazati elemente liste. U slučaju ove aplikacije, koristi se metoda `itemsIndexed` te se sa svaki element liste poziva Composable metoda `LetterCard`. Metoda `LetterCard` kreira karticu na način da se na lijevoj strani korištenjem objekta tipa Painter [25] postavlja slika koja je kao resurs spremljena u mapi definiranoj metodom `painterResource(id = lett.imageRes)`. Ostatak kartice popunjavaju naslov (tj. znak) te opis izvođenja znaka, na hrvatskom ili engleskom, ovisno o trenutno odabranom jeziku.

```
@Composable
fun LetterCard(
    painter: Painter,
    title: String,
    language: Language,
    letter: Letter,
    modifier: Modifier = Modifier
) {...}
```

Programski kod 3.6 Deklaracija metode LetterCard

```
@Composable
fun LetterScreen(
    context: Context,
    dataStore: DataStore<DataAndSettings>
) {...}
```

Programski kod 3.7 Deklaracija metode LetterScreen

```
data class Letter(
    val imageRes: Int,
    val title: String,
    val descriptionEng: String,
    val descriptionHrv: String
)

object LetterList {
    val letters = listOf(...)
}
```

Programski kod 3.8 Klasa Letter i objekt LetterList

3.2.1. Promjena jezika opisa izvedbe znakova

Elementi prikazani u obliku liste korištenjem komponente LazyColumn objekti su klase `Letter` te njihov konstruktor kao parametar zahtijeva opis na engleskom i na hrvatskom. Ovakvom implementacijom omogućen je prikaz opisa za izvođenje znakova na ta dva jezika. Promjenu jezika jednostavno je implementirana na način da se iznad liste s karticama u kojima su prikazani detalji za svaki znak dodaju dva gumba. Za promjenu jezika potrebno je samo u onClick parametru gumba promijeniti jezik iz trenutnog jezika u onaj naznačen tekstom na gumbu. Za lakšu implementaciju, u zasebnoj Kotlin datoteci kreirana je enumeracija `Language` sa dva objekta, HRVATSKI i ENGLESKI. Osim enumeracije, kreirana je i podatkovna klasa (eng. data class) `DataAndSettings` čiji konstruktor prima objekt enumeracije `Language`, a ako se taj objekt ne preda, inicijalna vrijednost je HRVATSKI. Enumeracija i podatkovna klasa prikazane su odsječkom (Programski kod 3.9). Na ovaj način jednostavno je implementirana promjena jezika opisa izvedbe znakova, no čim korisnik zatvori aplikaciju, jezik opisa vratit će se na inicijalnu vrijednost, tj. HRVATSKI. Kako bi se ovaj problem riješio, potrebno je prilikom promjene jezika klikom na gumb također spremiti trenutnu vrijednost enumeracije `Language` u trajnu pohranu. Za spremanje je korištena biblioteka DataStore [26]. Korištenjem ove biblioteke vrijednost trenutnog jezika jednostavno se može spremiti i dohvatiti iz trajne pohrane. Biblioteka DataStore omogućava pohranu podataka u obliku parova ključ-vrijednost ili kao podatkovne klase korištenjem dodatka Kotlin Serialization. Za spremanje parova kluč-vrijednost koristi se Preference DataStore [27]. Vrsta biblioteke DataStore korištena u ovoj aplikaciji, Proto DataStore [28], omogućava spremanje složenijih tipova

podataka koristeći serijalizaciju. Kako bi mogli raditi s bibliotekom Proto Dana Store, potrebno ju je prvo inicijalizirati. Inicijalizacija se obavlja odmah prilikom pokretanja aplikacije, čak i prije poziva metode `MyApp`. (Programski kod 3.10) prikazuje inicijalizaciju instance Proto DataStore. Za inicijalizaciju DataStore-a potrebno je navesti ime datoteke u koju će se serijalizirani podatci spremati, u slučaju ove aplikacije to je *dataStorage.json*. Također je potrebno navesti serijalizator.

```
@Serializable
data class DataAndSettings(
    val language: Language = Language.HRVATSKI
)

@Serializable
enum class Language {
    HRVATSKI,
    ENGLESKI
}
```

Programski kod 3.9 Podatkovna klasa DataAndSettings i enumeracija Language

```
private val Context.dataStore by dataStore(
    fileName: "dataStorage.json",
    DataAndSettingsSerializer
)
```

Programski kod 3.10 Inicijalizacija Proto DataStore-a

Kako bi se omogućila seriajlizacija podatkovne klase `DataAndSettings`, potrebno ju je označiti oznakom `@Serializable`, a zbog toga što klasa `DataAndSettings` koristi objekte enumeracije `Language`, nju je također potrebno označiti istom oznakom. Zatim se u istoj datoteci kreira objekt `DataAndSettingsSerializer` koji je potrebno navesti prilikom inicijalizacije Proto DataStore-a. Serijalizator implementira parametrizirano (generičko) sučelje `Serializer<T>` [29], gdje je T tip podataka koji se serijaliziraju i deserijaliziraju. U slučaju objekta `DataAndSettingsSerializer`, parametar T zamjenjuje se podatkovnom klasom `DataAndSettings`. Zbog činjenice da je `Serializer<T>` sučelje, zahtijeva se implementacija metoda tog sučelja kako bi se objekti tipa `DataAndSettings` mogli uspješno serijalizirati i deserijalizirati. (Programski kod 3.11) prikazuje metode koje je potrebno implementirati. Prva metoda vraća zadanu vrijednost objekta tipa `DataAndSettings` koja se koristi pri prvom

pokretanju aplikacije kada još nema spremljenih vrijednosti u Proto DataStore-u. Metoda `readFrom(input: InputStream)` deserijalizira podatake. Korištenjem biblioteke Kotlin Serialization [30] i metode `decodeFromString` serijalizirani podatci spremljeni u JSON obliku pretvaraju se u objekte tipa `DataAndSettings`. U slučaju greške prilikom deserijalizacije metoda vraća inicijalnu vrijednost koju dobijemo korištenjem prve metode. Treća metoda koju je potrebno implementirati je metoda `writeTo(t: DataAndSettings, output: OutputStream)` koja serijalizira podatke i zapisuje ih u Proto DataStore korištenjem metode `encodeToString` i `encodeToByteArray`.

```
object DataAndSettingsSerializer : Serializer<DataAndSettings> {
    override val defaultValue: DataAndSettings
        get() = DataAndSettings()

    override suspend fun readFrom(input: InputStream): DataAndSettings {
        return try {
            Json.decodeFromString(
                deserializer = DataAndSettings.serializer(),
                string = input.readBytes().decodeToString()
            )
        } catch (e: SerializationException) {
            e.printStackTrace()
            defaultValue
        }
    }

    override suspend fun writeTo(t: DataAndSettings, output: OutputStream) {
        output.write(
            Json.encodeToString(
                serializer = DataAndSettings.serializer(),
                value = t
            ).encodeToByteArray()
        )
    }
}
```

Programski kod 3.11 DataAndSettingsSerializer objekt

Za praćenje trenutne vrijednosti jezika na početku metode `LetterScreen` koristi se metoda `collectAsState` kojom se iz instance biblioteke Proto DataStore prikupljaju spremljeni podatci u objekt tipa State iz objekta `dataStore.data` tipa Flow svaki puta kada se promijene podatci spremljeni u instanci biblioteke DataStore. Metoda `collectAsState` tip podatka Flow pretvara u objekt tipa State i omogućava rekompoziciju potrebnih dijelova zaslona ako se podatci u State-u promjene. Metoda

`collectAsState` također sadrži parametar `initial = DataAndSettings()` kojim se postavlja inicijalna vrijednost State-a dok nema podataka učitanih iz DataStore-a. Trenutna vrijednost jezika sprema se u varijablu `currentLang` tako da se iz objekta tipa State spremljenog u varijabli `currentLangState` dohvaća trenutna vrijednost jezika. Kako je ažuriranje podataka spremljenih u Proto DataStore-u asinkrona operacija, potrebno je koristiti korutine [31], a za to je potrebno kreirati scope koristeći `rememberCoroutineScope()`. Zatim u onClick parametru gumba za promjenu jezika koristimo scope i unutar njega ažuriramo podatke u Proto DataStore-u, što je prikazano odsječkom (Programski kod 3.12), dok je odsječkom (Programski kod 3.13) prikazano prikupljanje podataka iz DataStore-a korištenjem objekata tipa Flow i State.

```
// Promjeni jezik u engleski
Button(
    onClick = {
        currentLang = DataAndSettings(Language.ENGLESKI)
        scope.launch { this: CoroutineScope
            dataStore.updateData { it: DataAndSettings
                it.copy(language = currentLang.language)
            }
        }
    },
    colors = ButtonDefaults.buttonColors(Color.LightGray),
    shape = RoundedCornerShape(15.dp),
    modifier = Modifier
        .width(85.dp)
        .height(50.dp)
        .padding(0.dp, 0.dp, 0.dp, 5.dp)
) { this: RowScope
    Text(
        text: "ENG",
        fontSize = 15.sp,
        fontWeight = FontWeight.Bold,
        textAlign = TextAlign.Center,
        color = Color.Black
    )
}
```

Programski kod 3.12 Kod jednog gumba za mijenjanje jezika

```
val myLetters = remember { LetterList.letters }
// Zadnji odabrani jezik
val currentLangState = dataStore.data.collectAsState(
    initial = DataAndSettings()
)
var currentLang = currentLangState.value
val scope = rememberCoroutineScope()
```

Programski kod 3.13 Prikupljanje podataka iz Proto DataStore-a

3.3. Dozvole za kameru i njene funkcionalnosti

Za izvršavanje glavne funkcionalnosti izrađene aplikacije, klasifikacije znakovnog jezika u stvarnom vremenu, potrebno je u aplikaciju implementirati otvaranje kamere i prijenos kamere (eng. camera preview), tj. prikaz sadržaja dobivenog iz kamere na zaslonu ekrana. Otvaranje kamere izvršava se klikom na plutajući akcijski gumb s ikonom kamere koji se nalazi u donjoj navigacijskoj traci početnog zaslona.

3.3.1. Upravljanje dozvolama za kameru

U slučaju da aplikacija prilikom klika na plutajući akcijski gumb nema dozvolu za pristup kameri, korisniku se prikazuje skočni prozor s upitom želi li korisnik dozvoliti aplikaciji korištenje kamere. Skočni prozor sadrži sljedeće opcije:

1. Prilikom korištenja aplikacije – aplikacija ima dozvolu za korištenje kamere dok korisnik koristi aplikaciju
2. Samo ovaj put – aplikacija može koristiti kameru sve dok korisnik potpuno ne zatvori aplikaciju
3. Odbij – aplikacija ne dobiva dozvolu za korištenje aplikacije

Ako korisnik klikne na prvu opciju, aplikacija dobiva dozvolu za korištenje kamere te se otvara prijenos kamere. Jedini način na koji se nakon odabira ove opcije aplikaciji može poništiti dozvola je odlaskom na detalje o aplikaciji u postavkama uređaja. Odabir druge opcije omogućava korištenje kamere dok korisnik ne izađe i potpuno zatvori aplikaciju. Ponovnim otvaranjem aplikacije i klikom na plutajući akcijski gumb korisniku se ponovno prikazuje skočni prozor s istim opcijama.

U slučaju da korisnik odbije aplikaciji dati dozvolu za korištenje kamere, korisniku se prikazuje novi skočni prozor s porukom zašto aplikacija treba dozvolu za korištenje kamere. Osim poruke, prikazuje se i tekst 'OK'. Klikom na OK ili bilo gdje na zaslon, korisniku se ponovno prikazuje skočni prozor s prethodno navedenim opcijama. U slučaju da korisnik ponovno odbije aplikaciji dati dozvolu za korištenje kamere, prikazuje se novi skočni prozor s uputama kako dopustiti aplikaciji korištenje kamere u slučaju da korisnik to ipak želi napraviti. Pri dnu skočnog prozora prikazuje se tekst 'Grant permission' klikom na koji korisnik se preusmjerava na detalje o aplikaciji u postavkama uređaja gdje ručno može dati dozvolu za korištenje kamere. Ako korisnik i dalje ne želi dati dozvolu i vrati se natrag u aplikaciju, umjesto prijenosa kamere prikazuje se poseban zaslon s tekstom koji korisnika traži da dopusti korištenje kamere gumbom 'Grant permission' koji ga ponovno vodi u detalje o aplikaciji u postavkama uređaja. (Programski kod 3.14) prikazuje metodu `NoCameraPermissionScreen`.

```
@Composable
fun NoCameraPermissionScreen(
    onNavigateToSettings: () -> Unit
) {
    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(MaterialTheme.colorScheme.background),
        contentAlignment = Alignment.Center,
    ) { this: BoxScope
        Column() { this: ColumnScope
            Text(
                text = "Please allow camera permission to use the camera" +
                        " for sign language detection",
                fontWeight = FontWeight.Bold,
                fontSize = 16.sp,
                fontStyle = FontStyle.Italic,
                textAlign = TextAlign.Center,
                textDecoration = TextDecoration.Underline,
                color = Color.LightGray
            )
            Spacer(modifier = Modifier.height(16.dp))
            Button(
                onClick = onNavigateToSettings,
                modifier = Modifier.align(Alignment.CenterHorizontally)
            ) { this: RowScope
                Text(
                    text = "Grant Permission",
                    fontWeight = FontWeight.Bold,
                    fontSize = 16.sp,
                    textAlign = TextAlign.Center,
                    color = Color.Black
                )
            }
        }
    }
}
```

Programski kod 3.14 Metoda NoCameraPermissionScreen

Za implementaciju koraka opisanog pod rednim brojem tri korištena je biblioteka Accompanist alata Jetpack Compose. Biblioteka Accompanist [32] pruža alate i funkcionalnosti koje olakšavaju razvoj aplikacija. Neke od funkcionalnosti koje ova biblioteka pruža su podrška za animacije prilikom navigacije između različitih zaslona, olakšavanje učitavanja slika, upravljanje dozvolama, izrada složenih animacija, itd. Za upravljanje dozvolama unutar izrađene aplikacije koristi se dio biblioteke Accompanist, Accompanist Permissions [33]. Izgled skočnog prozora koji se prikazuje nakon odbijanja davanja dopuštenja definiran je Composable metodom `PermissionDialog`, čija deklaracija je prikazana odsječkom (Programski kod 3.16). Ovisno o tome je li korisnik već jednom odbio upit za dopuštenjem ili ga je odbio tek prvi puta, izgled skočnog prozora ima drugačiji opis te tekst i funkcionalnost gumba. U kodu je definirano pomoćno sučelje `PermissionText` s jednom metodom `getDescription` kojom se objektu `PermissionDialog` predaje određeni tekst. U slučaju više potrebnih dozvola, moguće je napisati više klasa koje sve implementiraju ovo sučelje, a svaka vraća opis za jednu dozvolu. U slučaju ove aplikacije potrebna je samo dozvola za korištenje kamere, pa je definirana samo jedna klasa koja implementira sučelje `PermissionText`, a to je klasa `CameraPermissionText`. Sučelje `PermissionText` i klasa `CameraPermissionText` prikazane su odsječkom (Programski kod 3.15).

```
interface PermissionText {
    fun getDescription(isPermanentlyDeclined: Boolean): String
}

class CameraPermissionText : PermissionText {
    override fun getDescription(isPermanentlyDeclined: Boolean): String {
        return if (isPermanentlyDeclined) {
            "You permanently declined camera permission. Go to the app settings to grant it."
        } else {
            "This app needs access to the camera to be able to recognize sign language."
        }
    }
}
```

Programski kod 3.15 Sučelje PermissionText i klasa CameraPermissionText

```
@Composable
fun PermissionDialog(
    permissionText: PermissionText,
    isPermanentlyDeclined: Boolean,
    onDismiss: () -> Unit,
    onOkClick: () -> Unit,
    onGoToAppSettings: () -> Unit,
    modifier: Modifier = Modifier
) {...}
```

Programski kod 3.16 Metoda Permission Dialog

Zatražene dozvole spremaju se u strukturu podataka red (eng. Queue) implementiran korištenjem `MutableStateListOf` kojom se kreira lista koja omogućuje automatsko ažuriranje kada se njen sadržaj promijeni. Kako aplikacija prilikom svog pokretanja ne traži od korisnika odmah dozvolu za korištenje kamere, red se kreira kao prazna lista. Ovaj način kreiranja reda i dodatne metode potrebne za pravilno korištenje reda implementirane su u zasebnoj klasi `PermissionViewModel` koja nasljeđuje `ViewModel` komponentu [34]. `ViewModel` služi za pohranu i upravljanje podacima povezanim s korisničkim sučeljem te omogućava da se podaci zadrže u slučaju promjena na korisničkom sučelju, npr. rotacija zaslona. Metoda `dismissDialog` uklanja prvi element iz liste spremljene u varijabli `permissionDialogQueue`, čime se simulira funkcionalnost strukture podataka red. Metoda `onPermissionResult` provjerava korisnikov odgovor na zatraženu dozvolu. Ako je korisnik odbio zatraženo dopuštenje, metoda provjerava postoji li već u redu zahtjev za istom dozvolom. Ako takvog zahtjeva nema, metoda dodaje zahtjev za tom dozvolom u red. (Programski kod 3.17) prikazuje klasu `PermissionViewModel`. Instance reda i klase `PermissionViewModel` stvaraju se u metodi `MyApp`. Nakon njihove inicijalizacije, korištenjem metode `rememberLauncherForActivityResult` koja kao rezultat daje objekt `ActivityResultLauncher` pomoću kojeg se omogućuje pokretanje zahtjeva za dozvole. Ova metoda kao parametre zahtjeva parametre contract i onResult. Za parametar contract se koristi `ActivityResultContracts.RequestPermission()` koji također označava da se traži jedna dozvola. U slučaju više dozvola, koristilo bi se `RequestPermissions()`. Parametar onResult je lambda funkcija koju se poziva kada se dobije rezultat zahtjeva za dozvolu. Ako je dozvola dobivena, korištenjem biblioteke Jetpack Compose Navigation i njenih metoda s trenutnog zaslona prelazi se na zaslon kamere i otvara se prijenos kamere. Korištenje metode

`rememberLauncherForActivityResult` prikazano je odsječkom (Programski kod 3.18).

```
class PermissionViewModel : ViewModel() {
    val permissionDialogQueue = mutableStateListOf<String>()

    fun dismissDialog() {
        permissionDialogQueue.removeFirst()
    }

    fun onPermissionResult(
        permission: String,
        isGranted: Boolean
    ) {
        if (!isGranted && !permissionDialogQueue.contains(permission)) {
            permissionDialogQueue.add(permission)
        }
    }
}
```

Programski kod 3.17 Klasa PermissionViewModel i njene metode

```
val cameraPermissionResultLauncher = rememberLauncherForActivityResult(
    contract = ActivityResultContracts.RequestPermission(),
    onResult = { isGranted ->
        permissionViewModel.onPermissionResult(
            permission = Manifest.permission.CAMERA,
            isGranted = isGranted
        )

        if (isGranted) {
            Toast.makeText(context, text: "Open Camera", Toast.LENGTH_SHORT).show()
            navController.navigate(Screen.Camera.route) { this: NavOptionsBuilder
                popUpTo( id: 0)
            }
        }
    }
)
```

Programski kod 3.18 Korištenje metode rememberLauncherForActivityResult

Klikom na plutajući akcijski gumb pokreće se zahtjev za dozvolu kamere. Zatim se za svaki zahtjev spremljen u redu kreira skočni prozor `PermissionDialog`, i to krećući od kraja reda. Od kraja reda kreće se zato što se skočni prozori nakon kreiranja prikazuju od onog koji je zadnji kreiran prema onom koji je kreiran prvi, a kako je željeni ishod prikazati prvi zahtjev u redu na početku, nužno je skočne prozore za zahtjeve kreirati od zadnjeg prema prvom u redu. U slučaju ove aplikacije nije važno kreira li se prvo skočni prozor za zadnji ili za prvi zahtjev pošto se u redu nalazi samo jedan zahtjev, onaj za

dozvolu korištenja kamere. Metoda `PermissionDialog` kao jedan od parametara traži vrijednost tipa Boolean koja provjerava je li dozvola trajno odbijena (tj. je li korisnik dva puta odbio zahtjev za dozvolom). Parametrom onOkClick od korisnika se ponovno traži dozvola kada klikne gumb 'OK' na skočnom prozoru, dok se parametrom onDismiss skočni prozor zatvara i zahtjev se miče iz reda. Zadnji parametar metode `PermissionDialog` je onGoToAppSettings koji otvara detalje o aplikaciji u postavkama telefona korištenjem pomoćne funkcije `openAppSettings` prikazane odsječkom (Programski kod 3.19). Skočni prozor se zatvara i nakon povratka u aplikaciju provjerava se je li dozvola sada odobrena pomoću `ContextCompat.checkSelfPermission`. Ako je odobrena, otvara se kamera, a inače se prikazuje poseban zaslon definiran Composable funkcijom `NoCameraPermissionScreen`, već prethodno prikazanom odsječkom (Programski kod 3.14).

```
fun Activity.openAppSettings() {
    Intent(
        Settings.ACTION_APPLICATION_DETAILS_SETTINGS,
        Uri.fromParts( scheme: "package", packageName, fragment: null)
    )//.also(::startActivity)
        .also { it: Intent
            it.data = Uri.parse( uriString: "package:$packageName")
            startActivity(it)
        }
}
```

Programski kod 3.19 Metoda openAppSettings

3.3.2. Klasifikacija korištenjem TFLite modela

Nakon korisnikovog odobrenja zahtjeva za korištenjem kamere aplikacija otvara prijenos kamere (camera preview). Funkcionalnost korištenja kamere omogućava biblioteka CameraX [35]. CameraX je biblioteka razvijena od strane Googlea koja olakšava implementaciju i korištenje raznih funkcionalnosti kamere u vlastitim Android aplikacijama. Biblioteka CameraX podržava velik broj Android verzija, uključujući mnogo starijih verzija. Osim toga, ova biblioteka jednostavno se može integrirati s raznim Googleovim alatima za strojno učenje (npr. ML Kit, MediaPipe…). Neke mogućnosti koje biblioteka CameraX podržava su prikaz fotografije u stvarnom vremenu (eng. Preview), analiza fotografije (eng. Image Analysis) i uslikavanje fotografija (eng. Image Capture). Korištenjem ove biblioteke Composable metodom `CameraPreview`, prikazanom

odsječkom (Programski kod 3.20), otvara se prijenos kamere. Izgled cijelog zaslona na kojem se otvara kamera definiran je Composable metodom `CameraScreen`, prikazanom odsječkom (Programski kod 3.21). Ova metoda poziva metodu `CameraPreview` te zatim na vrhu zaslona tijekom klasifikacije znakovnog jezika prikazuje najvjerojatniji rezultat i postotak točnosti prepoznavanja prikazanog kamerom.

```
@Composable
fun CameraPreview(
    controller: LifecycleCameraController,
    modifier: Modifier = Modifier
) {
    AndroidView(
        factory = { it: Context
            PreviewView(it).apply { this: PreviewView
                this.controller = controller
                controller.bindToLifecycle(it as LifecycleOwner)
            }
        },
        modifier = modifier
    )
}
```

Programski kod 3.20 Metoda CameraPreview

```
@Composable
fun CameraScreen(
    controller: LifecycleCameraController,
    classifications: List<Classification>,
    listItemViewModel: ListItemViewModel
) {
    val lifecycleOwner = LocalLifecycleOwner.current
    DisposableEffect(lifecycleOwner) { this: DisposableEffectScope
        controller.bindToLifecycle(lifecycleOwner)
        onDispose { controller.unbind() } ^DisposableEffect
    }

    Box(
        modifier = Modifier.fillMaxSize()
    ) { this: BoxScope
        CameraPreview(controller = controller, lifecycleOwner, Modifier.fillMaxSize())
        Column(
            modifier = Modifier.align(Alignment.TopCenter)
        ) { this: ColumnScope
            classifications.forEach { it: Classification
                Text(
                    text = it.name + " : " + it.score,
                    modifier = Modifier
                        .fillMaxWidth()
                        .background(MaterialTheme.colorScheme.primaryContainer)
                        .padding(8.dp),
                    textAlign = TextAlign.Center,
                    fontSize = 20.sp,
                    color = MaterialTheme.colorScheme.primary
                )
            }
        }
    }
}
```

Programski kod 3.21 Metoda CameraScreen

Za analizu i klasifikaciju slova američkog znakovnog jezika u stvarnom vremenu koristi se analizator koji se inicijalizira pri pokretanju aplikacije, što je prikazano odsječkom (Programski kod 3.22). Analizator je objekt tipa `SignLangImageAnalyzer`, što je klasa u kojoj su definirane metode analizatora. Deklaracija klase `SignLangImageAnalyzer` prikazana je odsječkom (Programski kod 3.24). Ova klasa nasljeđuje sučelja `ImageAnalysis.Analyze` [36] te `HandLandmarkerHelper.LandmarkerListener`. Prvo od ova dva sučelja je sučelje biblioteke CameraX koje omogućava analizu slika u stvarnom vremenu, dok je drugo sučelje definirano u klasi `HandLandmarkerHelper`, a koje definira metode za rad s radnim okvirom MediaPipe Hands, tj. rezultatima prepoznavanja dlanova i greškama koje se mogu pojaviti. (Programski kod 3.23) prikazuje sučelje `LandmarkerListener`.

```
val analyzer = remember {
    SignLangImageAnalyzer(
        classifier = TfLiteSignLangClassifier(
            context = applicationContext
        ),
        onResults = { it: List<Classification>
            classifications = it
        },
        listItemViewModel = listItemViewModel,
        applicationContext
    )
}
```

Programski kod 3.22 Inicijalizacija analizatora

```
interface LandmarkerListener {
    fun onError(error: String, errorCode: Int = OTHER_ERROR)
    fun onResults(resultBundle: ResultBundle)
}
```

Programski kod 3.23 Sučelje LandmarkerListener

```
class SignLangImageAnalyzer(
    private val classifier: SignLangClassifier,
    private val onResults: (List<Classification>) -> Unit,
    private val listItemViewModel: ListItemViewModel,
    private val context: Context
) : ImageAnalysis.Analyzer, HandLandmarkerHelper.LandmarkerListener {
```

Programski kod 3.24 Klasa SignLangImageAnalyzer

Kako su `ImageAnalysis.Analyze` i `LandmarkListener` sučelja, potrebno je implementirati njihove metode, a to su metoda `analyze` sučelja `ImageAnalysis.Analyze` te metode `onResults` i `onError` sučelja `LandmarkListener`. Metoda `analyze`, prikazana odsječkom (Programski kod 3.25), prima parametar tipa ImageProxy koji predstavlja sliku ili video frame dobiven kamerom u stvarnom vremenu. Metoda sprema vrijeme kad je slika dobivena te, ako je od prošle analize prošlo više od dvije sekunde, analizira sliku pozivanjem metode `detectLiveStream` klase `HandLandmarkerHelper`. U slučaju da nije prošlo dovoljno vremena od zadnje analize, metode preskače analizu i čeka novi ulazni frame.

```
override fun analyze(image: ImageProxy) {
    val currentTime = System.currentTimeMillis()
    if (currentTime - lastAnalyzedTime >= analysisInterval) {
        lastAnalyzedTime = currentTime
        handLandmarkerHelper?.detectLiveStream(
            image,
            isFrontCamera: CameraSelector.LENS_FACING_FRONT == CameraSelector.LENS_FACING_FRONT
        )
    } else {
        image.close()
    }
}
```

Programski kod 3.25 Metoda analyze

Metoda `onResults` kao parametar prima instancu podatkovne klase `ResultBundle`, također definirane u klasi `HandLandmarkerHelper`. `ResultBundle` je klasa u kojoj su spremljeni rezultati korištenja radnog okvira MediaPipe Hands, tj. ona sadrži koordinate točaka dlana te ostale podatke dobivene korištenjem MediaPipe Hands radnog okvira. Koordinate točaka dlana normaliziraju se korištenjem metode `normalizeLandmarks` te se predaju modelu TensorFlow Lite kao ulazi, a model zatim klasificira te ulaze koristeći metodu `classify` klasifikatora tipa `SignLangClassifier`. Podatkovna klasa `ResultBundle` prikazana je odsječkom (Programski kod 3.26), a metoda `onResults` i metoda `onError` odsječkom (Programski kod 3.27). Osim implementiranih metoda sučelja, klasa `SignLangImageAnalyzer` sadrži pomoćnu metodu `flattenLandmarks`, koja se koristi za izdvajanje koordinata iz točaka dlana. (Programski kod 3.28) prikazuje metode `normalizeLandmarks` i `flattenLandmarks`.

```
data class ResultBundle(
    val results: List<HandLandmarkerResult>,
    val inferenceTime: Long,
    val inputImageHeight: Int,
    val inputImageWidth: Int,
)
```

Programski kod 3.26 Podatkovna klasa ResultBundle

```
override fun onResults(resultBundle: HandLandmarkerHelper.ResultBundle) {
    if (resultBundle.results.isNotEmpty()) {
        val landmarks = resultBundle.results.first().landmarks().flatMap { it }
        if (landmarks.isNotEmpty()) {
            val normalizedLandmarks = normalizeLandmarks(landmarks)
            val flattenLandmarks = flattenLandmarks(normalizedLandmarks)
            val (mostProbableLabel, index) = classifier.classify(flattenLandmarks.toFloatArray())
            val classifications = listOf(
                Classification(
                    mostProbableLabel,
                    index
                )
            )
            onResults(classifications)
            listItemViewModel.addOrUpdateClassification(
                name: classifications.firstOrNull()?.name ?: "Unknown"
            )
        } else {
            println("No landmarks")
        }

    }
}

override fun onError(error: String, errorCode: Int) {
    Log.e( tag: "SignLangImageAnalyzer",  msg: "Error: $error (code: $errorCode)")
}
```

Programski kod 3.27 Metode onResults i onError

```
private fun flattenLandmarks(landmarks: List<Pair<Float, Float>>): List<Float> {
    return landmarks.flatMap { listOf(it.first, it.second) }
}

private fun normalizeLandmarks(landmarks: List<NormalizedLandmark>): List<Pair<Float, Float>> {
    val minX = landmarks.minOf { it.x() }
    val maxX = landmarks.maxOf { it.x() }
    val minY = landmarks.minOf { it.y() }
    val maxY = landmarks.maxOf { it.y() }
    val width = maxX - minX
    val height = maxY - minY

    return landmarks.map { landmark ->
        val normalizedX = (landmark.x() - minX) / width
        val normalizedY = (landmark.y() - minY) / height
        Pair(normalizedX, normalizedY) *map
    }
}
```

Programski kod 3.28 Metode flattenLandmarks i normalizeLandmarks

Klasifikator kojim se klasificiraju koordinate točaka dlana dobivene analizatorom definiran je klasom `TfLiteSignLangClassifier`, a njegova deklaracija prikazana je odsječkom (Programski kod 3.29). Ova klasa nasljeđuje sučelje `SignLangClassifier`, prikazano odsječkom (Programski kod 3.30), koje definira jednu metodu, metodu `classify`. Klasifikator za klasifikaciju znakovnog jezika koristi

TensorFlow Lite interpreter koji se inicijalizira učitavanjem modela TensorFlow Lite i tekstualne datoteke s oznakama (labelama) koje predstavljaju moguće rezultate klasifikacije. Za učitavanje modela TensorFlow Lite i datoteke sa oznakama definirane su dvije pomoćne metode `loadModelFile` i `loadLabels`, prikazane odsječkom (Programski kod 3.31). Metoda `classify` obrađuje koordinate i na temelju njih predviđa najvjerojatniji rezultat. Kao parametar metodi `classify` predaje se niz koji sadrži x i y koordinate točaka dlana. Koordinate se učitavaju u ulazni međuspremnik (eng. inputBuffer), dok se u izlazni međuspremnik (eng. outputBuffer) spremaju rezultati klasifikacije. U poseban niz spremaju se dobivene vjerojatnosti svake oznake koje se zatim filtriraju tako da se dobije najvjerojatniji rezultat. Taj rezultat, zajedno sa predviđenom točnošću i indeksom koji određuje poziciju u rezultatnoj listi, vraća se kao tip podatka `Pair` metodi `analyze`. (Programski kod 3.32) prikazuje metodu `classify`.

```
class TfLiteSignLangClassifier(
    private val context: Context,
    private val threshold: Float = 0.7f,
    private val maxResults: Int = 1
) : SignLangClassifier {
```

Programski kod 3.29 Klasa TfLiteSignLangClassifier

```
interface SignLangClassifier {
    fun classify(landmarks: FloatArray): List<Classification>
}
```

Programski kod 3.30 Sučelje SignLangClassifier

```
private fun loadModelFile(context: Context): MappedByteBuffer {
    val assetFileDescriptor = context.assets.openFd( fileName: "model.tflite")
    val inputStream = FileInputStream(assetFileDescriptor.fileDescriptor)
    val fileChannel = inputStream.channel
    val startOffset = assetFileDescriptor.startOffset
    val declaredLength = assetFileDescriptor.declaredLength
    return fileChannel.map(FileChannel.MapMode.READ_ONLY, startOffset, declaredLength)
}

private fun loadLabels(context: Context): List<String> {
    return context.assets.open( fileName: "labels.txt").bufferedReader().useLines { it.toList() }
}
```

Programski kod 3.31 Metode loadModelFile i loadLabels

```
override fun classify(landmarks: FloatArray): Pair<String, Float> {
    val inputBuffer = TensorBuffer.createFixedSize(
        intArrayOf(1, 42),
        DataType.FLOAT32
    )
    inputBuffer.loadArray(landmarks)
    val outputBuffer = TensorBuffer.createFixedSize(
        intArrayOf(1, labels.size),
        DataType.FLOAT32
    )
    interpreter?.run(inputBuffer.buffer, outputBuffer.buffer.rewind())
    val outputArray = outputBuffer.floatArray

    var maxScore = -1f
    var maxIndex = -1
    for (i in outputArray.indices) {
        if (outputArray[i] > maxScore) {
            maxScore = outputArray[i]
            maxIndex = i
        }
    }
    val mostProbableLabel = if (maxIndex != -1) labels[maxIndex] else "Unknown"
    return Pair(mostProbableLabel, maxScore)
}
```

Programski kod 3.32 Metoda classify

Inicijalizacija objekta tipa `HandLandmarker` koji se koristi za detekciju točaka dlana izvršava se u klasi `HandLandmarkerHelper` [37]. Klasa i inicijalizacija `HandLandmarker` objekta prikazane su odsječkom (Programski kod 3.33). Inicijalizacija se izvršava metodom `setupHandLandmarker` koja se poziva u bloku `init`, a prikazana je odsječkom (Programski kod 3.35). Ova metoda podešava potrebne opcije objekta `HandLandmarker` te određuje putanju do modela za detekciju točaka dlana. Na kraju se kreira objekt tipa `HandLandmarker` pomoću konteksta i podešenih opcija. U slučaju greške pri inicijalizaciji objekta ispisuje se poruka greške. Za detekciju točaka ruku u stvarnom vremenu koristi se metoda `detectLiveStream`, prikazana odsječkom (Programski kod 3.34). Metoda prvo provjerava trenutni način rada, i u slučaju da je on drugačiji od traženog načina rada (način LIVE_STREAM), dojavljuje se greška. Inače se učitava slika, tj. frame, dobiven metodom `analyze` analizatora tipa `SignLangImageAnalyzer`, te se taj frame rotira i skalira ako je to potrebno. Na kraju se frame pretvara u objekt tipa MPImage i poziva se metoda `detectAsync`, prikazana

odsječkom (Programski kod 3.36), koja asinkrono detektira dlan i točke dlana iz objekta MPImage.

```
class HandLandmarkerHelper(
    var minHandDetectionConfidence: Float = DEFAULT_HAND_DETECTION_CONFIDENCE,
    var minHandTrackingConfidence: Float = DEFAULT_HAND_TRACKING_CONFIDENCE,
    var minHandPresenceConfidence: Float = DEFAULT_HAND_PRESENCE_CONFIDENCE,
    var maxNumHands: Int = DEFAULT_NUM_HANDS,
    var currentDelegate: Int = DELEGATE_CPU,
    var runningMode: RunningMode = RunningMode.IMAGE,
    val context: Context,
    val handLandmarkerHelperListener: LandmarkerListener? = null
) {

    private var handLandmarker: HandLandmarker? = null

    init {
        setupHandLandmarker()
    }
```

Programski kod 3.33 Klasa HandLandmarkerHelper i inicijalizacija HandLandmarker objekta

```
fun detectLiveStream(imageProxy: ImageProxy) {
    if (runningMode != RunningMode.LIVE_STREAM) {
        throw IllegalArgumentException(
            "Attempting to call detectLiveStream while not" +
                    " using RunningMode.LIVE_STREAM"
        )
    }
    val frameTime = SystemClock.uptimeMillis()
    val bitmapBuffer = imageProxyToBitmap(imageProxy)
    imageProxy.close()

    val matrix = Matrix().apply { this: Matrix
        postRotate(imageProxy.imageInfo.rotationDegrees.toFloat())
    }
    val rotatedBitmap = Bitmap.createBitmap(
        bitmapBuffer,
        x: 0,
        y: 0,
        bitmapBuffer.width,
        bitmapBuffer.height,
        matrix,
        filter: true
    )
    val mpImage = BitmapImageBuilder(rotatedBitmap).build()
    detectAsync(mpImage, frameTime)
}
```

Programski kod 3.34 Metoda detectLiveStream

```
fun setupHandLandmarker() {                                    ▲2 ▲6 ✓11
    val baseOptionBuilder = BaseOptions.builder()
    when (currentDelegate) {
        DELEGATE_CPU -> baseOptionBuilder.setDelegate(Delegate.CPU)
        DELEGATE_GPU -> baseOptionBuilder.setDelegate(Delegate.GPU)
    }
    baseOptionBuilder.setModelAssetPath(MP_HAND_LANDMARKER_TASK)
    when (runningMode) {
        RunningMode.LIVE_STREAM -> {
            if (handLandmarkerHelperListener == null) {
                throw IllegalStateException("handLandmarkerHelperListener must be set when" +
                        " runningMode is LIVE_STREAM.")
            }
        }

        else -> {}
    }

    try {
        val baseOptions = baseOptionBuilder.build()
        val optionsBuilder = HandLandmarker.HandLandmarkerOptions.builder()
            .setBaseOptions(baseOptions)
            .setMinHandDetectionConfidence(minHandDetectionConfidence)
            .setMinTrackingConfidence(minHandTrackingConfidence)
            .setMinHandPresenceConfidence(minHandPresenceConfidence)
            .setNumHands(maxNumHands)
            .setRunningMode(runningMode)

        if (runningMode == RunningMode.LIVE_STREAM) {
            optionsBuilder
                .setResultListener(this::returnLivestreamResult)
                .setErrorListener(this::returnLivestreamError)
        }
        val options = optionsBuilder.build()
        handLandmarker = HandLandmarker.createFromOptions(context, options)
    } catch (e: Exception) {
        handLandmarkerHelperListener?.onError(
            error: "Hand Landmarker failed to initialize. See error logs for details",
            if (e is IllegalStateException) OTHER_ERROR else GPU_ERROR
        )
        Log.e(TAG,  msg: "MediaPipe failed to load the task with error: ${e.message}")
    }
}
```

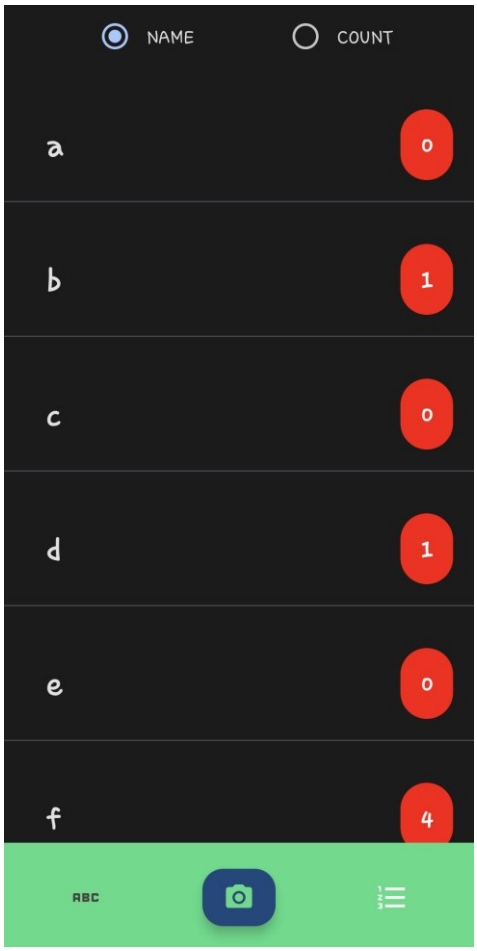
Programski kod 3.35 Metoda setupHandLandmarker

```
@VisibleForTesting
fun detectAsync(mpImage: MPImage, frameTime: Long) {
    handLandmarker?.detectAsync(mpImage, frameTime)
}
```

Programski kod 3.36 Metoda detectAsync

3.4. Room baza rezultata klasifikacije

U prethodnim poglavljima opisana su dva od tri gumba s ikonama koji se nalaze u donjoj navigacijskoj traci komponente `Scaffold` u metodi `MyApp`. Preostali gumb s ikonom omogućava navigaciju na zaslon s prikazanom listom svih slova engleske abecede koje izrađeni model TensorFlow Lite može prepoznati, zajedno s brojem na desnoj strani elementa liste koji označava broj dosadašnjih prepoznavanja određenog slova. Osim prikaza liste implementiranog korištenjem komponente LazyColumn, omogućeno je brisanje cijelog elementa iz liste povlačenjem tog elementa u lijevo. Brisanjem elementa iz liste, njegov broj prepoznavanja postavlja se na nulu, a element nestaje iz komponente LazyColumn i ponovno se prikazuje tek nakon što je ponovno prepoznat korištenjem modela ili nakon ponovnog ulaska u aplikaciju. Osim prikaza elemenata, pri vrhu zaslona prikazuju se dva gumba za odabir pomoću kojih se prikazana lista sortira po nazivu (tj. abecedno) ili po broju prepoznavanja silazno. Dizajn opisanog zaslona prikazan je na (Slika 3.2).



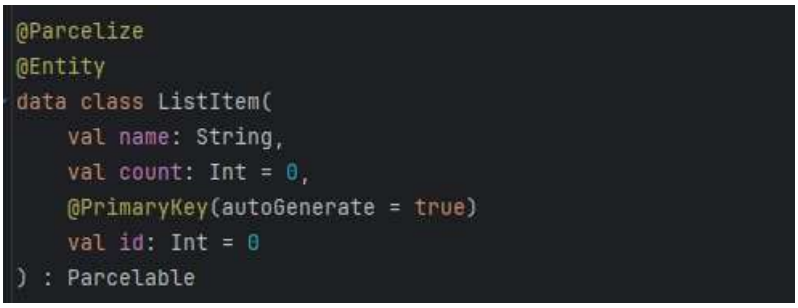
Slika 3.2 Zaslon s prikazom rezultata

Kako se podaci prepoznati korištenjem modela TensorFlow Lite ne bi izgubili nakon zatvaranja aplikacije, korištena je biblioteka Room Database [38] za spremanje podataka. Room Database biblioteka je za upravljanje bazom podataka u Android aplikacijama. Biblioteka olakšava rad s bazama podataka i smanjuje mogućnost pojave grešaka ako se ispravno implementira. Rad s bazama podataka pojednostavljuje se korištenjem posebnih anotacija (npr. `@Entity`, `@Database`, `@Dao`…) te korištenjem posebnog objekta za definiranje svih metoda kojima se pristupa bazi. Ovaj poseban objekt zovemo Data Access Object (skraćeno DAO). DAO omogućava jednostavno korištenje i definiranje operacija za rad s podacima spremljenim u bazi podataka, također koristeći anotacije (npr. `@Insert`, `@Update`, `@Delete`, `@Transaction`, `@Query`…).

Za korištenje Room baze podataka definirana je apstraktna klasa `ListItemDB` koja nasljeđuje klasu `RoomDatabase` te definira DAO objekt. Ova klasa označena je anotacijom `@Database` koja označava da se radi o bazi podataka. Anotacijom se također definira koje tablice sadrži baza te koja je trenutna verzija baze. Podatkovna klasa `ListItem`, označena anotacijom `@Entity`, predstavlja tablicu baze podataka, a svaka instanca klase predstavlja jedan redak tablice, tj. jedan entitet. Parametri konstruktora te klase označavaju stupce tablice (tj. atribute). Za kreiranje jednog entiteta (instance klase `ListItem`) potrebni su parametri 'name' i 'count', dok je parametar 'id' primarni ključ tablice. Ovaj parametar označen je anotacijom `@PrimaryKey(autoGenerate = True)`, što znači da će ga baza podataka automatski generirati. (Programski kod 3.37) i (Programski kod 3.38) prikazuju klasu `ListItemDB`, odnosno klasu `ListItem`. Osim anotacije `@Entity`, podatkovna klasa `ListItem` anotirana je i sa `@Parcelize` te implementira sučelje `Parcelable` [39] koje omogućuje serijalizaciju i deserijalizaciju objekata tipa `ListItem` na učinkovit način te njihov prijenos između aktivnosti i različitih komponenti Android aplikacija. Označavanjem klase `ListItem` anotacijom `@Parcelize` automatski se generiraju sve potrebne metode sučelja `Parcelable`.

```
@Database(
    entities = [ListItem::class],
    version = 1
)
abstract class ListItemDB : RoomDatabase() {
    abstract val dao: ListItemDao
}
```

Programski kod 3.37 Abstraktna klasa ListItemDB



```
@Parcelize
@Entity
data class ListItem(
    val name: String,
    val count: Int = 0,
    @PrimaryKey(autoGenerate = true)
    val id: Int = 0
) : Parcelable
```

Programski kod 3.38 Podatkovna klasa ListItem

Sve metode koje rade s podacima sadržanim u Room bazi podataka definirane su sučeljem `ListItemDao` koje je označeno anotacijom `@Dao` kojom se označava da je to sučelje DAO baze podataka. Metode definirane ovim sučeljem služe za umetanje, brisanje i dohvaćanje objekata tipa `ListItem` iz baze. Za dohvaćanje podataka postoje dvije metode, `getListItemsOrderedByName` kojom se objekti dohvaćaju na temelju atributa 'name' te `getListItemsOrderedByCount` kojom se objekti dohvaćaju na temelju atributa 'count' Metoda `upsertListItem` omogućava umetanje objekta tipa `ListItem` u bazu podataka, a u slučaju da objekt već postoji u bazi, umjesto dodavanja metoda ažurira postojeći objekt. Metoda `deleteListItem` jednostavno briše objekt iz baze podataka. Osim ovih metoda, definirana je i metoda `addOrUpdateListItem` anotirana sa `@Transaction`. Ova anotacija označava da definirana metoda koristi transakciju za ažuriranje podataka spremljenih u bazi podataka. Na ovaj način sve operacije nad podacima izvršavaju se kao jedna cjelina te se u slučaju greške tijekom transakcije poništavaju sve odrađene operacije i stanje u bazi podataka bit će jednako stanju baze prije izvršavanja funkcije `addOrUpdateListItem`. Ovom metodom pretražuje se baza podataka te se pokušava pronaći objekt s atributom 'name' jednak parametru 'name' objekta tipa `ListItem` predanog metodi kao parametar. Pretraživanje se obavlja korištenjem metode `getItemByName` označenom anotacijom `@Query`. Ova anotacija definira upit bazi podataka za određenim podacima, u ovom slučaju entitetom s traženim atributom 'name'. U slučaju da takav objekt ne postoji u bazi podataka, korištenjem metode `upsertListItem` objekt se dodaje u bazu. Ako je objekt s jednakim atributom 'name' pronađen u bazi, stvara se kopija pronađenog objekta, s time da se atribut 'count' uvećava za jedan. Ovo se postiže metodom `copy`. Nakon stvaranja kopije objekta iz baze, isti taj objekt korištenjem metode `upsertListItem` ažurira se pomoću stvorene kopije. Na ovaj način jednostavno i brzo se može objektu u bazi podataka

promijeniti atribut 'count' kada je to potrebno. Sučelje `ListItemDao` prikazano je odsječkom (Programski kod 3.39).

```
@Dao
interface ListItemDao {
    @Upsert
    suspend fun upsertListItem(listItem: ListItem)

    @Delete
    suspend fun deleteListItem(listItem: ListItem)

    @Query("SELECT * FROM listitem ORDER BY name ASC")
    fun getListItemsOrderedByName(): Flow<List<ListItem>>

    @Query("SELECT * FROM listitem ORDER BY count DESC")
    fun getListItemsOrderedByCount(): Flow<List<ListItem>>

    // Custom method to add or update a ListItem
    @Transaction
    suspend fun addOrUpdateListItem(item: ListItem) {
        val existingItem = getItemByName(item.name)
        if (existingItem == null) {
            // Item does not exist, insert new item
            upsertListItem(item)
        } else {
            // Item exists, increment its count
            val updatedItem = existingItem.copy(count = existingItem.count + 1)
            upsertListItem(updatedItem)
        }
    }

    // Query to get item by name
    @Query("SELECT * FROM listitem WHERE name = :name LIMIT 1")
    suspend fun getItemByName(name: String): ListItem?
}
```

Programski kod 3.39 Sučelje ListItemDao i njegove metode

Sealed sučeljem `ListItemEvent` definirane su akcije koje se izvršavaju kada korisnik obavi nekakvu interakciju s aplikacijom, npr. klik na gumb za sortiranje podataka u bazi. Aplikacija reagira na tri događaja, od kojih su dva klik na određeni gumb za sortiranje liste, a treći je spremanje objekta u bazu podataka. Za jednostavnije određivanje načina sortiranja definirana je enumeracija `SortType` s dvije vrijednosti: NAME i COUNT. (Programski kod 3.40) prikazuje sučelje `ListItemEvent` i enumeraciju `SortType`. Klasom `ListItemViewModel`, koja nasljeđuje `ViewModel`, povezuju se metode sučelja `ListItemDao` i akcija aktiviranih interakcijom korisnika s aplikacijom, definiranih sučeljem `ListItemEvent`. Kako `ListItemViewModel` klasa koristi `ListItemDao` sučelje i njegove metode, omogućava lakšu komunikaciju s bazom podataka. Prilikom same inicijalizacije objekta klase `ListItemViewModel`, korištenjem korutina poziva se metoda `initializeListItems` kojom se inicijaliziraju elementi liste (tj. slova abecede) zajedno sa svojim brojačima prepoznavanja.

Brojači se postavljaju na onu vrijednost spremljenu u bazi podataka, a u slučaju da tog objekta nema u bazi, postavljaju se na nulu. Dio kojim se inicijaliziraju stavke liste iz baze podataka prikazan je odsječkom (Programski kod 3.42). Nakon inicijalizacije liste, kreira se varijabla `_sortType` tipa `MutableStateFlow(SortType)` kojom se prati trenutni način sortiranja liste, `_listItems` tipa koja sadržava listu elemenata koja se mijenja ovisno o trenutnom načinu sortiranja te varijablu `_state` tipa `MutableStateFlow(ListItemState())` kojom se prati trenutno stanje elemenata liste. Klasa `ListItemState` je klasa koja definira trenutno stanje elemenata liste koja se prikazuje. Kada dođe do promjena (npr. promjena načina sortiranja, dodavanje novog elementa u listu…), korištenjem klase `ListItemState` ažurira se prikazana lista i novi podaci prikazuju se na zaslonu. Klasa `ListItemState` prikazana je odsječkom (Programski kod 3.41).

```
sealed interface ListItemEvent {
    object SaveListItem : ListItemEvent
    data class SortListItems(val sortType: SortType) : ListItemEvent
    data class DeleteListItem(val listItem: ListItem) : ListItemEvent
}

enum class SortType {
    NAME,
    COUNT
}
```

Programski kod 3.40 Sučelje ListItemEvent i enumeracija SortType

```
data class ListItemState(
    val listItems: List<ListItem> = emptyList(),
    val name: String = "",
    val count: Int = 0,
    val sortType: SortType = SortType.NAME
)
```

Programski kod 3.41 Podatkovna klasa ListItemState

Osim varijable `_state`, kreira se i varijabla `state` koja kombinira više tokova (objekti tipa Flow) u jedan tok koji se koristi za ažuriranje korisničkog sučelja kada dođe do promjene stanja. Metoda `onEvent` kojoj se kao parametar predaje objekt tipa `ListItemEvent` koji označava neki događaj definiran u klasi `ListItemEvent`, obrađuje predani događaj i na temelju njega izvršava određenu akciju korištenjem metoda sučelja `ListItemDao`. (Programski kod 3.43) prikazuje kreiranje tokova podataka

_sortType, _listItems, _state i state. Metoda addOrUpdateClassification pomoćna je metoda koja, koristeći metodu addOrUpdateListItem, ažurira atribut 'count' entiteta baze podataka određenog parametrom 'name' predanom ovoj metodi. Prikaz metoda onEvent i addOrUpdateClassification prikazan je odsječkom (Programski kod 3.44).

```
@OptIn(ExperimentalCoroutinesApi::class)
class ListItemViewModel(
    private val dao: ListItemDao
) : ViewModel() {
    init {
        viewModelScope.launch { this: CoroutineScope
            initializeListItems()
        }
    }

    private suspend fun initializeListItems() {
        val alphabet = ('A' ≤ .. ≤ 'Z').map { it.toString() }
        alphabet.forEach { letter ->
            // Check if item with this name already exists
            val existingItem = dao.getItemByName(letter)
            if (existingItem == null) {
                dao.addOrUpdateListItem(ListItem(name = letter, count = 0))
            }
        }
    }
```

Programski kod 3.42 Inicijalizacija elemenata liste iz baze podataka

```
private val _sortType = MutableStateFlow(SortType.NAME)
private val _listItems = _sortType
    .flatMapLatest { sortType ->
        when (sortType) {
            SortType.NAME -> dao.getListItemsOrderedByName() ^flatMapLatest
            SortType.COUNT -> dao.getListItemsOrderedByCount() ^flatMapLatest
        }
    }.stateIn(viewModelScope, SharingStarted.WhileSubscribed(), emptyList())

private val _state = MutableStateFlow(ListItemState())
val state = combine(_state, _sortType, _listItems) { state, sortType, listItems ->
    state.copy(
        listItems = listItems,
        sortType = sortType
    )
}.stateIn(viewModelScope, SharingStarted.WhileSubscribed( stopTimeoutMillis: 5000), ListItemState())
```

Programski kod 3.43 Kreiranje tokova

```
fun onEvent(event: ListItemEvent) {
    when (event) {
        is ListItemEvent.DeleteListItem -> {...}

        ListItemEvent.SaveListItem -> {...}

        is ListItemEvent.SortListItems -> {...}
    }
}

// Function to add or update a classification in the database
fun addOrUpdateClassification(name: String) {
    viewModelScope.launch { this: CoroutineScope
        val existingItem = dao.getItemByName(name)
        if (existingItem != null) {
            // Item exists, update count
            val updatedItem = existingItem.copy(count = existingItem.count + 1)
            dao.addOrUpdateListItem(updatedItem)
        } else {
            // Item does not exist, insert new item
            val newItem = ListItem(name = name, count = 1)
            dao.addOrUpdateListItem(newItem)
        }
    }
}
```

Programski kod 3.44 Metode onEvent i addOrUpdateClassification

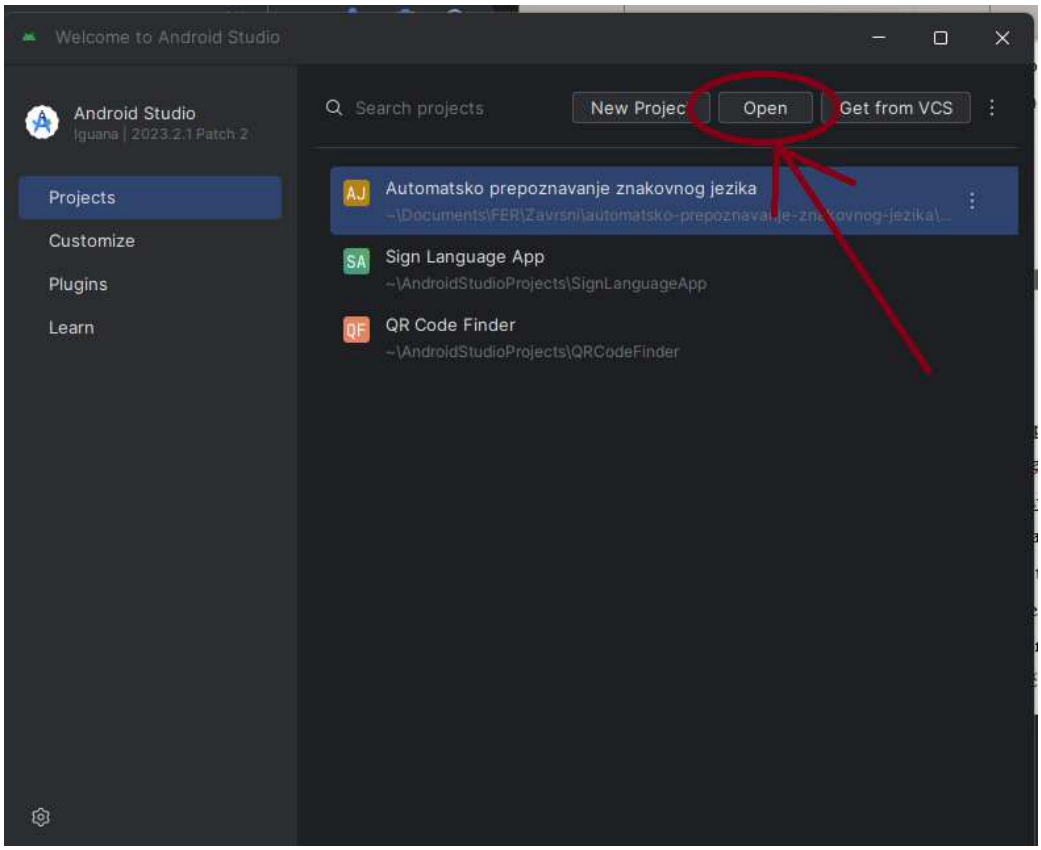
4. Upute za pokretanje

Ovo poglavlje sadrži upute za pokretanje programa 'makeDatabase.py' i 'train.py' pisanih u verziji 3.10.0 programskog jezika Python, te izrađene Android aplikacije. Prije nego što je uopće moguće pokrenuti programe pisane u programskom jeziku Python, potrebno je instalirati nekoliko dodatnih biblioteka:

- biblioteka NumPy – verzija 1.26.4
- biblioteka Tkinter – verzija 8.6
- biblioteka MediaPipe – verzija 0.10.11
- biblioteka Tensorflow - verzija 2.15.0
- biblioteka Pandas – verzija 2.2.1
- biblioteka Scikit-Learn – verzija 1.4.1

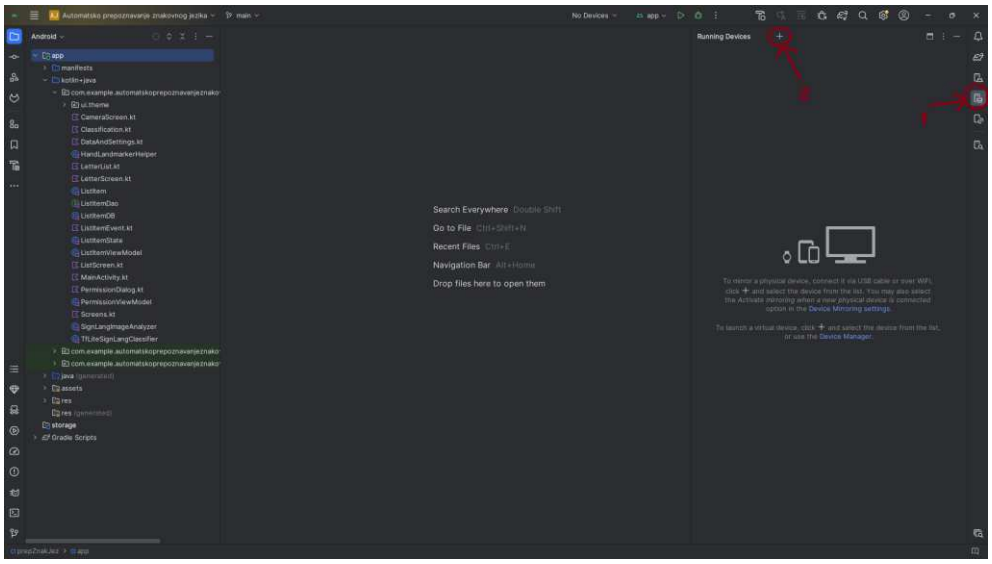
Programi 'makeDatabase.py' i 'train.py' pokreću se pozicioniranjem u mapu u kojoj se oni nalaze te upisivanjem naredbe `python makeDatabase.py`, odnosno `python train.py` u naredbeni redak.

Za instalaciju i pokretanje izrađene aplikacije potrebno je imati instaliran alat Android Studio. Verzija alata korištena za izradu aplikacije je Android Studio Iguana 2023.2.1 Patch 2. Pokretanjem alata Android Studio prikazuje se prozor kao na (Slika 4.1). Ukoliko je projekt s kodom izrađene aplikacije već prije otvaran, prikazat će se u prozoru pod opcijom Projects. Ako to nije slučaj, potebno je odabrati opciju Open pri vrhu zaslona te odabrati mapu sa spremljenim kodom (ukoliko je kod preuzet sa git repozitorija, tražena mapa bit će mapa prepZnakJez).

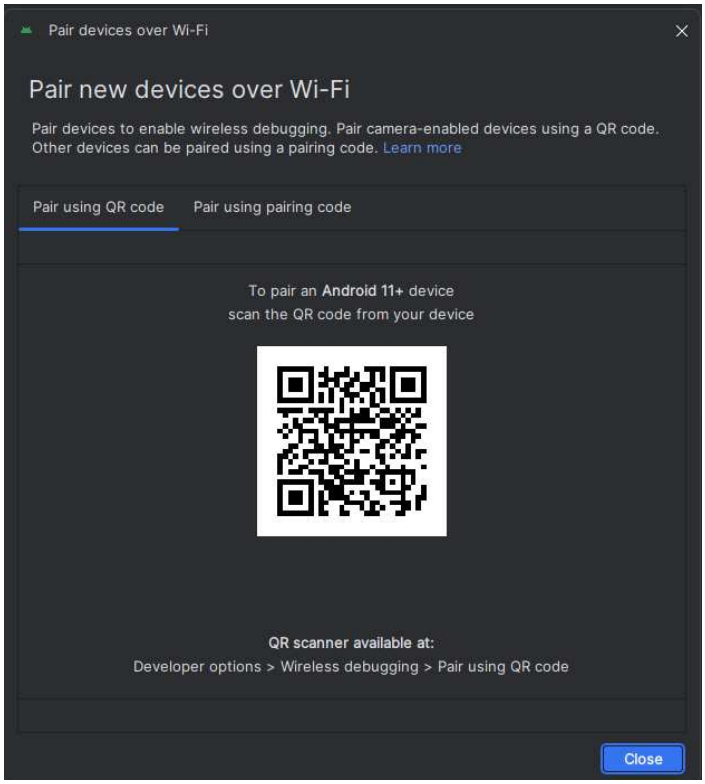


Slika 4.1 Otvaranje projekta

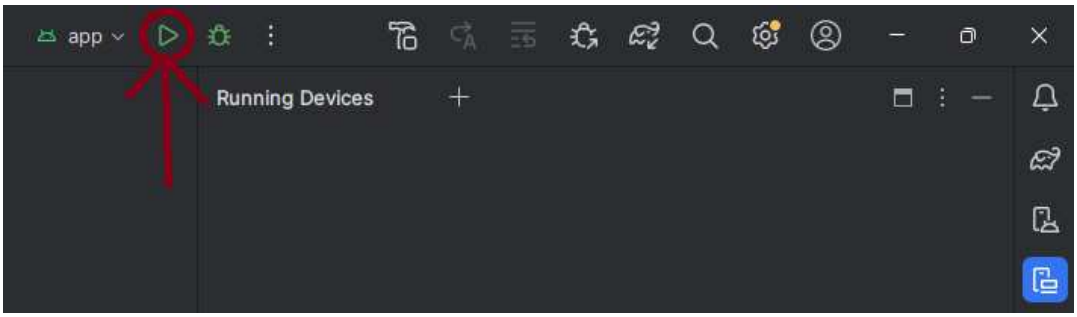
Nakon odabira mape otvara se uređivač koda alata Android Studio. Za instalaciju izrađene aplikacije potrebno je spojiti uređaj sa sustavom Android na računalo (pomoću USB kabela ili bežično). Kako bi alat prepoznao uređaj, u opcijama za razvojne programere u postavkama Android uređaja potrebno je uključiti opciju 'Otklanjanje pogrešaka putem USB-a' (eng. USB Debugging), odnosno 'Bežično otklanjanje pogrešaka' (eng. Wireless Debugging). Ukoliko se koristi prva opcija, nakon priključenja Android uređaja u računalo, dovoljno je stisnuti gumb 'Run app' prikazan na (Slika 4.4) te pričekati dok se aplikacija ne instalira na uređaj i pokrene se. U slučaju da je korišteno bežično povezivanje, potrebno je nakon odabira opcije 'Bežično otklanjanje pokrešaka' kliknuti gumb 'Running Devices' na desnoj strani prozora, prikazan na (Slika 4.2), te zatim na znak + te opciju 'Pair Devices Using Wi-Fi'. Odabirom ove opcije prikazuje se prozor s QR kodom kao na (Slika 4.3), čijim skeniranjem se Android uređaj povezuje s alatom Android Studio. Nakon uspješnog povezivanja, dovoljno je kliknuti gumb 'Run app' i pričekati da se instalacija završi.



Slika 4.2 Bežično povezivanje Android uređaja



Slika 4.3 Prozor s QR kodom za povezivanje



Slika 4.4 Pokretanje instalacije

Zaključak

Ovim radom detaljno je opisan postupak pripreme i spremanja podataka u csv datoteku, izrada i konverzija modela koji koristi te podatke u TensorFlow Lite oblik te izrada aplikacije koja koristi taj model za klasifikaciju koordinata točaka dlana dobivenih korištenjem MediaPipe Hands radnog okvira. Rezultat klasifikacije slova su engleske abecede, zajedno sa postotkom točnosti prepoznavanja. Ideja za izradu ovog projekta vrlo je jednostavna, no njen razvoj mnogo je složeniji od toga. Za pripremu podataka i izradu modela umjetne inteligencije bilo je potrebno upoznati se s raznim bibliotekama programskog jezika Python (npr. NumPy, OpenCv, Pandas…) te alatima TensorFlow Lite i MediaPipe Hands. Za postizanje željenih mogućnosti aplikacije, izrađeni model bilo je potrebno testirati na skupu za testiranje te pomoću izrađene aplikacije. Ako je nužno, model se trebao poboljšavati Trenutno izrađena aplikacija jedino ima mogućnost prepoznavanja američkog znakovnog jezika, no ovu funkcionalnost moguće je proširiti i na druge vrste znakovnog jezika. Jedan od načina na koji bi se to moglo postići dodavanje je opcije za odabir određenog TensorFlow Lite modela za prepoznavanje određenog znakovnog jezika. Dodavanje ove opcije podrazumijevalo bi i izradu modela za te jezike. Osim trenutno dostupne mogućnosti prepoznavanja znakovnog jezika, ovu funkcionalnost moglo bi se proširiti i na prepoznavanje jednostavnih kratkih izraza i gesti, a nakon nekog vremena čak i na prepoznavanje kompliciranih dijelova znakovnog jezika ili čak cijelih rečenica. Dodavanjem opcije za pretvaranje prepoznatog znakovnog jezika u govor još je jedna mogućnost poboljšanja izrađene aplikacije. Prepoznavanje znakovnog jezika vrlo je korisno i omogućava jednostavniju komunikaciju među korisnicima aplikacije i ljudima koji koriste znakovni jezik. Osim omogućavanja lakše komunikacije, jedan od ciljeva ove aplikacije može biti i educiranje korisnika o načinu korištenja znakovnog jezika.

Literatura

[1] Službena stranica razvojnog okruženja Android Studio
https://developer.android.com/studio/intro

[2] Službena stranica programskog jezika Kotlin
https://kotlinlang.org/docs/home.html

[3] Dokumentacija alata Jetpack Compose
https://developer.android.com/develop/ui/compose

[4] Composable metode
https://medium.com/@williamrai13/exploring-composable-functions-in-android-an-introductory-guide-837504d51064

[5] Službena stranica Tensorflow alata
https://www.tensorflow.org

[6] Službena stranica Tensorflow Extended alata
https://www.tensorflow.org/tfx/api_overview

[7] Službena stranica Tensorflow.js alata
https://js.tensorflow.org/api/latest/

[8] Službena stranica Tensorflow Lite alata
https://www.tensorflow.org/lite/api_docs

[9] Službena stranica MediaPipe radnog okvira
https://ai.google.dev/edge/mediapipe/solutions/guide

[10] Dokumentacija MediaPipe Hands radnog okvira
https://mediapipe.readthedocs.io/en/latest/solutions/hands.html

[11] Dokumentacija Python biblioteke tkinter
https://docs.python.org/3/library/tkinter.html

[12] Dokumentacija OpenCv biblioteke
https://docs.opencv.org/4.x/dd/d43/tutorial_py_video_display.html

[13] Dokumentacija NumPy biblioteke
https://numpy.org/doc/stable/user/whatisnumpy.html

[14] Dokumentacija Pandas biblioteke
https://pandas.pydata.org/docs/

[15] Dokumentacija Python biblioteke random

https://docs.python.org/3/library/random.html

[16] Dokumentacija biblioteke scikit-learn za izradu matrice konfuzije https://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#confusion-matrix

[17] Pandas DataFrame format podataka https://pandas.pydata.org/docs/user_guide/dsintro.html#dataframe

[18] B. Dalbelo Bašic, M. Čupić, J. Šnajder. Umjetne neuronske mreže https://www.fer.unizg.hr/_download/repository/UmjetneNeuronskeMreze.pdf

[19] Keras Sequential model https://keras.io/guides/sequential_model/

[20] ReLu aktivacijska funkcija https://www.kaggle.com/code/dansbecker/rectified-linear-units-relu-in-deep-learning

[21] Softmax aktivacijska funkcija https://machinelearningmastery.com/softmax-activation-function-with-python/

[22] Optimizacijski algoritam Adam https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[23] Biblioteka Jetpack Compose Navigation https://developer.android.com/develop/ui/compose/navigation

[24] Komponenta LazyColumn alata Jetpack Compose https://medium.com/@mal7othify/lists-using-lazycolumn-in-jetpack-compose-c70c39805fbc

[25] Biblioteka Painter https://developer.android.com/reference/kotlin/androidx/compose/ui/graphics/painter/package-summary

[26] DataStore biblioteka https://developer.android.com/topic/libraries/architecture/datastore

[27] Alat Preference DataStore https://medium.com/androiddevelopers/all-about-preferences-datastore-cc7995679334

[28] Alat Proto DataStore https://medium.com/androiddevelopers/all-about-proto-datastore-1b1af6cd2879

[29] Dokumentacija sučelja Serializer https://developer.android.com/reference/kotlin/androidx/datastore/core/Serializer

[30] Kotlin Serialization biblioteka https://kotlinlang.org/docs/serialization.html#formats

[31] Korutine u alatu Jetpack Compose https://developer.android.com/kotlin/coroutines

[32] Jetpack Compose Accompanist biblioteka https://medium.com/androiddevelopers/jetpack-compose-accompanist-an-faq-b55117b02712

[33] Accompanist Permissions biblioteka https://medium.com/@lukohnam/jetpack-compose-permissions-using-accompanist-library-b1c0fbbe8831

[34] Sučelje ViewModel https://developer.android.com/topic/libraries/architecture/viewmodel

[35] Biblioteka CameraX https://developer.android.com/media/camera/camerax, https://medium.com/@rohitgarg2016/to-integrate-a-simple-camera-using-camerax-in-a-kotlin-android-app-cefa4cfd0edc

[36] Sučelje ImageAnalysis.Analyzer https://developer.android.com/reference/androidx/camera/core/ImageAnalysis.Analyzer

[37] Primjer koda klase HandLandmarkerHelper sa službene stranice MediaPipe Hands Landmarker-a https://ai.google.dev/edge/mediapipe/solutions/vision/hand_landmarker/android , https://github.com/google-ai-edge/mediapipe-samples/blob/main/examples/hand_landmarker/android/app/src/main/java/com/google/mediapipe/examples/handlandmarker/HandLandmarkerHelper.kt

[38] Biblioteka Room Database https://developer.android.com/training/data-storage/room, https://amitraikwar.medium.com/getting-started-with-room-database-in-android-fa1ca23ce21e

[39] Sučelje Parcelable https://developer.android.com/reference/kotlin/android/os/Parcelable

Sažetak

Autor: Luka Kurtin

Naslov: Automatsko prepoznavanje znakovnog jezika

Ovim radom detaljno je opisan postupak izrade mobilne aplikacije za uređaje sa sustavom Android kojom se omogućava prepoznavanje slova američke znakovne abecede. Za pripremu podataka i izradu modela napisani su programi u programskom jeziku Python koji koriste biblioteke NumPy, OpenCv, tkinter i Pandas te alat TensorFlow Lite i MediaPipe Hands radni ok. Program za izradu modela također koristi podatke dobivene iz programa za pripremu podataka spremljene u csv datoteci kao ulaz za model. Dobiveni model, zajedno s MediaPipe Hands radnim okvirom, koristi se za prepoznavanje znakovnog jezika u izrađenoj Android aplikaciji. Nakon objašnjenja Python programa, detaljno je opisan postupak izrade aplikacije, zajedno s prikazima i isječcima koda, te način na koji izrađena aplikacija prepoznaje znakovni jezik i sprema podatke u Room bazu podataka.

Ključne riječi: američki znakovni jezik, klasifikacija znakovnog jezika, MediaPipe Hands, TensorFlow Lite, Android Studio, programski jezik Kotlin, Jetpack Compose, Room baza podataka

Summary

Author: Luka Kurtin

Title: Automatic sign language recognition

This paper provides a detailed description of the process of creating a mobile application for Android devices that enables the recognition of letters from the American Sign Language alphabet. For preparation of data and creation of the model, programs were written in the Python programming language using libraries such as NumPy, OpenCV, tkinter, and Pandas, as well as TensorFlow Lite tool and the MediaPipe Hands framework. The model creation program also utilizes data obtained from the data preparation program, saved in a CSV file, as input for the model. The resulting model, along with the MediaPipe Hands framework, is used for sign language recognition in the developed Android application. Following the explanation of the Python programs, the process of developing the application is described in detail, including screenshots and code snippets, and the way the application recognizes sign language and stores data in a Room database.

Key words: American Sign Language, sign language classification, MediaPipe Hands, TensorFlow Lite, Android Studio, Kotlin programming language, Jetpack Compose, Room Database

Skraćenice

ASL	American Sign Language	američki znakovni jezik
TFLite	TensorFlow Lite	TensorFlow Lite
CSV	Comma-Separated Values	vrijednosti odvojene zarezom