Izrada aplikacije za zvučno upravljanje računalom uz predtrenirani model strojnog učenja

Mrkonjić, Marko

Undergraduate thesis / Završni rad

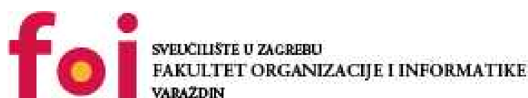
2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Zagreb, Faculty of Organization and Informatics / Sveučilište u Zagrebu, Fakultet organizacije i informatike**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:211:551228

Rights / Prava: Attribution-NonCommercial 3.0 Unported/Imenovanje-Nekomercijalno 3.0

Download date / Datum preuzimanja: **2025-02-20**



Repository / Repozitorij:

Faculty of Organization and Informatics - Digital Repository

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Marko Mrkonjić

Izrada aplikacije za zvučno upravljanje računalom uz predtrenirani model strojnog učenja

ZAVRŠNI RAD

Varaždin, 2024.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Marko Mrkonjić

Matični broj: 0016144864

Studij: ***Primjena informacijske tehnologije u poslovanju***

Izrada aplikacije za zvučno upravljanje računalom uz predtrenirani model strojnog učenja

ZAVRŠNI RAD

Mentor/Mentorica:

Prof. dr. sc. Markus Schatten

Varaždin, rujan 2024.

Marko Mrkonjić

Izjava o izvornosti

Izjavljujem da je moj završni/diplomski rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor/Autorica potvrdio/potvrdila prihvaćanjem odredbi u sustavu FOI-radovi

---

Sažetak

Ova tema završnog rada ima za cilj razviti aplikaciju za zvučno upravljanje računalom pomoću pred treniranog ili vlastitog modela strojnog učenja. U početnom pristupu probao bi probati napisati vlastiti model pomoću Framework PyTorcha kako bi se omogućilo upravljanje Windows operativnim sustavom glasovnim naredbama. Prvi korak bio bi prikupljanje podataka za treniranje modela. Zatim, koristeći PyTorch, implementirao bi se model za prepoznavanje glasovnih naredbi. Ovaj korak uključuje treniranje modela s velikom količinom podataka kako bi se model naučio prepoznavati različite glasovne naredbe.

Nakon toga, razvila bi se aplikacija koja koristi taj model za prepoznavanje i interpretaciju glasovnih naredbi. Aplikacija bi bila integrirana s Windows operativnim sustavom kako bi omogućila korisnicima upravljanje računalom kroz glasovne naredbe.

Ukoliko prvi pokušaj s vlastitim modelom ne bude uspješan, razmotrio bi upotrebu već pred treniranog modela strojnog učenja. Ovo bi ubrzalo proces implementacije i omogućilo efikasnije zvučno upravljanje računalom. Konačni program trebao bi omogućiti korisnicima jednostavno i intuitivno upravljanje Windows operativnim sustavom pomoću glasa, poboljšavajući korisničko iskustvo.

Ključne riječi: zvučno upravljanje računalom, model strojnog učenja, PyTorch, Windows, glasovne naredbe, prikupljanje podataka, implementacija modela, prepoznavanje glasovnih naredbi, pred trenirani model, korisničko iskustvo.

Sadržaj

1. Uvod ... 1
1.1. Problem kojim se bavim u ovom radu ... 1
1.2. Cilj ovog rada ... 1
1.3. Metode i tehnike rada ... 2
1.4. Motivacija ... 4
2. Tehnički dio izrada ... 5
2.1. Skupljanje podataka za treniranje ... 5
2.2. Preprocesiranje podataka (augmentacija) ... 6
2.3. Zašto RNN ? ... 12
2.4. Arhitektura RNN modela ... 13
2.4.1. Osnovna Arhitektura RNN-a ... 13
2.4.2. Varijante RNN-a ... 13
2.4.3. Arhitektura EnhancedRNN_v3 ... 14
2.4.4. Ključni dijelovi EnhancedRNN_v3 arhitekture ... 15
2.4.5. Zašto RNN ? ... 15
2.5. Prvi neuspjeli pokušaj treniranja vlastitog modela ... 16
2.5.1. Detaljan pregled implementacije modela ... 16
2.5.2. Proces treniranja ... 17
2.5.3. Izazovi i problemi ... 18
2.5.4. Evaluacija Modela ... 19
2.6. Nova strategija: Wav2Vec2 ... 20
2.6.1. Zašto Wav2Vec2 ? ... 21
2.6.2. Arhitektura Wav2Vec2 ... 21
2.6.4. Proces Fine-tuninga Wav2Vec2 ... 22
2.6.5. Prednosti pristupa fine-tuningom ... 23
2.7. Pogrešan pristup fine-tuningu Wav2Vec2 ... 23
2.7.1. Ograničenja u strukturiranju podataka ... 23
2.7.2. Posljedice jednostavne klasifikacije ... 23
2.7.3. Lekcije naučene ... 24
2.8. Novi pristup Fine-tuningu Wav2Vec2 ... 25
2.8.1. Restrukturiranje podataka za fine-tuning ... 25
2.8.2. Prilagodba i generiranje novih podataka ... 25
2.8.3. Prednosti novog pristupa ... 28
2.9. Fine-tuning Wav2Vec2.0 CTC Modela ... 28
2.9.1. Priprema podataka za fine-tuning ... 28
2.10. 3-stage Model: YAMNet, Wav2Vec2.0 i DistilBERT ... 38

2.10.1. Prvi korak: IsHumanVoice (YAMNet) 38
2.10.2. Drugi korak: DetermineSpokenWord (Wav2Vec2.0) 39
2.10.3. Treći korak: ClassifyItSpokenWord (DistilBERT) 40
2.10.4. Integrirani 3-stage Model 40
2.11. Fine-tuning DistilBERT za Klasifikaciju Glasovnih Naredbi 41
2.12. Windows Integracija za Glasovno Upravljanje 44
2.12.1. Uvod u Windows integraciju 44
2.12.2. Inicijalizacija ključnih komponenti 44
2.12.3. Učitavanje i konfiguracija modela 45
2.12.4. Implementacija VoiceCommandRecognizer klase 46
2.12.5. Klasifikacija naredbi i izvršavanje 47
2.12.6. Funkcija classify_audio_v2 48
2.12.7. Funkcija callback_v2 49
2.12.8. Izvršavanje Windows naredbi 50
2.12.9. Pokretanje aplikacije i grafičko sučelje 51
3. Konačni rezultati 53
3.1. Prikaz rezultata 53
4. Zaključak 56
Popis slika 58
Popis literature 58

Uvod

1.1. Problem kojim se bavim u ovom radu

Danas ljudi uglavnom upravljaju računalima putem klasičnih metoda kao što su tipkovnica, miš i zasloni osjetljivi na dodir. Ove metode nisu uvijek intuitivne ili dostupne svima. Koristiti takve klasične načine kontrole često predstavlja problem za osobe s tjelesnim ograničenjima ili one koji žele upravljati računalima bržim i prirodnijim načinima. Upravljanje glasom moglo bi pružiti rješenje, ali trenutna rješenja imaju nedostatke u smislu točnosti, fleksibilnosti ili integracije s operativnim sustavom, posebno s Windowsom.

I tu leži glavni problem a to je osmisliti sučelje koje omogućuje korisnicima da upravljaju Windows operativnim sustavom jednostavnim glasovnim naredbama, pružajući intuitivniji osjećaj i učinkovito iskustvo. Među glavnim pitanjima koja se postavljaju su ona o tome kako stvoriti sustav prepoznavanja glasa koji ne samo da može točno razumjeti govor, već i brzo, kako ga integrirati s Windowsom te kako stvoriti rješenje koje je robusno za stvarne uvjete. Najveći izazovi su u razvoju ili prilagodbi modela strojnog učenja koji će pouzdano prepoznavati glasovne naredbe, prikupljanju i obradi dovoljno kvalitetnih podataka za treniranje modela te kreiranju aplikacije koja će učinkovito komunicirati između prepoznavanja glasa i operativnog sustava. Ako to nije dovoljno za očekivanja izgradnjom vlastitog modela, onda bi se morali istražiti putevi fino podešavanja istreniranih modela za željenu funkcionalnost.

1.2. Cilj ovog rada

Cilj ovog završnog rada je razviti Windows aplikaciju koja omogućava upravljanje Windows operativnim sustavom putem glasovnih naredbi. Kroz ovaj projekt, planiram izraditi sustav koji koristi strojno učenje za prepoznavanje i interpretaciju glasovnih naredbi, čime se korisnicima pruža jednostavno i intuitivno iskustvo zvučnog upravljanja računalom. Prvi korak u tom procesu uključuje prikupljanje podataka za treniranje modela i razvoj vlastitog modela koristeći PyTorch Framework. Cilj je omogućiti modelu da prepozna određene naredbe koje se potom mogu koristiti za upravljanje različitim funkcijama unutar Windows operativnog sustava.

Ako se pokaže da vlastiti model ne daje zadovoljavajuće rezultate, istražit ću mogućnost prilagodbe već pred treniranih modela kroz postupak fine-tuninga, kako bih postigao željeni nivo točnosti i brzine.

Krajnji rezultat trebao bi biti aplikacija koja omogućava korisnicima upravljanje računalom na način koji je prirodan, učinkovit i dostupan širokom spektru korisnika.

1.3. Metode i tehnike rada

Za izradu aplikacije koja omogućuje zvučno upravljanje Windowsom, koristit ću niz tehnologija, alata i modela strojnog učenja, uz posebnu pažnju na prepoznavanje glasovnih naredbi. Projekt se temelji na nekoliko koraka:

1. **Prikupljanje podataka**:

- Podatke za treniranje prikupljam putem mikrofona i mobilnog uređaja. Ovi zvučni zapisi zatim se obrađuju i koriste za treniranje modela.

2. **Tehnologije i alati**:

- **PyTorch**: Glavni Framework koji koristim za izradu i treniranje modela strojnog učenja, uključujući prilagodbu postojećih modela poput Wav2Vec2.
- **TensorFlow i TensorFlow Hub**: Ove alate koristim za dodatne modele, poput YAMNet-a za prepoznavanje ljudskog glasa.
- **Librosa**: Knjižnica za obradu zvuka koja pomaže u analiziranju i ekstrakciji značajki iz zvučnih podataka.
- **PyAudio**: Za rad sa zvučnim signalima u stvarnom vremenu.
- **PyQt5**: Koristi se za razvoj grafičkog korisničkog sučelja (GUI) koje omogućava interakciju s korisnikom.

3. **Modeli i pristupi**:

- **RNN Arhitektura**: Prvi pokušaj s vlastitim modelom koristio je rekurentne neuronske mreže (RNN) za klasifikaciju glasovnih naredbi. Iako ovaj pristup nije dao željene rezultate, poslužio je kao temelj za daljnje iteracije.
- **Prepoznavanje ljudskog glasa**: U prvoj fazi obrade, koristi se pred treniran model YAMNet koji detektira je li ulaz ljudski glas ili neki drugi zvuk.
- **Wav2Vec2 For CTC (Connectionist Temporal Classification)**: Ovaj model je prilagođen za transkripciju zvučnih zapisa u tekst. Koristi se nakon što je potvrđeno da je ulaz ljudski glas.

- **VoiceCommandClassifier**: Konačni model za klasifikaciju prepoznatih tekstualnih naredbi u specifične akcije koje će Windows izvršiti. Korištena je pipeline funkcija iz HuggingFace Transformers knjižnice kako bi se pojednostavio proces klasifikacije.

4. **Preprocesiranje podataka**:

- Prikupljeni zvučni zapisi prolaze kroz postupke poput augmentacije, normalizacije i ekstrakcije značajki koristeći librosa i torchaudio. Također, koristim Resample iz torchaudio.transforms za prilagodbu uzoraka različitim frekvencijama.

5. **Treniranje modela**:

- Modeli su trenirani na PyTorch-u s korištenjem GPU-a. Proces treniranja uključuje tehnike poput regulacije (npr. clip_grad_norm_) kako bi se izbjeglo prenaučavanje. Također sam koristio TrainingArguments i Trainer klase iz Transformers knjižnice za automatizaciju fine-tuninga.

6. **Implementacija zvučnog upravljanja**:

- Nakon što model prepozna i klasificira glasovnu naredbu, koriste se pyautogui i web browser za izvršavanje odgovarajućih radnji na sustavu, poput otvaranja aplikacija ili navigacije kroz datoteke.

7. **Konačna aplikacija**:

- Aplikacija koristi PyQt5 za grafičko sučelje i omogućava korisniku jednostavno pokretanje, praćenje i podešavanje sustava za prepoznavanje glasovnih naredbi.

Kroz kombinaciju ovih tehnologija i modela cilj je stvoriti robusnu aplikaciju koja omogućava učinkovito glasovno upravljanje Windows operativnim sustavom.

1.4. Motivacija

Glasovno upravljanje računalima postaje sve popularnije kako tehnologija napreduje, no postojeća rješenja često su skupa, komplicirana ili ograničena po pitanju funkcionalnosti. Kroz ovaj projekt želim istražiti mogućnosti razvoja prilagođenog sustava koji bi omogućio jednostavno i pristupačno glasovno upravljanje Windows operativnim sustavom. Osobna motivacija dolazi iz želje da unaprijedim korisničko iskustvo i istražim kako strojno učenje i tehnologije prepoznavanja glasa mogu poboljšati interakciju s računalima.

Osim tehničkog izazova, motivacija za ovaj rad leži i u rješavanju problema pristupačnosti. Glasovno upravljanje može olakšati rad osobama s invaliditetom ili onima koje imaju poteškoće s korištenjem standardnih metoda interakcije poput tipkovnice i miša. Implementacija fleksibilnog i prilagodljivog sustava, koji se može koristiti u svakodnevnom radu na računalu, može imati velik praktični doprinos.

Također, razvoj ovog projekta omogućuje mi dublje razumijevanje tehnologija poput PyTorch-a, modela učenja, kao i prilagodbe već postojećih modela. Kroz ovaj rad, želim dodatno proširiti svoja znanja u području primjene strojnog učenja, rada s audio podacima te integracije rješenja u stvarne sustave.

Tehnički dio izrada

1.5. Skupljanje podataka za treniranje

Kvalitetno treniranje modela za prepoznavanje glasovnih naredbi uvelike ovisi o raznolikosti i obujmu prikupljenih podataka. Stoga sam se odlučio pristupiti procesu skupljanja podataka na način koji će osigurati što širu pokrivenost različitih stilova izgovora, intonacija i varijacija u glasu. Da bi postigao ovaj cilj, u proces prikupljanja podataka uključio sam šest osoba, uključujući i sebe. Razlog zbog kojeg sam odabrao više govornika je kako bih osigurao raznovrsnost u glasovima, čime model postaje robusniji i sposobniji prepoznati glasovne naredbe neovisno o pojedinačnim razlikama među korisnicima.
Svaka od tih šest osoba imala je zadatak snimiti tri različite vrste glasovnih naredbi. Komande koje sam odabrao za ovaj projekt su:

1. **Open web browser**
2. **Search**
3. **Windows Menu**

Ove komande su odabrane zbog svoje praktične primjene u svakodnevnom korištenju računala. Kako bi se osiguralo dovoljno podataka za treniranje, svaka osoba je snimila po 100 uzoraka za svaku naredbu. Ukupno, svaka osoba je snimila 300 uzoraka (100 za svaku od tri komande), što je rezultiralo s ukupno 1800 zvučnih zapisa za cijeli projekt. Kako bih povećao raznolikost unutar uzoraka, dao sam jasne upute svim sudionicima da pokušaju mijenjati način izgovora između različitih uzoraka. Konkretno, zamolio sam ih da mijenjaju ton glasa, brzinu govora i naglasak. Na primjer, jedan uzorak bi mogao biti izgovoren s višim tonom i bržim tempom, dok bi sljedeći bio sporiji i s nižim tonom. Na ovaj način sam već u početnoj fazi uveo elemente augmentacije podataka, što inače zahtijeva dodatne postupke tijekom obrade podataka. Varijacije u glasu su ključne jer povećavaju robusnost modela i omogućuju mu da bude učinkovitiji u prepoznavanju naredbi od različitih korisnika u različitim uvjetima. Svi zvučni zapisi snimani su pomoću mobilnih uređaja. Ovaj pristup omogućio je jednostavno prikupljanje podataka bez potrebe za dodatnom opremom, čime je proces bio pristupačan i brz. Svaki zapis je trajao maksimalno dvije sekunde, što je dovoljno vremena da se jasno i precizno izgovori naredba, dok s druge strane ograničava nepotrebne pauze ili šumove koje bi mogle otežati treniranje modela.

Nakon što su svi sudionici završili sa snimanjem, prikupljene zvučne zapise sam organizirao u strukturu mapa kako bih olakšao daljnji proces rada. Podaci su bili raspoređeni u mape prema imenu govornika i vrsti komande koju su izgovarali. Na primjer, svaka osoba je imala svoju mapu s podmapama za "Open web browser," "Search" i "Windows Menu." Ova organizacija omogućuje jednostavnu navigaciju kroz podatke i lakšu manipulaciju prilikom preprocesiranja i treniranja modela.

Jedan od ključnih koraka u obradi podataka bilo je pretvaranje svih zvučnih zapisa u **WAV** format. Iako su izvorni zapisi bili u različitim formatima (kao što su MP3 ili M4A), odlučio sam koristiti WAV zbog njegove visoke kvalitete i kompatibilnosti s alatima za obradu audio signala i treniranje modela. Korištenje uniformnog formata podataka ključno je za održavanje dosljednosti tijekom cijelog procesa obrade i treniranja.

Za konverziju svih zapisa u WAV format koristio sam besplatni softver **freac** (https://www.freac.org/), koji je jednostavan za korištenje, a istovremeno nudi mogućnost brze i masovne konverzije audio datoteka. Ovaj alat mi je omogućio da u kratkom vremenu obradim sve snimljene podatke i pripremim ih za daljnje korake. Nakon što su svi podaci pretvoreni u odgovarajući format, mogao sam ih koristiti za različite obrade kao što su normalizacija zvučnih signala, ekstrakcija značajki i priprema podataka za treniranje modela.

Cjelokupan proces prikupljanja podataka dizajniran je tako da osigura visoku raznolikost u glasovnim uzorcima, dok istovremeno zadržava organiziranost i jednostavnost pristupa. Tako prikupljeni podaci čine čvrstu osnovu za treniranje modela koji mora biti sposoban prepoznati naredbe izgovorene u različitim uvjetima, s različitim glasovima i načinima izgovora. Kroz ovaj proces osigurao sam realistične i reprezentativne uzorke koji simuliraju stvarnu upotrebu aplikacije, što je ključno za uspješnu implementaciju zvučnog upravljanja Windows operativnim sustavom.

1.6. Preprocesiranje podataka (augmentacija)

Preprocesiranje podataka za model koji prepoznaje zvukove ili klasificira audio zapise uključuje nekoliko koraka kako bi se osigurala konzistentnost podataka i povećala otpornost modela na varijacije u zvučnim zapisima. Ovaj proces uključuje pretvorbu audio signala, augmentaciju (generiranje dodatnih podataka), te konačnu pripremu podataka za treniranje modela. U ovom projektu, preprocesiranje je implementirano pomoću biblioteka kao što su **librosa**, **torchaudio**, **pandas**, te moduli iz PyTorcha. Detaljan opis procesa preprocesiranja slijedi u nastavku, s naglaskom na kod koji ste pružili

1. Importiranje potrebnih biblioteka

Biblioteke koje su korištene u preprocesiranju audio signala uključuju:

- **torchaudio**: Korišten za učitavanje, transformaciju i manipulaciju audio podacima.
- **librosa**: Koristi se za manipulaciju i ekstrakciju značajki iz audio signala.
- **pandas**: Korišten za rad s tabličnim podacima u obliku DataFramea.
- **scikit-learn**: Korišten za enkodiranje klasa i podjelu podataka u trening, validacijski i testni set.

Ostale biblioteke koje se koriste su standardne biblioteke za rad s PyTorchom i numeričkim podacima (npr. **torch**, **numpy**, **matplotlib**).

2. Treniranje modela uz pomoć GPU

U drugom koraku, kod osigurava da se koristi GPU ako je dostupan, što značajno ubrzava treniranje modela.

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"GPU is available: {torch.cuda.is_available()}")
print(f"GPU: {torch.cuda.get_device_name(0)}")
```

3. Priprema podataka

U trećem koraku, svi audio zapisi unutar mape TrainingData se učitavaju i pripremaju za pretvorbu. Podaci su organizirani prema klasama (etiketama), a svaka klasa sadrži više audio zapisa.

```
training_data_path = Path("../TrainingData")
files_to_convert = listdir(training_data_path)
```

4. Audio transformacije

Implementirao sam klasu AudioUtils_Transformer koja sadrži metode za transformaciju audio zapisa. Ova klasa omogućuje učitavanje, rekanalizaciju, resampliranje, te druge manipulacije zvukom. Neke ključne transformacije uključuju:

- **Resampliranje i rekanalizacija**: Uniformizira frekvenciju uzorkovanja na 16 kHz i postavlja sve zapise na jedan kanal (mono).

```
@staticmethod
def resample(aud, new_sr):
    sig, sr = aud
    if sr == new_sr:
        return aud
    resig = Resample(sr, new_sr)(sig)
    return (resig, new_sr)

@staticmethod
def rechanel(aud, new_channel):
    sig, sr = aud
    if sig.shape[0] == new_channel:
        return aud
    if new_channel == 1:
        resig = sig[:1, :]
    else:
        resig = torch.cat([sig, sig])
    return (resig, sr)
```

Augmentacija: Kroz razne metode, podaci se augmentiraju kako bi se povećala količina podataka za treniranje te simulirale različite varijacije u stvarnim uvjetima.

- **Vremensko pomicanje**: Simulira pomicanje signala unutar zapisa.
- **Dodavanje šuma**: Dodaje nasumične šumove u signal kako bi model bio robusniji na šumne uvjete.
- **Promjena brzine reprodukcije**: Mijenja brzinu signala za simulaciju sporijeg ili bržeg govora.
- **Dodavanje odjeka**: Simulira odjeke u prostorijama kako bi model bio otporniji na akustičke efekte.

```
@staticmethod
def aug_time_shift(aud, shift_limit):
    sig, sr = aud
    _, sig_len = sig.shape
    shift_amt = int(np.random.random() * shift_limit * sig_len)
    return (sig.roll(shift_amt), sr)

@staticmethod
def inject_noise(signal, noise_factor=0.01):
    sig, sr = signal
    noise = torch.randn_like(sig)
    augmented_signal = sig + noise_factor * noise
    return (augmented_signal, sr)

@staticmethod
def change_speed(signal, speed_factor=1.1):
    sig, sr = signal
    np_signal = sig.numpy()
    speed_changed_signal = librosa.effects.time_stretch(np_signal,
rate=speed_factor)
    if speed_changed_signal.shape[1] > sig.shape[1]:
        speed_changed_signal = speed_changed_signal[:, :sig.shape[1]]
    elif speed_changed_signal.shape[1] < sig.shape[1]:
        padding = sig.shape[1] - speed_changed_signal.shape[1]
        speed_changed_signal = np.pad(speed_changed_signal, ((0, 0), (0,
padding)), mode='constant')
    return (torch.from_numpy(speed_changed_signal), sr)
```

Spektrogramska transformacija: Svaki audio zapis se pretvara u spektrogram, što je vizualna reprezentacija frekvencijskih komponenti signala kroz vrijeme. Koristi se Mel-frekvencijski spektrogram (MFCC) kao značajka ulaza.

```
@staticmethod
def to_spectogram(aud, n_mels=64, hop_len=512):
    sig, sr = aud
    top_db = 80.0
    spectrogram = torchaudio.transforms.MelSpectrogram(sr, n_mels=n_mels,
hop_length=hop_len)(sig)
    spectrogram =
torchaudio.transforms.AmplitudeToDB(top_db=top_db)(spectrogram)
return spectrogram
```

5. Prikupljanje podataka

Podaci su organizirani u Pandas DataFrame, gdje su pohranjeni putevi do audio zapisa i pripadajuće klase. Ovaj DataFrame se zatim koristi za podjelu podataka na trening, validacijski i testni set koristeći **train_test_split** iz **scikit-learn** biblioteke.

```
df = pd.DataFrame(data)
df['class'] = coder.fit_transform(df['class'])

data_train, data_test, label_train, label_test =
train_test_split(data_model, label_model, test_size=0.2, random_state=42)
data_train, data_val, label_train, label_val = train_test_split(data_train,
label_train, test_size=0.3, random_state=42)
```

6. Generiranje augmentiranih podataka

Za svaki audio zapis u trening setu generira se nekoliko augmentiranih verzija koristeći ranije opisane metode. Svaka od tih verzija se zatim pretvara u spektrogram, čime se povećava broj primjera za treniranje modela.

```
data_train_spec = []
label_train_augmented = []

for i, path in enumerate(data_train):
    class_id = label_train[i]
    aud = AudioUtils.open(path)
    resampled_aud = AudioUtils.resample(aud, sr)
    rechanneled_aud = AudioUtils.rechanel(resampled_aud, channel)
    padded_aud = AudioUtils.pad_trunc(rechanneled_aud, duration)
    spectogram = AudioUtils.to_spectogram(padded_aud, n_mels=64,
hop_len=512)
    aud_speed = AudioUtils.change_speed(padded_aud,1.1)
    aud_echo = AudioUtils.add_echo(padded_aud)
    aud_reverse = AudioUtils.reverse(padded_aud)
    aug_shifted_aud = AudioUtils.aug_time_shift(padded_aud, shift_pct)
    aud_noise = AudioUtils.inject_noise(padded_aud)
    aud_gain = AudioUtils.random_gain(padded_aud)
    spectogram_speed = AudioUtils.to_spectogram(aud_speed, n_mels=64,
hop_len=512)
```

```
  spectogram_echo = AudioUtils.to_spectogram(aud_echo, n_mels=64,
hop_len=512)
    spectogram_reverse = AudioUtils.to_spectogram(aud_reverse, n_mels=64,
hop_len=512)
    spectogram_shift = AudioUtils.to_spectogram(aug_shifted_aud, n_mels=64,
hop_len=512)
    spectogram_aug = AudioUtils.spectro_augment(spectogram)
    spectogram_noise = AudioUtils.to_spectogram(aud_noise, n_mels=64,
hop_len=512)
    spectogram_gain = AudioUtils.to_spectogram(aud_gain, n_mels=64,
hop_len=512)
    data_train_spec.extend([
        spectogram,
        spectogram_shift,
        spectogram_aug,
        spectogram_noise,
        spectogram_gain,
        spectogram_speed,
        spectogram_echo,
        spectogram_reverse
        ])
    label_train_augmented.extend([class_id] * 8)
```

7. Kreiranje PyTorch dataset-a

Nakon pretvorbe svih audio zapisa u spektrograme, ti se podaci kombiniraju u PyTorch dataset za treniranje, validaciju i testiranje.

```
data_train = torch.cat(data_train_spec, dim=0)
label_train = torch.tensor(label_train_augmented)
data_val = torch.cat(data_val_spec, dim=0)
label_val = torch.tensor(label_val)
data_test = torch.cat(data_test_spec, dim=0)
label_test = torch.tensor(label_test)
train_dataset = TensorDataset(data_train, label_train)
val_dataset = TensorDataset(data_val, label_val)
test_dataset = TensorDataset(data_test, label_test)
```

8. Postavljanje DataLoader-a

Za svaku od tri vrste skupa podataka (trening, validacija, testiranje), kreiraju se **DataLoaderi** koji omogućuju učitavanje podataka u mini-batchevima tijekom treniranja modela.

```
batch_size = 130
train_loader = DataLoader(train_dataset, batch_size=batch_size,
shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size, shuffle=False)
test_loader = DataLoader(test_dataset, batch_size=batch_size,
shuffle=False)
```

Ovim procesom, podaci su pripremljeni za efikasno treniranje modela u PyTorchu s dodanim slojem augmentacije za povećanje raznolikosti podataka i otpornosti modela na različite varijacije zvuka.

1.7. Zašto RNN ?

RNN (Recurrent Neural Networks) su odabrane kao prvi način rješavanja problema zato što su posebno prilagođene za rad sa sekvencijalnim podacima, što je ključno u zadacima obrade zvuka. Kod prepoznavanja zvučnih signala, poput govornih komandi, RNN-ovi mogu bolje hvatati kontekst i uzorke jer koriste rekurentne veze koje omogućuju zadržavanje informacija o prethodnim stanjima.

Glavne prednosti RNN-a za ovaj problem:

1. **Kontekst kroz vrijeme:** RNN-ovi čuvaju informacije o prethodnim ulazima kroz skrivene slojeve. Ovo je bitno kod audio obrade, gdje svaka audio komponenta (frame) utječe na konačni ishod.
2. **Prilagođenost za sekvencijalne podatke:** Zvučni podaci predstavljaju niz uzastopnih signala koji zajedno nose značenje. RNN modeli prirodno obrađuju sekvencijalne podatke.
3. **Obrada varijabilne duljine:** RNN može raditi s ulazima različitih duljina, što je korisno kod audio podataka koji se mogu razlikovati po trajanju.
4. **Pamćenje dugoročnih ovisnosti:** Zvučni signali mogu sadržavati uzorke koji se pojavljuju kroz različite vremenske trenutke. RNN-ovi zadržavaju sjećanje na ranije podatke u sekvenci.

Ovaj model pruža temelj za obradu audio signala, uz sposobnost hvatanja kompleksnih ovisnosti kroz vremenski slijed, što ga čini prvim logičnim izborom u rješavanju zadataka obrade govora.

1.8. Arhitektura RNN modela

Rekurentne neuronske mreže (RNN) predstavljaju vrstu neuronske mreže posebno dizajnirane za obradu sekvencijalnih podataka. Za razliku od tradicionalnih feedforward neuronskih mreža, RNN-ovi imaju petlje unutar svoje arhitekture koje omogućuju da informacije perzistiraju. Ovo ih čini idealnima za zadatke gdje je redoslijed podataka važan, poput obrade govora, jezika, vremenskih serija i slično.

1.8.1. Osnovna Arhitektura RNN-a

RNN se sastoji od ponavljajućih modula neurona, gdje svaki modul, osim standardnog ulaza i izlaza, ima i povratnu vezu koja šalje informacije o prethodnim stanjima naprijed kroz mrežu. Ova povratna veza omogućuje mreži da zadrži informacije o prethodnim podacima u sekvenci, što je ključno za obradu podataka koji imaju kontekstualni značaj.

Osnovni dijelovi RNN-a uključuju:

1. **Ulazni sloj**: Prima sekvencijalne podatke u obliku vektora.
2. **Skriveni sloj**: Sastoji se od neurona koji obrađuju podatke kroz vremenske korake, uzimajući u obzir prethodne i trenutne ulaze. Svaki korak koristi informacije iz prethodnog koraka kako bi ažurirao svoje stanje.
3. **Izlazni sloj**: Proizvodi rezultat za svaki vremenski korak ili za cijelu sekvencu, ovisno o vrsti zadatka.

1.8.2. Varijante RNN-a

Zbog problema poput eksplozije ili gubitka gradijenata, koji su česti u klasičnim RNN-ovima, razvijene su sofisticiranije varijante poput LSTM (Long Short-Term Memory) i GRU (Gated Recurrent Unit). Ove varijante koriste specijalne mehanizme poput "vrata" (gates) koji kontroliraju protok informacija kroz mrežu, čime se omogućuje bolja memorija i preciznost u zadacima s dugim ovisnostima.

1.8.3. Arhitektura EnhancedRNN_v3

Na temelju temeljnih koncepata RNN-a, arhitektura **EnhancedRNN_v3** koju sam implementirao proširuje osnovnu RNN strukturu korištenjem GRU slojeva, dropout slojeva za regularizaciju, i batch normalizacije kako bi se postigla robusnija i otpornija mreža.

```
class EnhancedRNN_v3(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers, 
dropout_prob=0.5):
        super(EnhancedRNN_v3, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers

        self.gru = nn.GRU(input_dim, hidden_dim, num_layers, 
batch_first=True, dropout=dropout_prob, bidirectional=True)
        self.dropout = nn.Dropout(dropout_prob)
        self.batch_norm = nn.BatchNorm1d(hidden_dim*2)

        self.classifier = nn.Sequential(
            nn.Linear(hidden_dim*2, hidden_dim),
            nn.ReLU(),
            nn.BatchNorm1d(hidden_dim),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.BatchNorm1d(hidden_dim),
            nn.Linear(hidden_dim, output_dim)
        )

    def forward(self, x):
            h0 = torch.zeros(self.num_layers*2, x.size(0), 
    self.hidden_dim).to(x.device)
        out, _ = self.gru(x, h0)
        out = self.dropout(out[:, -1, :])
        out = self.batch_norm(out)
        out = self.classifier(out)
        return out
    def get_l1_penalty(self):
        l1_norm = 0.0
        for param in self.parameters():
            l1_norm += torch.norm(param, p=1)
        return l1_norm
```

1.8.4. Ključni dijelovi EnhancedRNN_v3 arhitekture

1. **GRU sloj (Gted Recurrent Unit):**
   - Broj slojeva : Postavljeni su višestruki GRU slojevi kako bi se povećala sposobnost modela da uči složenije uzorke iz podataka. Svaki sloj ima svoj set skrivenih stanja koji se prenose kroz sekvence.
   - Bidirekcionalni GRU: Ova značajka omogućuje mreži da uči informacije iz oba smjera sekvence, što je korisno kada redoslijed podataka može utjecati na rezultat.
2. **Dropout sloj:**
   - Dropout je tehnika regularizacije koja se koristi za sprečavanje prenaučenosti modela. Tijekom treniranja, slučajno se isključuje određeni postotak neurona kako bi se model prisilio da generalizira bolje.
3. **Batch Normalizacija:**
   - Batch normalizacija se primjenjuje na izlaz GRU sloja, prije nego što se podaci proslijede dalje kroz mrežu. Ova tehnika stabilizira treniranje i omogućuje brže konvergiranje modela.
4. **Klasifikacijski sloj:**
   - Sastoji se od nekoliko potpuno povezanih (fully connected) slojeva sa ReLU aktivacijskim funkcijama i batch normalizacijom. Završni sloj daje konačni izlaz modela, što može biti klasifikacija u različite klase.
5. **L1 Regularizacija:**
   - Funkcija get_l1_penalty omogućava primjenu L1 regularizacije, koja može dodatno pomoći u sprječavanju prenaučenosti penaliziranjem velikih težina unutar modela.

1.8.5. Zašto RNN ?

Kao što je ranije opisano, RNN je odabran zbog svoje sposobnosti da obrađuje sekvencijalne podatke, poput govora. Klasične neuronske mreže ne mogu učinkovito koristiti informacije iz prethodnih vremenskih koraka, što je ključna prednost RNN-a. Međutim, kako bi se prevladali neki inherentni problemi RNN-a poput eksplozije ili gubitka gradijenata, odabrao sam GRU, koji je u EnhancedRNN_v3 integriran na način koji omogućuje dvosmjernu obradu i robusnu klasifikaciju.

EnhancedRNN_v3 model kombinira snagu GRU slojeva s dodatnim tehnikama regularizacije i normalizacije, čime se postiže ravnoteža između složenosti modela i sposobnosti generalizacije. Ova arhitektura je posebno prilagođena za zadatke gdje je

redoslijed informacija ključan, poput prepoznavanja govora, i osigurava da model može precizno obraditi i klasificirati složene sekvencijalne podatke.

1.9. Prvi neuspjeli pokušaj treniranja vlastitog modela

U razvoju modela za prepoznavanje glasovnih naredbi, prvi pokušaj korištenja prilagođenog rekurentnog neuronskog mreža (RNN) modela donio je niz poučnih izazova. Unatoč tehnički ispravnom postupku i primjeni naprednih tehnika učenja, model nije postigao zadovoljavajuću razinu generalizacije potrebnu za stvarne primjene. U nastavku je detaljno opisan proces treniranja, identificirani problemi i njihovi mogući uzroci.

1.9.1. Detaljan pregled implementacije modela

Model EnhancedRNN_v3 je dizajniran s ciljem efikasnog prepoznavanja sekvencijalnih zvučnih podataka korištenjem GRU (Gated Recurrent Unit) slojeva. GRU slojevi su odabrani zbog njihove sposobnosti da efikasno rješavaju probleme nestajućih gradijenata koji su česti kod standardnih RNN-ova. Model uključuje sljedeće komponente:

- **Dvostruki GRU slojevi** : omogućuju modelu da uči kako iz prošlih tako i iz budućih kontekstualnih informacija, čime se povećava razumijevanje trenutnog ulaznog podatka.
- **Dropout slojevi :** služe kao forma regularizacije koja nasumično isključuje neke od neurona tijekom treniranja, smanjujući time prenaučavanje na trening podacima.
- **Batch Normalization slojevi :** pomažu u stabilizaciji učenja normalizacijom ulaza za svaki sloj unutar mreže.
- **Klasifikatorski slojevi :** sastavljeni od potpuno povezanih slojeva i ReLU aktivacijskih funkcija koriste se za finalnu klasifikaciju ulaznih sekvencija.

```
class EnhancedRNN_v3(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim, num_layers, 
dropout_prob=0.5):
        super(EnhancedRNN_v3, self).__init__()
        self.hidden_dim = hidden_dim
        self.num_layers = num_layers
        self.gru = nn.GRU(input_dim, hidden_dim, num_layers, 
batch_first=True, dropout=dropout_prob,     bidirectional=True)
        self.dropout = nn.Dropout(dropout_prob)
        self.batch_norm = nn.BatchNorm1d(hidden_dim*2)
        self.classifier = nn.Sequential(
```

```
                nn.Linear(hidden_dim*2, hidden_dim),
                nn.ReLU(),
                nn.BatchNorm1d(hidden_dim),
                nn.Linear(hidden_dim, hidden_dim),
                nn.ReLU(),
                nn.BatchNorm1d(hidden_dim),
                nn.Linear(hidden_dim, output_dim)
            )

    def forward(self, x):
        h0 = torch.zeros(self.num_layers*2, x.size(0),
self.hidden_dim).to(x.device)
        out, _ = self.gru(x, h0)
        out = self.dropout(out[:, -1, :])
        out = self.batch_norm(out)
        out = self.classifier(out)
```

1.9.2. Proces treniranja

Treniranje modela provedeno je koristeći standardni pristup sa sljedećim koracima:

1. **Inicijalizacija stanja:** Za svaku epohu treniranja, početno stanje skrivenih slojeva resetira se kako bi se izbjeglo prenošenje informacija između batcheva.
2. **Optimizacija:** Korišten je optimizator Adam zbog njegove efikasnosti u radu s nestajućim gradijentima i automatskom prilagodbom stope učenja.
3. **Backpropagation i prilagodba težina:** Gradijenti se izračunavaju za svaku iteraciju i koriste za ažuriranje težina modela. Primijenjena je i tehnika ograničavanja gradijenata kako bi se spriječili preveliki koraci učenja koji mogu dovesti do nestabilnosti.
4. **Regularizacija:** L1 regularizacija dodana je kako bi se kontrolirala složenost modela, smanjujući težinu manje važnih veza unutar mreže.

```
model.train()
for epoch in range(epochs):
    train_loss = 0
    for x,y in train_loader:
        x = x.squeeze(1).transpose(1, 2)
        x = x.to(device)
        y = y.to(device).long()
        h = torch.zeros(num_layers, x.size(0), hidden_dim).to(device)
        optimizer.zero_grad()
```

```
            out = model(x)
            loss = loss_fn(out, y)
            l1_penalty = model.get_l1_penalty()
            loss.backward()
            clip_grad_norm_(model.parameters(), clip_value)
            optimizer.step()
            h = h.detach()
            #train_loss += loss.item()
            train_loss += lambda_l1 * l1_penalty
        train_loss /= len(train_loader)
        if epoch % 10 == 0:
            print(f"Epoch : {epoch} ,\nTrain Loss : {train_loss}")

        model.eval()
        with torch.inference_mode():
            val_loss=0
            for x,y in val_loader:
                x = x.squeeze(1).transpose(1, 2)
                x = x.to(device)
                y = y.to(device).long()
                h = torch.zeros(num_layers, x.size(0),
hidden_dim).to(device)
                out = model(x)
                loss = loss_fn(out, y)
                val_loss += loss.item()
            val_loss /= len(val_loader)
            if epoch % 10 == 0:
                print(f"Validation Loss : {val_loss}")
        model.train()
```

1.9.3. Izazovi i problemi

Unatoč impresivnoj točnosti od 97% koju je model pokazao na testnim podacima, prvi pokušaj korištenja prilagođenog RNN modela nije uspješno generalizirao u stvarnim aplikacijama. Ovo otkriće sugerira da postoji jaz između performansi modela u kontroliranim testnim uvjetima i njegove efikasnosti u realnom svijetu. Sljedeći faktori mogu objasniti zašto model nije uspio kako je očekivano:

1. **Problemi s oznakama i klasifikacijom:** Tijekom treninga, svih 100 zapisa za svaku komandu poput "Open web browser" bilo je označeno istom klasom bez obzira na razlike u izgovoru, intonaciji ili brzini govora. Ovaj pristup može dovesti do toga da model ne razlikuje suptilne varijacije unutar iste komande. Možda bi bilo korisnije implementirati hijerarhijski pristup klasifikaciji gdje se prvo razlikuju detaljni zapisi, a zatim se grupiraju pod općenitiju kategoriju (npr., sve varijante "open web browser" pod jednu kategoriju). Ovo bi zahtijevalo kreiranje dva klasifikatora: jednog za detaljnu klasifikaciju i drugog za generalizaciju na višoj razini.

2. **Nedostatak podataka:** Iako je 100 snimaka po komandi značajan broj, u kontekstu dubokog učenja i obrade prirodnog jezika, ovaj broj može biti nedovoljan za postizanje robusne generalizacije. Duboki modeli zahtijevaju velike količine podataka za učenje kako bi učinkovito apstrahirali značajke iz podataka i generalizirali na nove, neviđene primjere.

1.9.4. Evaluacija Modela

Pored izazova tijekom treniranja, važno je razumjeti i kako model funkcionira kad se evaluira na testnim podacima. Proces evaluacije je ključan za provjeru stvarne sposobnosti modela da ispravno klasificira neviđene podatke. Proces evaluacije uključuje sljedeće korake:

1. **Postavljanje Modela u Evaluacijski Način:** Pomoću model.eval(), model se postavlja u stanje koje deaktivira slojeve kao što su Dropout i BatchNorm, osiguravajući konzistentnost u performansama modela tijekom evaluacije.

2. **Izračunavanje Točnosti:** U evaluaciji, model obrađuje podatke iz testnog skupa i izlaz se uspoređuje s pravim oznakama. Koristeći torch.inference_mode(), što minimizira potrošnju memorije i ubrzava izračune, model predviđa klase za svaki ulazni zapis:

```
model.eval()
correct = 0
total = 0
with torch.inference_mode():
    for x, y in test_loader:
        x = x.squeeze(1).transpose(1, 2)
        x = x.to(device)
        y = y.to(device).long()
        h = torch.zeros(num_layers, x.size(0), hidden_dim).to(device)
        out = model(x)
```

```
_, predicted = torch.max(out.data, 1)
total += y.size(0)
correct += (predicted == y).sum().item()
```

1. **Prikaz Rezultata:** Na kraju procesa evaluacije, izračunava se i ispisuje postotak točnosti modela na testnim podacima kako bi se dobio jasan uvid u njegovu efikasnost:

   print('Test Accuracy of the model on the test data: {} %'.format(100 * correct / total))

Ova evaluacija pomaže identificirati kako model funkcionira s neviđenim podacima i pruža daljnji uvid u moguće probleme s generalizacijom koji su ranije spomenuti. Ako model pokazuje visoku točnost na testnim podacima, ali loše performanse u stvarnim aplikacijama, to dodatno potvrđuje postojanje problema s generalizacijom.

1.10. Nova strategija: Wav2Vec2

Nakon što sam se suočio s izazovima i ograničenjima prilikom prvog pokušaja izgradnje vlastitog modela za prepoznavanje glasovnih naredbi, bilo je jasno da je potreban drugačiji pristup kako bi se postigla bolja točnost i pouzdanost u stvarnim uvjetima. Problemi s lošom generalizacijom modela, usprkos visokim rezultatima na testnim podacima, pokazali su mi da jednostavno treniranje modela od nule možda nije najučinkovitiji način za rješavanje ovog problema. To me nagnalo da istražim alternativne metode koje bi mogle bolje iskoristiti postojeće resurse i omogućiti brži napredak u razvoju aplikacije za prepoznavanje glasovnih naredbi.

Jedan od pristupa koji je brzo privukao moju pažnju bio je fine-tuning (prilagodba) već postojećeg, visoko sofisticiranog modela za prepoznavanje govora zvanog Wav2Vec2. Ovaj model je razvijen od strane tima u Facebook AI i predstavlja jedan od najnaprednijih modela u području obrade govora. Wav2Vec2 koristi suvremene tehnike dubokog učenja i samostalnog učenja za izvlačenje značajki iz sirovih audio podataka, što ga čini izuzetno moćnim alatom za zadatke prepoznavanja govora.

Korištenje modela kao što je Wav2Vec2 nudi brojne prednosti, osobito u kontekstu rješavanja problema s kojima sam se suočio. Prvo, ovaj model je već prošao opsežnu fazu pretreniranosti na velikim skupovima podataka, što znači da je naučio složene obrasce i strukture u audio signalima. To znanje može se prilagoditi specifičnim zadacima kao što su

prepoznavanje određenih glasovnih naredbi, što smanjuje potrebu za prikupljanjem ogromnih količina podataka ili dugotrajnim treniranjem modela od nule.

1.10.1. Zašto Wav2Vec2 ?

Wav2Vec2 predstavlja vrhunac istraživanja u području obrade zvuka i automatiziranog prepoznavanja govora, koristeći duboke neuronske mreže za efikasno prepoznavanje govornih obrazaca iz sirovih audio podataka. Ovaj model, koji koristi metode dubokog nenadziranog učenja za prvu fazu treniranja, dokazao se kao iznimno uspješan u prepoznavanju i transkripciji govora s velikom točnošću i brzinom. Pred treniran na stotinama sati govornih podataka, Wav2Vec2 je usavršio sposobnost izvlačenja bogatih zvučnih značajki koje se mogu prilagoditi specifičnim potrebama kroz proces fine-tuninga.

1.10.2. Arhitektura Wav2Vec2

Arhitektura Wav2Vec2 je zasnovana na konceptu samostalnog učenja koje se fokusira na učenje korisnih reprezentacija iz neoznačenih audio podataka, nakon čega slijedi faza fine-tuninga za specifične zadatke poput transkripcije. Glavne komponente modela uključuju:

1. **Enkoder zvučnih valova**: Ova komponenta pretvara sirove zvučne valove u reprezentacije koje su bogate informacijama.

2. **Context network**: Nakon enkodiranja, reprezentacije se dalje obrađuju u kontekstualnoj mreži koja koristi mehanizme pažnje kako bi poboljšala kvalitetu značajki koristeći informacije iz cijelog audio zapisa.

3. **Klasifikator**: Na kraju, u fazi fine-tuninga, dodaje se klasifikator koji koristi naučene značajke za predviđanje točnih transkripcija ili klasa govora na temelju treniranih etiketa.

1.10.3. Prednosti Wav2Vec2 modela

Razlozi za odabir Wav2Vec2 modela su:

• **Napredna arhitektura**: Wav2Vec2 koristi slojevitu arhitekturu koja se sastoji od enkodera zvučnih valova i kontekstualne mreže koja poboljšava kvalitetu značajki korištenjem

samostalnog učenja. Ova arhitektura omogućava modelu da iz sirovih audio podataka izvuče bogate i informativne značajke koje su ključne za točno prepoznavanje govora.

• **Sposobnost za generalizaciju**: Zahvaljujući svojoj pretreniranosti na velikim skupovima podataka, Wav2Vec2 već ima razvijenu sposobnost generalizacije na razne govorne obrasce i jezike. Fine-tuning ovog modela za specifične glasovne naredbe omogućava da se ta sposobnost iskoristi i prilagodi specifičnim zahtjevima mog projekta.

• **Ušteda vremena i resursa**: Umjesto dugotrajnog treniranja vlastitog modela od nule, fine-tuning Wav2Vec2 omogućava da iskoristim već postojeće resurse i postignem visoke performanse uz relativno manje vremena i računalnih resursa. Ova strategija također smanjuje potrebu za velikim količinama podataka, što je značajna prednost u projektima koji imaju ograničen pristup velikim skupovima podataka.

• **Prilagodljivost specifičnim zadacima**: Iako je Wav2Vec2 pred treniran na općim skupovima podataka, fine-tuning omogućava prilagodbu modela specifičnim naredbama i jeziku koji se koriste u mom projektu. Ova prilagodljivost je ključna za postizanje visoke točnosti u prepoznavanju specifičnih glasovnih naredbi koje su relevantne za upravljanje Windows operativnim sustavom.

1.10.4. Proces Fine-tuninga Wav2Vec2

Korištenjem Wav2Vec2 modela kao osnove, krenuo sam u prilagođavanje modela za potrebe aplikacije prepoznavanja tri ključne glasovne naredbe: OpenWebBrowser, Search i WinMenu. Proces fine-tuninga uključuje sljedeće korake:

1. **Učitavanje Predtreniranog Modela**: Model Wav2Vec2ForCTC, koji koristi CTC loss idealan za sekvencijalne podatke, učitan je s predtrenirane konfiguracije facebook/wav2vec2-base-960h. Ovaj model već posjeduje sposobnost prepoznavanja širokog spektra govornih obrazaca.
2**. Priprema Podataka**: Audio snimci su prethodno obrađeni kako bi se uskladili s formatom prihvatljivim za Wav2Vec2. To uključuje konverziju audio zapisa u valne oblike i njihovu normalizaciju.
3. Adaptacija na Ciljne Naredbe: S obzirom na to da model već ima sposobnost prepoznavanja raznovrsnih audio obrazaca, glavni fokus fine-tuninga bio je prilagoditi ga za precizno prepoznavanje specifičnih naredbi relevantnih za aplikaciju. Svaki audio zapis je

označen odgovarajućom oznakom, a model je treniran da te oznake prepoznaje iz audio zapisa.

1.10.5. Prednosti pristupa fine-tuningom

Ova nova strategija fokusirana na fine-tuning Wav2Vec2 modela obećavala je da će riješiti mnoge probleme s kojima sam se suočavao u prvom pokušaju. Umjesto da se borim s izgradnjom i treniranjem vlastitog modela od nule, odlučio sam iskoristiti prednosti ovog moćnog modela i prilagoditi ga svojim potrebama. Ovaj pristup ne samo da je ubrzao razvoj mog projekta, već je također otvorio vrata za postizanje viših performansi u prepoznavanju glasovnih naredbi, što je ključno za uspjeh aplikacije koju razvijam.

1.11. Pogrešan pristup fine-tuningu Wav2Vec2

Nakon što sam odlučio prilagoditi prethodno istrenirani model Wav2Vec2 svojim potrebama kroz proces fine-tuninga, suočio sam se s novim izazovima koji su na kraju pokazali da moj pristup nije bio optimalan. S obzirom na pozitivna iskustva i uspjeh koji Wav2Vec2 pokazuje u zadacima prepoznavanja govora, očekivanja su bila visoka. Međutim, način na koji sam strukturirao svoje podatke i definirao klasifikacijske oznake doveo je do problema s točnošću i pouzdanošću modela, slično kao što je bio slučaj s prethodnim pokušajem izgradnje vlastitog klasifikatora.

1.11.1. Ograničenja u strukturiranju podataka

Glavni problem ležao je u načinu na koji sam označio svoje podatke za trening. Umjesto da precizno diferenciram različite varijante istih glasovnih naredbi, sve sam ih sveo na samo tri klase. Na primjer, audio zapisi koji sadrže naredbe poput "Open web browser" i "Open internet" svi su bili označeni istom oznakom "OpenWebBrowser". Ovaj pristup je pojednostavio proces treniranja, ali je donio niz problema kada je model trebao generalizirati u stvarnim uvjetima.

1.11.2. Posljedice jednostavne klasifikacije

Ovaj pojednostavljeni pristup označavanju podataka može funkcionirati u scenarijima gdje imate obilje podataka i gdje su varijacije između različitih naredbi minimalne. Međutim, u

mom slučaju, gdje je količina podataka bila ograničena, takav pristup se pokazao problematičnim. Konkretno:

1. Nedostatak raznolikosti u podacima: Zbog malog broja klasa i ograničenog skupa podataka, model nije mogao dovoljno diferencirati između različitih varijanti unutar iste klase. To je dovelo do situacija u kojima model nije mogao precizno prepoznati specifične naredbe, već bi često klasificirao slične zvučne signale kao istu klasu, čak i kada su oni značajno različiti u stvarnoj aplikaciji.

2. Visoka sličnost između različitih klasa: Kada se slični zvučni zapisi grupiraju pod istu oznaku, model počinje učiti da su svi slični zvukovi povezani s istom akcijom. Na primjer, ako korisnik izgovori "Open browser" i "Open settings", oba zapisa mogu biti dovoljno slična da ih model klasificira kao "OpenWebBrowser", što nije ispravna akcija. Ovo je posebno problematično u stvarnim aplikacijama gdje korisnik očekuje preciznost i točnost.

3. Potreba za velikim brojem podataka: Kada se koriste široke klase koje obuhvaćaju više različitih varijanti glasovnih naredbi, modelu je potrebno puno više podataka kako bi pouzdano naučio različite obrasce unutar tih klasa. U mom slučaju, s ograničenim skupom podataka, model nije imao dovoljno informacija da razlikuje slične, ali različite naredbe. Ovo je uzrokovalo preklapanje između klasa i dovelo do pogrešnih klasifikacija.

4. Mogući scenariji brkanja u stvarnim aplikacijama: U stvarnim uvjetima korištenja, kada korisnik izgovori naredbu koja nije savršeno u skladu s onim što je model naučio, postoji velika vjerojatnost da će model pogrešno klasificirati tu naredbu kao drugu poznatu naredbu koja zvuči slično. To se događa zato što model nije naučio razlikovati sve varijante unutar šire klase, već je naučio da sve što zvuči slično pripada istoj akciji.

1.11.3. Lekcije naučene

Ovaj pogrešan pristup fine-tuningu bio je važna lekcija o tome kako strukturiranje podataka može drastično utjecati na performanse modela. Umjesto pojednostavljivanja kroz reduciranje broja klasa, u mom slučaju bi bilo bolje razmotriti složeniji pristup s više specifičnih oznaka, a zatim integrirati vise stupanjski klasifikator koji bi prvo razlikovao fine varijante unutar istih klasa, a zatim grupirao te varijante u šire akcijske klase.

Također, ovaj problem me podsjetio na važnost balansiranja između jednostavnosti i preciznosti u treniranju modela. Iako je privlačno pokušati pojednostaviti problem grupiranjem sličnih podataka, takav pristup može rezultirati smanjenjem sposobnosti modela da pouzdano generalizira, osobito kada su podaci ograničeni.

U narednim koracima, nužno je istražiti načine za poboljšanje strukture podataka i označavanja kako bi se povećala preciznost modela, te razmotriti alternativne strategije koje mogu bolje iskoristiti prednosti fine-tuninga, ali na način koji omogućuje točno prepoznavanje specifičnih glasovnih naredbi u stvarnim aplikacijama.

1.12. Novi pristup Fine-tuningu Wav2Vec2

Nakon što sam prepoznao ozbiljne nedostatke u prethodnom pristupu fine-tuningu modela Wav2Vec2, shvatio sam da je potreban potpuno novi način strukturiranja podataka kako bih poboljšao performanse modela. Prvobitni pokušaj, gdje sam sve glasovne naredbe grupirao u samo tri široke klase (OpenWebBrowser, Search, WinMenu), pokazao je da takav pristup nije dovoljno precizan za učinkovitu primjenu u stvarnim uvjetima. Iako je model bio sposoban prepoznati određene obrasce, nije mogao točno razlikovati slične, ali različite naredbe, što je dovodilo do pogrešnih klasifikacija i smanjene pouzdanosti u realnim scenarijima. Stoga sam odlučio radikalno promijeniti svoj pristup i detaljno restrukturirati podatke, što je uključivalo niz važnih koraka i odluka.

1.12.1. Restrukturiranje podataka za fine-tuning

U ranijem pristupu, sve varijacije glasovnih naredbi koje su se mogle izgovoriti za otvaranje web preglednika, pretragu ili otvaranje izbornika Windows, bile su grupirane pod samo tri opće oznake. Na primjer, svi zapisi koji su sadržavali naredbe poput "Open web browser" i "Open internet" imali su zajedničku oznaku "OpenWebBrowser". Ovaj pristup se naizgled činio jednostavnim i logičnim, ali je zapravo maskirao ključne razlike između različitih naredbi, što je modelu otežavalo točno razlikovanje ovih naredbi u stvarnim uvjetima.

Novi pristup temeljio se na ideji da je potrebno preciznije diferencirati glasovne naredbe, tako da svaki audio zapis dobije svoju specifičnu oznaku koja točno opisuje što se izgovara u tom zapisu. Ovaj pristup omogućava modelu da nauči ne samo opće obrasce, već i suptilne razlike između sličnih naredbi. Tako sada, umjesto samo tri široke klase, postoji mnogo više specifičnih oznaka, kao što su "Open web browser", "Launch web browser", "Start browser", i tako dalje. Svaka od tih oznaka precizno odgovara određenoj naredbi koja se može izgovoriti, što omogućava modelu da detaljnije i preciznije prepozna što korisnik zapravo želi postići.

1.12.2. Prilagodba i generiranje novih podataka

Kako bih proveo ovu promjenu, bilo je potrebno restrukturirati podatke na više razina, uključujući generiranje novih, sintetičkih podataka s greškama te dodavanje nove klase za prepoznavanje pogrešnih naredbi.

1. **Generiranje sintetičkih podataka s greškama:** Svjestan sam da se u stvarnom svijetu korisnici često ne izgovaraju savršeno jasno ili da mogu pogrešno izgovoriti određene riječi. Kako bih model prilagodio ovakvim situacijama, razvio sam proces generiranja sintetičkih grešaka u transkriptima. Na primjer, naredba "Open web browser" može biti pogrešno prepoznata kao "Open web browsqer". Ovakve greške su u model unesene namjerno kako bi on mogao naučiti prepoznavati i ispravljati manje greške u prepoznavanju govora.

```
Def
generate_synthetic_dataset(transcript,synthetic_dataset,label,num_err
ors=2):
    for command in transcript:
        for _ in range(50):
            misrecognized_command =
generate_synthetic_errors(command,num_errors)
            synthetic_dataset.append((misrecognized_command,label))
```

Ova strategija osigurava da model ne samo da prepoznaje savršeno izgovorene naredbe, već i one koje sadrže manje pogreške ili varijacije u izgovoru. Time se povećava njegova otpornost i sposobnost generalizacije na različite vrste govora.

2. **Promjena naziva datoteka u skladu s transkriptima:** Jedan od tehničkih izazova bio je osigurati da svaka zvučna datoteka bude ispravno povezana s odgovarajućim transkriptom. Kako bih to postigao, koristio sam kod za preimenovanje datoteka tako da njihovo ime odgovara transkriptu koji opisuje što se točno kaže u tom zapisu. Na primjer, ako je transkript "Open web browser", tada je zvučna datoteka preimenovana u "Open web browser.wav". Ova precizna povezanost između datoteka i transkripta bila je ključna za točnost modela, jer je osigurala da se model trenira na točno odgovarajućim podacima.

```
def transcribe_audio_v3(folder_input, folder_output, transcript_list,
file_list):
    for i in tqdm(range(len(transcript_list))):
        file_path = os.path.join(folder_input, file_list[i])
        transcript = transcript_list[i]
        print(f"Processing: {file_path}, {transcript}")
        new_file_name = f"{transcript}.wav"
```

```
        new_file_path = os.path.join(folder_output, new_file_name)
        counter = 1
        while os.path.exists(new_file_path):
            new_file_name = f"{transcript}_{counter}.wav"
            new_file_path = os.path.join(folder_output, new_file_name)
            counter += 1
        shutil.copy(file_path, new_file_path)
        print(f"Copied to: {new_file_path}")
```

Ovaj proces je omogućio preciznu organizaciju podataka, čime se dodatno poboljšala točnost modela u prepoznavanju specifičnih naredbi.

3. **Dodavanje "Bad Command" klase:** Jedan od ključnih dodataka u novom pristupu bio je uvođenje klase "BadCommand". Ova klasa je dizajnirana za prepoznavanje slučajeva u kojima korisnik izgovori naredbu koja nije predviđena ili koja nema smisla. U tu svrhu generirao sam sintetičke podatke koji predstavljaju nasumične ili kratke nizove znakova, kako bi model mogao naučiti prepoznati i razlikovati ove besmislene naredbe od pravih, validnih naredbi.

```
def generate_random_string(length):
    return ''.join(random.choices(string.ascii_lowercase, k=length))
def generate_bad_commands(num_commands, max_length=10):
    bad_commands = []
    for _ in range(num_commands):
        # Randomly choose the type of bad command to generate
        command_type = random.choice(['random', 'short'])
        if command_type == 'random':
            length = random.randint(1, max_length)
            bad_commands.append(generate_random_string(length))
        elif command_type == 'short':
            length = random.randint(1, 3)
            bad_commands.append(generate_random_string(length))
    return bad_commands
def generate_synthetic_dataset_with_bad_commands(synthetic_dataset,
num_bad_commands=1000):
    bad_commands = generate_bad_commands(num_bad_commands)
    for bad_command in bad_commands:
      synthetic_dataset.append((bad_command, 'BadCommand'))
```

Generiranjem ovih "Bad Command" podataka, model je postao sposobniji prepoznati kada je primio nevažeću ili besmislenu naredbu, što je kritično za osiguravanje pouzdanosti sustava u stvarnim uvjetima.

1.12.3. Prednosti novog pristupa

Ovaj novi pristup restrukturiranju podataka i fine-tuningu modela Wav2Vec2 donosi nekoliko ključnih prednosti:

- **Povećana preciznost i pouzdanost:** Kroz detaljniju klasifikaciju i preciznije oznake, model sada može naučiti finije razlike između različitih naredbi, što značajno smanjuje vjerojatnost pogrešne klasifikacije. Ovo je posebno važno u situacijama gdje slične naredbe mogu imati vrlo različite rezultate.
- **Otpornost na varijacije i greške:** Generiranjem sintetičkih grešaka i dodavanjem "BadCommand" klase, model postaje otporniji na uobičajene greške u prepoznavanju govora, kao i na nesavršeno izgovorene naredbe. Time se povećava njegova sposobnost da točno prepozna korisnikove namjere, čak i u slučaju manjih grešaka.
- **Bolja generalizacija:** S preciznijim oznakama i bolje strukturiranim podacima, model ima veće šanse za generalizaciju na nove, neviđene podatke, čime se poboljšava njegova upotrebljivost u različitim kontekstima i aplikacijama.

Ovaj novi pristup predstavlja značajan napredak u odnosu na prethodni pokušaj. Umjesto jednostavnog grupiranja sličnih naredbi pod jedinstvene oznake, ovaj detaljan i precizan način strukturiranja podataka omogućava modelu da se prilagodi specifičnim potrebama aplikacije za glasovno upravljanje Windows operativnim sustavom. Na taj način, model postaje robusniji, pouzdaniji i spremniji za primjenu u stvarnim uvjetima, gdje preciznost i pouzdanost igraju ključnu ulogu.

1.13. Fine-tuning Wav2Vec2.0 CTC Modela

Nakon što sam se odlučio za novi pristup u fine-tuningu modela Wav2Vec2.0, ključno je bilo osigurati da su podaci i model pravilno pripremljeni i optimizirani kako bi se postigli najbolji mogući rezultati u prepoznavanju specifičnih glasovnih naredbi. Ovaj proces uključivao je više koraka, uključujući pripremu podataka, augmentaciju, prilagodbu modela, te implementaciju i evaluaciju samog procesa treniranja. Cilj je bio stvoriti model koji može precizno i pouzdano prepoznati glasovne naredbe u stvarnim uvjetima, unatoč varijacijama u govoru, šumovima u pozadini i drugim izazovima.

1.13.1. Priprema podataka za fine-tuning

1. Priprema okruženja i osnovni importi

Kao prvi korak, učitani su svi potrebni moduli i biblioteke koje će se koristiti tijekom pripreme podataka i treniranja modela. Ove biblioteke uključuju torch, librosa, torchaudio, transformers, i druge, koje su ključne za manipulaciju audio podacima, treniranje modela i optimizaciju performansi.

```
import torch
import os
import pathlib
import matplotlib
import glob
import librosa
import torch
import torch.nn as nn
import numpy as np
import torch.nn.functional as F
import math
import pandas as pd
import torchaudio
import time
import re
import json
from torch.hub import load_state_dict_from_url
from torch.nn.utils import clip_grad_norm_
from torchaudio import transforms,load as torch_load
from IPython.display import Audio
from torch.utils.data import Dataset, DataLoader, TensorDataset
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from torch.nn import GRU, Softmax, Linear, Dropout, LayerNorm
from os import listdir
from matplotlib import pyplot as plt
from pathlib import Path
from io import BytesIO
from librosa import display, load, stft, amplitude_to_db
from numpy import mean, std
from torch.nn.utils.rnn import pad_sequence
from sklearn.model_selection import KFold
from torchaudio.transforms import Resample, TimeMasking,
FrequencyMasking
```

```
from transformers import Wav2Vec2Processor, Wav2Vec2FeatureExtractor, Wav2Vec2CTCTokenizer, TrainerCallback, Trainer, TrainingArguments, Wav2Vec2ForCTC, is_torch_tpu_available, get_constant_schedule
from datasets import Dataset as HFDataset
from torch.nn.utils import clip_grad_norm_
from transformers.trainer_pt_utils import get_parameter_names
from transformers.trainer_utils import TrainOutput, PREFIX_CHECKPOINT_DIR
from transformers.trainer_pt_utils import get_parameter_names
```

2. Audio Utility Klasa za Obradu Podataka

Razvijena je klasa AudioUtils_Transformer koja sadrži niz statičnih metoda za obradu audio podataka. Ove metode uključuju otvaranje audio datoteka, promjenu broja kanala, resampliranje, obrezivanje i dodavanje vremenskih pomaka, ubrzavanje ili usporavanje audio zapisa, dodavanje buke, promjenu glasnoće, dodavanje eha, te obrtanje audio signala.

○ Otvaranje audio datoteke (open) koristi torchaudio za učitavanje audio signala i brzine uzorkovanja.

○ Promjena broja kanala (rechanel) omogućava promjenu broja kanala u audio signalu (mono/stereo).

○ Resampliranje (resample) prilagođava brzinu uzorkovanja audio signala kako bi odgovarala specifikacijama modela.

○ Obrezivanje i proširivanje (pad_trunc) osigurava da svi audio zapisi imaju istu dužinu.

○ Ubrzavanje/Uspravljanje (change_speed) koristi librosa za promjenu brzine audio signala, što može pomoći u augmentaciji podataka.

○ Dodavanje eha (add_echo) i Obrtanje audio signala (reverse) dodatno obogaćuju podatke generiranjem varijacija u audio signalima.

```
class AudioUtils_Transformer:
    @staticmethod
    def open(audio_file):
        sig, sr = torchaudio.load(audio_file)
        return sig, sr
    @staticmethod
    def rechanel(aud, new_channel):
        sig, sr = aud
        if sig.shape[0] == new_channel:
```

```
            return aud
        if new_channel == 1:
            resig = sig[:1, :]
        else:
            resig = torch.cat([sig, sig])
        return (resig, sr)
    @staticmethod
    def resample(aud, new_sr):
        sig, sr = aud
        if sr == new_sr:
            return aud
        resig = Resample(sr, new_sr)(sig)
        return (resig, new_sr)
    @staticmethod
    def pad_trunc(aud, max_ms):
        sig, sr = aud
        num_rows, sig_len = sig.shape
        max_len = sr // 1000 * max_ms
        if sig_len > max_len:
            sig = sig[:, :max_len]
        elif sig_len < max_len:
            pad_begin_len = np.random.randint(0, max_len - sig_len)
            pad_end_len = max_len - sig_len - pad_begin_len
            pad_begin = torch.zeros((num_rows, pad_begin_len))
            pad_end = torch.zeros((num_rows, pad_end_len))
            sig = torch.cat([pad_begin, sig, pad_end], 1)
        return (sig, sr)
    @staticmethod
    def aug_time_shift(aud, shift_limit):
        sig, sr = aud
        _, sig_len = sig.shape
        shift_amt = int(np.random.random() * shift_limit * sig_len)
        return (sig.roll(shift_amt), sr)
    @staticmethod
    def inject_noise(signal, noise_factor=0.01):
        sig, sr = signal
        noise = torch.randn_like(sig)
        augmented_signal = sig + noise_factor * noise
        return (augmented_signal, sr)
    @staticmethod
```

```
    def random_gain(signal, min_gain=0.8, max_gain=1.2):
        sig, sr = signal
        gain = torch.rand(1) * (max_gain - min_gain) + min_gain
        augmented_signal = sig * gain
        return (augmented_signal, sr)
    @staticmethod
    def change_speed(signal, speed_factor=1.1):
        sig, sr = signal
        np_signal = sig.numpy()
        speed_changed_signal=librosa.effects.time_stretch(np_signal,
rate=speed_factor)
        if speed_changed_signal.shape[1] > sig.shape[1]:
            speed_changed_signal = speed_changed_signal[:,:sig.shape[1]]
        elif speed_changed_signal.shape[1] < sig.shape[1]:
             padding = sig.shape[1] - speed_changed_signal.shape[1]
             speed_changed_signal = np.pad(speed_changed_signal,((0, 0),
(0, padding)), mode='constant')
        return (torch.from_numpy(speed_changed_signal), sr)
    @staticmethod
    def add_echo(signal, delay=0.1, attenuation=0.5):
        sig, sr = signal
        sig = sig.numpy()
        echo_length = int(delay * sr)
        echo = np.pad(sig, ((0, 0),(echo_length, 0)),mode='constant',
constant_values=0) * attenuation

        echo = echo[:, :sig.shape[1]]
        echo_sig = sig + echo
        if echo_sig.shape[1] > sig.shape[1]:
            echo_sig = echo_sig[:, :sig.shape[1]]
        elif echo_sig.shape[1] < sig.shape[1]:
            padding = sig.shape[1] - echo_sig.shape[1]
            echo_sig=np.pad(echo_sig,((0,0),(0,padding)),
mode='constant')

        return (torch.from_numpy(echo_sig), sr)
    @staticmethod
    def reverse(signal):
        sig, sr = signal
        reversed_sig = sig.flip(dims=[1])
        return (reversed_sig, sr)
```

3. Priprema podataka za treniranje

Nakon obrade audio zapisa pomoću AudioUtils_Transformer klase, podaci su organizirani i pripremljeni za treniranje. Prvo su podaci podijeljeni na trening, validacijski i testni skup koristeći train_test_split funkciju. Također je implementirana funkcija clean_labels za normalizaciju oznaka uklanjanjem dodatnih informacija iz naziva datoteka, poput brojeva i ekstenzija.

```
device = torch.device('cuda')
print(f"GPU is available :  {torch.cuda.is_available()}")
print(f"GPU : {torch.cuda.get_device_name(0)}")
coder = LabelEncoder()
training_data_path,target_data_path =
Path("../TrainingData"),Path("../TrainingData_Transcribed_Converted")
files_to_convert = []
training_files = listdir(training_data_path)
for file in training_files:
    files_to_convert.append(file)
data = []
sr = 16000
channel = 1
duration = 2000
shift_pct = 0.2
for file in files_to_convert:
    audio_folder_path = Path(target_data_path / file)
    files = listdir(audio_folder_path)
    for audio_file in files:
        audio_path = str(audio_folder_path / audio_file)
        data.append({'path': audio_path,'class': file})
df = pd.DataFrame(data)
data_model,label_model = [], []
for row in df.iterrows():
    index, data = row
    data_model.append(data['path'])
    label_model.append(data['class'])
def clean_labels(labels):
    cleaned_labels = []
    for label in labels:
        label = label.replace('.wav', '')
        label = re.sub(r'_\d+', '', label)
```

```
                cleaned_labels.append(label)
            return cleaned_labels
        label_model = clean_labels(label_model)


        torch.manual_seed(42)
        data_train, data_test, label_train, label_test = 
train_test_split(data_model, label_model, test_size=0.2, random_state=42)
        data_train, data_val, label_train, label_val = 
train_test_split(data_train, label_train, test_size=0.3, random_state=42)
```

4. Pretvorba audio podataka u prikladan format

Korištenjem funkcije prepare_dataset, audio podaci su pretvoreni u nizove koji su prikladni za ulaz u model. Ova funkcija koristi metodu audio_to_list za pretvorbu audio signala u liste brojeva koje predstavljaju audio valove.

```
def audio_to_list(audio):
    return audio[0].numpy().flatten().tolist()
def prepare_dataset(data, texts, sr=16000, channel=1, duration=2000):
    data_wav = []
    for path in data:
        aud = AudioUtils_Transformer.open(path)
        resampled_aud = AudioUtils_Transformer.resample(aud, sr)
        rechanneled_aud = 
AudioUtils_Transformer.rechanel(resampled_aud, channel)
        padded_aud = 
AudioUtils_Transformer.pad_trunc(rechanneled_aud, duration)
        data_wav.append(audio_to_list(padded_aud))
    return data_wav, texts


data_test_wav, texts_test  = prepare_dataset(data_test, label_test)
data_val_wav, texts_val  = prepare_dataset(data_val, label_val)
data_train_wav, texts_train  = prepare_dataset(data_train, 
label_train)
```

5. Normalizacija tekstualnih podataka

Nakon pripreme audio podataka, tekstualni podaci su normalizirani uklanjanjem specijalnih znakova. Ovo je ključno za osiguranje konzistentnosti tijekom treniranja modela.

```
texts_train = [remove_special_characters({"text": text})["text"] for 
text in texts_train]
texts_test = [remove_special_characters({"text": text})["text"] for 
text in texts_test]
```

```
    texts_val = [remove_special_characters({"text": text})["text"] for
text in texts_val]
```

6. Pretvorba podataka u HuggingFace Dataset format

Podaci su zatim konvertirani u HFDataset format, što omogućuje njihovu jednostavnu uporabu u Trainer API-ju iz transformers biblioteke.

```
    train_df = pd.DataFrame({"audio": data_train_wav, "text":
texts_train})
    val_df = pd.DataFrame({"audio": data_val_wav, "text": texts_val})
    test_df = pd.DataFrame({"audio": data_test_wav, "text": texts_test})
    train_dataset = HFDataset.from_dict({"audio": data_train_wav, "text":
texts_train})
    val_dataset = HFDataset.from_dict({"audio": data_val_wav, "text":
texts_val})
    test_dataset = HFDataset.from_dict({"audio": data_test_wav, "text":
texts_test})
```

7. Priprema vokabulara i procesora

Kreiran je vokabular iz svih tekstualnih podataka, koji se koristi za treniranje Wav2Vec2 modela. Vokabular se zatim sprema u JSON datoteku i koristi se za inicijalizaciju Wav2Vec2CTCTokenizer i Wav2Vec2Processor.

```
    def extract_all_chars(dataset):
        all_text = " ".join(dataset["text"])
        vocab = list(set(all_text))
        return vocab
    all_vocab = extract_all_chars(train_dataset)

    vocab_list = sorted(list(all_vocab))
    vocab_dict = {v: k for k, v in enumerate(vocab_list)}
    vocab_dict["|"] = vocab_dict.get(" ", len(vocab_dict))
    if " " in vocab_dict:
        del vocab_dict[" "]
    vocab_dict["[UNK]"] = len(vocab_dict)
    vocab_dict["[PAD]"] = len(vocab_dict)

    with open('vocab.json', 'w') as vocab_file:
        json.dump(vocab_dict, vocab_file)
    tokenizer = Wav2Vec2CTCTokenizer("results_wav2vec2ForCTC/vocab.json",
unk_token="[UNK]", pad_token="[PAD]", word_delimiter_token="|")
```

```
    feature_extractor = Wav2Vec2FeatureExtractor(feature_size=1,
sampling_rate=16000, padding_value=0.0, do_normalize=True,
return_attention_mask=False)
    processor = Wav2Vec2Processor(feature_extractor=feature_extractor,
tokenizer=tokenizer)
    special_tokens = {"additional_special_tokens": ["<badcmd>"]}

    processor.tokenizer.add_special_tokens(special_tokens)
```

8. Fine-tuning modela

Sam proces fine-tuninga započinje inicijalizacijom Wav2Vec2 modela(Wav2Vec2ForCTC) i definiranjem funkcije za predobradu koja priprema audio podatke i tekst za ulaz u model. Nakon toga, konfigurira se Trainer klasa koja se koristi za treniranje modela.

```
    model = Wav2Vec2ForCTC.from_pretrained("facebook/wav2vec2-base-960h")
    def preprocess_function_v6(examples):
        audio = examples["audio"]
        inputs = processor(audio, sampling_rate=16000,
return_tensors="pt", padding=True)
        with processor.as_target_processor():
            if examples["text"] == "bad command":
                labels = processor("<badcmd>", padding=True,
return_tensors="pt").input_ids
            else:
                labels = processor(examples["text"], padding=True,
return_tensors="pt").input_ids

        labels[labels == processor.tokenizer.pad_token_id] = -100

        attention_mask = inputs.attention_mask if "attention_mask" in
inputs else torch.ones_like(inputs.input_values, dtype=torch.bool)

        return {
            "input_values": inputs.input_values.squeeze(0),
            "attention_mask": attention_mask.squeeze(0),
            "labels": labels.squeeze(0)
        }

    train_dataset = train_dataset.map(preprocess_function_v6,
remove_columns=["audio"])
    val_dataset = val_dataset.map(preprocess_function_v6,
remove_columns=["audio"])
```

```
    test_dataset = test_dataset.map(preprocess_function_v6,
remove_columns=["audio"])
```

Za treniranje modela koristi se Trainer klasa iz transformers biblioteke. Definiraju se parametri poput broja epoha, veličine batcha, strategije evaluacije, i ostalih ključnih elemenata treniranja.

```
    def data_collator_v1(features):
        input_values = pad_sequence([torch.tensor(f["input_values"],
dtype=torch.float32) for f in features], batch_first=True)
        attention_mask = pad_sequence([torch.tensor(f["attention_mask"],
dtype=torch.long) for f in features], batch_first=True, padding_value=0)
        label_padded = pad_sequence([torch.tensor(f["labels"],
dtype=torch.long) for f in features], batch_first=True, padding_value=-100)

        return {
            "input_values": input_values,
            "attention_mask": attention_mask,
            "labels": label_padded
        }

    model.to(device)
    training_args = TrainingArguments(
        output_dir='./results_wav2vec2ForCTC',
        num_train_epochs=40,
        per_device_train_batch_size=2,
        per_device_eval_batch_size=2,
        warmup_steps=1000,
        weight_decay=0.005,
        logging_dir='./logs',
        gradient_checkpointing=True,
        logging_steps=10,
        evaluation_strategy="steps",
        eval_steps=500,
        learning_rate=2e-5,
        gradient_accumulation_steps=4,
        save_total_limit=2,
        fp16=torch.cuda.is_available(),
        load_best_model_at_end=True,
        max_grad_norm=1.0
    )

    trainer = Trainer(
```

```
        model=model,
        args=training_args,
        train_dataset=train_dataset,
        eval_dataset=val_dataset,
        tokenizer=processor.feature_extractor,
        data_collator=data_collator_v1,
        callbacks=[gradient_clipping_callback]
    )

    trainer.train()

    model.save_pretrained(training_args.output_dir)
    processor.save_pretrained(training_args.output_dir)
```

Proces fine-tuninga Wav2Vec2.0 modela bio je sveobuhvatan, uključujući složenu pripremu podataka, normalizaciju tekstualnih podataka, te precizno treniranje modela koristeći odgovarajuće hiperskalarne parametre. Korištenjem ovih koraka, model je postigao visoku točnost u prepoznavanju i transkripciji glasovnih naredbi, što ga čini ključnom komponentom u ukupnom sustavu za prepoznavanje govora.

1.14. 3-stage Model: YAMNet, Wav2Vec2.0 i DistilBERT

Nakon uspješnog fine-tuninga modela Wav2Vec2.0 na novo strukturiranim podacima, postigao sam značajno poboljšanje u prepoznavanju glasovnih naredbi. Međutim, korištenjem u stvarnom vremenu, došlo je do pojave novih izazova. Glavni problem s kojim sam se suočio bio je određivanje trenutka kada model treba prepoznati glasovnu naredbu korisnika. U idealnom slučaju, model bi trebao započeti prepoznavanje tek kada detektira da je korisnik izgovorio određenu naredbu. Da bi to bilo moguće, bilo je potrebno implementirati sustav koji prvo prepoznaje ljudski glas prije nego što pokrene prepoznavanje naredbi. Ovaj problem riješio sam dizajniranjem tro-stupanjskog modela koji se sastoji od tri ključna dijela:

1. **IsHumanVoice** - Prepoznavanje ljudskog glasa pomoću YAMNet
2. **DetermineSpokenWord** - Prepoznavanje izgovorene riječi pomoću Wav2Vec2
3. **ClassifySpokenWord** - Klasifikacija prepoznate fraze pomoću DistilBERT

1.14.1. Prvi korak: IsHumanVoice (YAMNet)

Prvi korak u trostupanjskom modelu je prepoznavanje ljudskog glasa. Ovdje se koristi YAMNet model koji je razvijen od strane Googlea, a dostupan je putem TensorFlow Huba

(hub.load('https://tfhub.dev/google/yamnet/1')). YAMNet je model koji je treniran na velikom broju audio zapisa i može prepoznati različite vrste zvukova, uključujući ljudski govor. U mom slučaju, YAMNet služi kao filter koji odlučuje hoće li se zvučni zapis dalje obrađivati.

Proces rada YAMNeta:

• Kada model primi audio zapis, prvo koristi YAMNet kako bi odredio je li zvuk ljudski glas. YAMNet prolazi kroz cijeli audio zapis i klasificira vrste zvukova prisutnih u zapisu.

• Ako YAMNet prepozna ljudski glas, tada se taj audio zapis prosljeđuje dalje u proces, gdje će ga analizirati Wav2Vec2.0 za prepoznavanje izgovorene riječi ili fraze.

• Ako YAMNet ne detektira ljudski glas, zvučni zapis se odbacuje, čime se izbjegava nepotrebna obrada zapisa koji nisu relevantni.

Prednosti upotrebe YAMNeta:

• Efikasnost: Koristeći YAMNet kao prvi filter, osiguravam da se samo relevantni audio zapisi dalje obrađuju, čime štedim računalne resurse.

• Preciznost: Time se smanjuje broj lažno pozitivnih rezultata, jer samo zapisi koji sadrže ljudski glas idu na daljnju analizu.

1.14.2. Drugi korak: DetermineSpokenWord (Wav2Vec2.0)

Nakon što je YAMNet potvrdio da audio zapis sadrži ljudski glas, taj se zapis prosljeđuje Wav2Vec2.0 modelu koji je prethodno fine-tuniran na posebno strukturiranim podacima. Ovaj model prepoznaje što je točno izgovoreno u audio zapisu. Wav2Vec2.0 je već bio detaljno opisan u prethodnim dijelovima, ali ukratko:

Proces rada Wav2Vec2.0:

• Wav2Vec2.0 prima audio signal i koristi svoje složene neuronske mreže za ekstrakciju značajki i prepoznavanje govora.

• Rezultat Wav2Vec2.0 modela je tekstualna transkripcija onoga što je korisnik izgovorio. Na primjer, ako korisnik kaže "Open web browser", Wav2Vec2.0 će to prepoznati i pretvoriti u odgovarajući tekst.

Prednosti korištenja Wav2Vec2.0:

• Točnost: Wav2Vec2.0 je jedan od najnaprednijih modela za prepoznavanje govora, a fine-tuning omogućuje preciznu prilagodbu specifičnim zahtjevima aplikacije.

• Robustnost: Model je treniran na velikom broju podataka, što mu omogućuje da precizno prepoznaje govor čak i u slučajevima kada korisnik ne izgovara naredbu savršeno jasno.

1.14.3. Treći korak: ClassifyItSpokenWord (DistilBERT)

Nakon što Wav2Vec2.0 model prepozna što je točno izgovoreno, potrebno je tu transkripciju klasificirati u odgovarajuću naredbu. Ovdje dolazi na scenu treći model, DistilBERT (distilbert-base-uncased), koji je također fine-tuniran na posebno pripremljenim podacima.

Strukturiranje podataka za DistilBERT:

• Podaci su strukturirani u četiri klase: OpenWebBrowser, Search, WinMenu, i BadCommand. Svaka klasa sadrži sintetičke transkripcije generirane u prethodnim koracima, kao što su browser_synthetic_dataset.csv, search_synthetic_dataset.csv, winMenu_synthetic_dataset.csv, i bad_command_synthetic_dataset.csv.

• U ovim CSV datotekama nalaze se transkripti audio zapisa, očišćeni od razmaka i velikih slova, kako bi bili ujednačeni za obradu u DistilBERT modelu.

Proces rada DistilBERT-a:

• Kada DistilBERT primi tekstualnu transkripciju iz Wav2Vec2.0 modela, analizira ga i klasificira u jednu od četiri definirane klase. Na primjer, ako je transkripcija "Open web browser", DistilBERT će je klasificirati kao "OpenWebBrowser".

• U slučaju da transkripcija ne odgovara nijednoj od definiranih naredbi, kao što bi to bio slučaj s besmislenim nizom znakova, DistilBERT će tu transkripciju klasificirati kao "BadCommand".

Prednosti upotrebe DistilBERT-a:

• Sposobnost razumijevanja konteksta: DistilBERT je moćan model zasnovan na arhitekturi BERT, što mu omogućava da precizno razumije i klasificira tekstualne ulaze.

• Učinkovitost: Budući da je DistilBERT lakša verzija BERT-a, ima manji broj parametara, što omogućava brže izvođenje bez značajnog gubitka točnosti.

1.14.4. Integrirani 3-stage Model

Ovaj trostupanjski model osigurava visoku točnost i pouzdanost u prepoznavanju i klasifikaciji glasovnih naredbi. Svaki od tri koraka (YAMNet, Wav2Vec2.0, DistilBERT) ima specifičnu ulogu koja doprinosi ukupnoj funkcionalnosti sustava:

1. YAMNet osigurava da se model aktivira samo kada je detektiran ljudski govor, smanjujući tako nepotrebnu obradu.

2. Wav2Vec2.0 precizno prepoznaje što je izgovoreno u glasovnoj naredbi, koristeći svoju duboku arhitekturu za ekstrakciju značajki.

3. DistilBERT klasificira prepoznate naredbe u specifične akcije ili identificira besmislene ili nepoznate naredbe kao "BadCommand".

Kombinacijom ovih modela, uspio sam stvoriti robusni sustav koji može učinkovito prepoznati, obraditi i klasificirati glasovne naredbe, osiguravajući tako glatku i pouzdanu interakciju korisnika s Windows operativnim sustavom putem glasa.

1.15. Fine-tuning DistilBERT za Klasifikaciju Glasovnih Naredbi

Nakon uspješnog fine-tuninga Wav2Vec2.0 modela za prepoznavanje i transkripciju glasovnih naredbi, sljedeći korak u implementaciji sustava za glasovno upravljanje bio je fine-tuning DistilBERT modela. Ovaj model je korišten za klasifikaciju transkribiranih naredbi u jednu od četiri moguće klase: BadCommand, OpenWebBrowser, Search, i WinMenu. Ovdje ću opisati korake koje sam poduzeo za fine-tuning modela distilbert-base-uncased koristeći prilagođeni skup podataka.

1. Učitavanje i priprema podataka

Prvi korak u procesu bio je učitavanje skupa podataka. Podaci su dolazili iz prethodnih faza gdje su stvoreni sintetički skupovi podataka za svaku od naredbi koje model treba prepoznati. Ovi skupovi podataka su spojeni u jedan veliki dataframe koji se potom koristi za treniranje modela.

```
import pandas as pd
from datasets import Dataset
from transformers import AutoTokenizer, AutoModelForSequenceClassification, Trainer, TrainingArguments
from sklearn.preprocessing import LabelEncoder
from transformers import pipeline

labels = ["BadCommand", "OpenWebBrowser", "Search", "WinMenu"]
label_encoder = LabelEncoder()
df_browser = pd.read_csv("./browser_synthetic_dataset.csv")
df_search = pd.read_csv("./search_synthetic_dataset.csv")
df_winMenu = pd.read_csv("./winMenu_synthetic_dataset.csv")
```

```
#df columns #text, #label
dfs = [df_browser, df_search, df_winMenu]
merged_df = pd.concat(dfs, axis=0)
merged_df['label'] = label_encoder.fit_transform(merged_df['label'])
```

Detalji koraka:

• **Labeliranje podataka**: Svaka transkripcija je povezana s odgovarajućom klasom, koja je predstavljena numeričkom oznakom. Korištenjem LabelEncoder, oznake su transformirane u numeričke vrijednosti koje model može razumjeti.

• **Spajanje podataka**: Svi podaci iz različitih CSV datoteka spojeni su u jedan dataframe (merged_df). Ovaj dataframe sadrži tekst transkripcije i pripadajuću oznaku klase.

2. Inicijalizacija DistilBERT modela i tokenizatora

Nakon pripreme podataka, sljedeći korak je inicijalizacija modela i tokenizatora. Korišten je model distilbert-base-uncased, koji je unaprijed treniran na velikom korpusu tekstova, ali je sada trebao biti prilagođen specifičnoj zadaći klasifikacije glasovnih naredbi.

```
model_name = "distilbert-base-uncased"

tokenizer = AutoTokenizer.from_pretrained(model_name)

model=AutoModelForSequenceClassification.from_pretrained(model_name,n
um_labels=len(merged_df['label'].unique()))
```

Detalji koraka:

• **Tokenizator:** AutoTokenizer je korišten za pretvorbu tekstualnih podataka u format koji model može procesuirati. Tokenizator prilagođava tekst tako da odgovara maksimalnoj duljini sekvenci i koristi specifične kodove koji odgovaraju DistilBERT modelu.

• **Model:** AutoModelForSequenceClassification koristi se za klasifikaciju teksta. Ovaj model je unaprijed treniran, ali sada se prilagođava specifičnim potrebama kroz proces fine-tuninga.

3. Tokenizacija podataka

Kako bi podaci bili u odgovarajućem formatu za model, potrebno ih je tokenizirati. Ovaj korak uključuje pretvorbu tekstualnih podataka u numeričke vektore koje model koristi za učenje.

```
dataset = Dataset.from_pandas(merged_df)
def tokenize_function(examples):
```

```
return tokenizer(examples["text"], padding="max_length", truncation=True)
tokenized_datasets = dataset.map(tokenize_function, batched=True)
```

Detalji koraka:

• **Dataset**: Korištenjem Dataset klase iz datasets biblioteke, dataframe je pretvoren u format koji je kompatibilan s transformers bibliotekama.

• **Tokenizacija**: Funkcija tokenize_function prilagođava svaki tekstualni unos, dodajući padding i primjenjujući truncation kako bi svi zapisi bili iste duljine.

4. Definiranje trening argumenata i treniranje modela

Nakon pripreme podataka, definirao sam trening argumente koji upravljaju procesom treniranja, kao što su brzina učenja, veličina batcha, broj epoha, i drugi parametri.

```
training_args = TrainingArguments(
    output_dir="./results",
    evaluation_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=3,
    weight_decay=0.01,
)
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_datasets,
    eval_dataset=tokenized_datasets,
)
trainer.train()
```

Detalji koraka:

• **TrainingArguments**: Ova klasa specificira parametre treniranja, uključujući broj epoha (u ovom slučaju 3), strategiju evaluacije (evaluacija na kraju svake epohe), stopu učenja (learning_rate), i druge ključne parametre za optimizaciju modela.

• **Trainer**: Trainer klasa iz transformers biblioteke koristi se za obavljanje samog treniranja modela. Ova klasa omogućava jednostavno upravljanje treniranjem, evaluacijom i spremanjem modela.

Kroz proces fine-tuninga DistilBERT modela, postigao sam prilagodbu ovog unaprijed treniranog modela za specifičnu zadaću klasifikacije glasovnih naredbi. Koraci uključuju pripremu podataka, tokenizaciju, te treniranje modela na specifičnom skupu podataka. Rezultat je model koji je sposoban točno klasificirati tekstualne transkripcije iz govora u specifične radnje, omogućujući efikasno glasovno upravljanje aplikacijama ili operativnim sustavom.

1.16. Windows Integracija za Glasovno Upravljanje

Nakon uspješnog fine-tuninga DistilBERT modela za klasifikaciju glasovnih naredbi, sljedeći ključni korak u razvoju sustava za glasovno upravljanje bio je integracija modela s Windows operativnim sustavom. Ova integracija omogućuje da model, koji je treniran za prepoznavanje specifičnih glasovnih naredbi, direktno kontrolira funkcionalnosti unutar Windowsa putem govora. U nastavku ću opisati korake koji su poduzeti kako bi se postigla ova integracija, uključujući korištenje različitih softverskih komponenti i modula.

1.16.1. Uvod u Windows integraciju

Korištenje glasovnih naredbi za upravljanje Windowsom zahtijeva povezivanje prepoznatih naredbi s odgovarajućim radnjama unutar operativnog sustava. Da bi se to postiglo, kreirana je aplikacija koja koristi prethodno fine-tunane modele za prepoznavanje i klasifikaciju glasovnih naredbi, te integrira ove funkcionalnosti s PyAutoGUI, PyQt5, i webbrowser modulima za izvršavanje naredbi unutar Windows okruženja.

1.16.2. Inicijalizacija ključnih komponenti

Prvi korak u integraciji uključuje učitavanje i inicijalizaciju potrebnih modula i modela. Korišteni su moduli kao što su pyaudio za hvatanje zvuka u stvarnom vremenu, tensorflow i torch za manipulaciju modelima, librosa za obradu audio podataka, te PyQt5 za razvoj grafičkog sučelja aplikacije.

```
import pyaudio
import numpy as np
import torch
import tensorflow as tf
import tensorflow_hub as hub
```

```
import librosa
import pyautogui
import webbrowser
import torchaudio
import wave
import os
import matplotlib.pyplot as plt
import sys

from matplotlib.backends.backend_qt5agg import FigureCanvasQTAgg as FigureCanvas
from PyQt5.QtWidgets import QApplication, QMainWindow, QVBoxLayout, QWidget, QLabel, QHBoxLayout, QPushButton
from PyQt5.QtCore import Qt, QThread, pyqtSignal
from PyQt5.QtGui import QPalette, QColor

from sklearn.preprocessing import LabelEncoder
from datasets import Dataset as HFDataset
from torchaudio.transforms import Resample
from transformers import Wav2Vec2Processor, Wav2Vec2ForCTC, pipeline
```

Detalji komponenata:

• **PyAudio**: Korišten za snimanje zvuka u stvarnom vremenu, omogućavajući aplikaciji da kontinuirano sluša korisničke naredbe.

• **Librosa**: Koristi se za obradu audio signala, uključujući promjene brzine, dodavanje buke i druge augmentacije.

• **PyAutoGUI**: Omogućuje aplikaciji da simulira pritiske tipki i pokrete miša, što je ključno za kontrolu različitih funkcionalnosti unutar Windowsa.

1.16.3. Učitavanje i konfiguracija modela

Nakon inicijalizacije modula, sljedeći korak je učitavanje unaprijed treniranih modela za prepoznavanje govora i klasifikaciju glasovnih naredbi. U ovom slučaju, koristi se Wav2Vec2 model za prepoznavanje govora i DistilBERT za klasifikaciju naredbi.

```
label_encoder = LabelEncoder()
labels = ["BadCommand","OpenWebBrowser","Search","WinMenu"]

MODEL_DIRECTORY = 'results_wav2vec2ForCTC/checkpoint-7000'
```

```
PROC_DIRECTORY = 'results_wav2vec2ForCTC/30062024'
MODEL_NLP_DIRECTORY = "./results_AutoModel/checkpoint-29000"
MODEL_NAME = "distilbert-base-uncased"
FORMAT = pyaudio.paInt16  # 16-bit resolution
CHANNELS = 1  # Mono
RATE = 16000  # 16kHz sampling rate
CHUNK = 2150  # 20ms per frame

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model=hub.load('https://tfhub.dev/google/yamnet/1')
voiceCommandRecognizer=VoiceCommandRecognizer(MODEL_DIRECTORY,PROC_DI
RECTORY,device)
voiceCommandClassifier=VoiceCommandClassifier(MODEL_NLP_DIRECTORY,
MODEL_NAME)
```

Detalji učitavanja modela:

• **YAMNet model**: Ovaj model koristi se za prepoznavanje govora unutar audio zapisa, identificirajući kada je izgovorena ljudska naredba.

• **Wav2Vec2**: Koristi se za pretvaranje govora u tekstualne naredbe.

• **DistilBERT**: Nakon što se naredba transkribira, DistilBERT model koristi se za klasifikaciju transkripcije u specifične klase poput "OpenWebBrowser", "Search", "WinMenu" i "BadCommand".

1.16.4. Implementacija VoiceCommandRecognizer klase

Klasa VoiceCommandRecognizer upravlja prepoznavanjem i transkripcijom glasovnih naredbi. Nakon što se iz audio signala prepozna govor, model ga transkribira u tekst, koji se zatim klasificira.

```
class VoiceCommandRecognizer:
    def __init__(self, model_directory, proc_directory, device):
        self.processor =
        Wav2Vec2Processor.from_pretrained(proc_directory)
        self.model = Wav2Vec2ForCTC.from_pretrained(model_directory)
        self.device = device
        self.model.to(self.device)
    def transcribe_audio_v1(self, audio_data, sample_rate):
        audio_tensor =
        torch.tensor(audio_data,
      dtype=torch.float32).unsqueeze(0).to(self.device)
```

```
            inputs = self.processor(audio_tensor,
            sampling_rate=sample_rate, return_tensors="pt", padding=True)
            input_values = inputs.input_values.to(self.device)
            attention_mask=
          inputs.attention_mask.squeeze(0).to(self.device)
            with torch.inference_mode():
     logits=self.model(input_values,attention_mask=attention_mask).logits
                pred_ids = torch.argmax(logits, dim=-1)
                transcription = self.processor.batch_decode(pred_ids)[0]
            print(f"Prediction: {transcription.replace(' ', '')}\n")
            return transcription
```

Ključne funkcije:

• **transcribe_audio_v1**: Ova funkcija pretvara audio signal u tekst koristeći Wav2Vec2 model. Tekstualni rezultat se koristi za daljnju klasifikaciju i izvršavanje naredbi.

1.16.5. Klasifikacija naredbi i izvršavanje

Nakon što se naredba prepozna i transkribira, koristi se DistilBERT model za klasifikaciju naredbe i povezivanje s odgovarajućom funkcijom unutar Windowsa. Ovo omogućava aplikaciji da automatski otvara preglednik, pokreće pretrage ili otvara Windows izbornik na temelju glasovnih uputa korisnika.

```
class VoiceCommandClassifier:
    def __init__(self, model_dir, model_name):
        self.model = pipeline("text-classification", model=model_dir,
tokenizer=model_name)
        self.device = device
    def classify_command(self, command: str):
        result = self.model(command)
        print(result[0]['score'])
        numerical_label = int(result[0]['label'].split('_')[-1])
        label = labels[numerical_label]
        return label
```

Detalji:

• **Classify_command**: Ova funkcija prima tekstualnu naredbu i klasificira je u jednu od definiranih klasa. Na temelju klasifikacije, aplikacija izvršava odgovarajuću funkciju.

1.16.6. Funkcija classify_audio_v2

Funkcija classify_audio_v2 odgovorna je za cijeli proces prepoznavanja govora i klasifikacije glasovnih naredbi. Ova funkcija obavlja sljedeće ključne zadatke:

1. Pretvorba audio podataka u waveform:

○ Audio podaci dolaze u binarnom formatu (audio_data). Ovi podaci se pretvaraju u numpy array tipa int16 koji predstavlja waveform (zvučni val).

○ Normalizira se waveform dijeljenjem sa maksimalnom vrijednošću int16 kako bi se dobile vrijednosti između -1 i 1, što je uobičajena praksa za obradu audio signala.

```
waveform = np.frombuffer(audio_data, dtype=np.int16)
waveform = waveform / tf.int16.max
waveform = waveform.astype(np.float32)
```

2. Prepoznavanje ljudskog govora pomoću YAMNet modela:

• Model YAMNet, koji je unaprijed treniran za prepoznavanje različitih zvukova, koristi se za klasifikaciju zvuka. U ovom slučaju, tražimo klasu koja odgovara ljudskom govoru (Speech).

• Izračunavaju se scores (vjerojatnosti) za svaku klasu, a klasa s najvišom prosječnom vjerojatnošću određuje koja vrsta zvuka je u pitanju.

```
scores, embeddings, spectrogram = model(waveform)
scores_np = scores.numpy()
infered_class = class_names[scores_np.mean(axis=0).argmax()]
```

3. Transkripcija zvuka u tekst pomoću Wav2Vec2 modela:

• Ako YAMNet model detektira da je riječ o ljudskom govoru (Speech), waveform se konvertira u tensor kako bi bio kompatibilan s Wav2Vec2 modelom.

• Wav2Vec2 model zatim transkribira audio signal u tekstualnu naredbu.

```
if infered_class == 'Speech':
    audio_tensor = torch.from_numpy(waveform)
    audio_tensor.to(device)
    voiceCommandRecognizer.model.to(device)
    command =
    voiceCommandRecognizer.transcribe_audio_v1(audio_tensor,
RATE)
    command = command.replace(" ", "")
```

4. Klasifikacija transkribirane naredbe:

• Nakon transkripcije, klasificira se naredba pomoću DistilBERT modela koji je prethodno fine-tunan.

• Rezultat klasifikacije koristi se za određivanje konkretne naredbe (npr. "OpenWebBrowser", "Search", "WinMenu").

```
command =voiceCommandClassifier.classify_command(command)
print(f"Classified command : {classified_command}")
```

5. Spremanje audio zapisa:

• Ako je sve prošlo uspješno, audio zapis se sprema na disk za buduću upotrebu ili analizu.

```
save_audio(audio_data)
```

1.16.7. Funkcija callback_v2

Funkcija callback_v2 je callback funkcija koja se koristi u PyAudio streamu kako bi obradila dolazne audio podatke u stvarnom vremenu. Ova funkcija se poziva svaki put kada PyAudio primi određeni broj okvira (frame-ova) audio podataka.

1. Spremanje dolaznih audio podataka:

○ Funkcija callback_v2 prima audio podatke (in_data) i pohranjuje ih u audio_buffer. Ovaj buffer se koristi za akumulaciju podataka dok se ne sakupi dovoljna količina za analizu.

```
global audio_buffer, start_stream
if start_stream:
    audio_buffer.append(in_data)
```

2. Provjera duljine audio podataka:

○ Kada akumulirani podaci premaše određeni prag (u ovom slučaju 35000 CHUNK-ova), buffer se obrađuje kao jedan cjeloviti audio zapis.

○ Zatim se zaustavlja stream (start_stream = False) kako bi se obradili podaci, a nakon toga ponovno pokreće stream.

```
if len(audio_buffer) * CHUNK >= 35000:
    start_stream = False
    audio_data = b''.join(audio_buffer)
```

```
        audio_buffer = []
        classify_audio_v2(audio_data)
        start_stream = True
```

3. Poziv funkcije classify_audio_v2:

```
    classify_audio_v2(audio_data)
```

4. Povrat vrijednosti za nastavak stream-a:

Funkcija vraća tuple s originalnim in_data i statusom koji omogućuje PyAudio streamu da nastavi s radom.

```
    return (in_data, pyaudio.paContinue)
```

Kombinacijom funkcija classify_audio_v2 i callback_v2, sustav za glasovno upravljanje može kontinuirano snimati, prepoznati i klasificirati glasovne naredbe u stvarnom vremenu. callback_v2 upravlja prikupljanjem i inicijalnom obradom podataka, dok classify_audio_v2 obavlja složeniju analizu, transkripciju i klasifikaciju, što omogućuje korisnicima efikasno upravljanje Windows operativnim sustavom putem govora.

1.16.8. Izvršavanje Windows naredbi

Na temelju klasifikacije, aplikacija izvršava odgovarajuće akcije unutar Windows okruženja. Ovo uključuje otvaranje web preglednika, simuliranje unosa u Windows izbornik ili prepoznavanje nevažećih naredbi.

```
class AudioStreamWorker(QThread):
    # ...
    def execute_command(self, label):
        command_mappings = {
            "OpenWebBrowser": self.command_browser,
            "Search": self.command_search,
            "WinMenu": self.command_winMenu,
            "BadCommand": self.command_badCommand,
        }
        command_function = command_mappings.get(label)
        if command_function:
            command_function()
    def command_browser(self):
        webbrowser.open('http://www.google.com')
```

```
def command_winMenu(self):
    pyautogui.press('win')
def command_badCommand(self):
    print("Not valid command")
def command_search(self):
    pyautogui.press('win')
    pyautogui.write("search query")
```

Ključne funkcije:

- **command_browser**: Otvara zadani web preglednik.
- **command_search**: Aktivira Windows pretragu.
- **command_winMenu**: Otvara Windows izbornik.
- **command_badCommand**: Obrađuje nevažeće naredbe.

1.16.9. Pokretanje aplikacije i grafičko sučelje

Aplikacija koristi PyQt5 za grafičko sučelje, omogućavajući korisniku da vizualno prati proces snimanja i prepoznavanja glasovnih naredbi. Aplikacija također nudi jednostavan način za pokretanje i zaustavljanje audio streama, kao i praćenje vizualizacije audio signala.

```
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.setWindowTitle("AI-Voice")
        self.setWindowOpacity(0.8)
        self.audio_stream_widget = AudioStreamWidget(self)
        self.setCentralWidget(self.audio_stream_widget)
        layout = QVBoxLayout(self.audio_stream_widget)
        layout.setContentsMargins(0, 0, 0, 0)
        layout.setSpacing(0)
        self.audio_worker = AudioStreamWorker()
self.audio_worker.update_waveform.connect(self.audio_stream_widget.update_p
lot)
        self.audio_worker.start()
    def closeEvent(self, event):
        self.audio_worker.stop()
        event.accept()
def main():
    app = QApplication(sys.argv)
```

```
    main_window = MainWindow()
    main_window.show()
    try:
        sys.exit(app.exec_())
    except KeyboardInterrupt:
        pass
    finally:
        if hasattr(main_window, 'audio_worker'):
            main_window.audio_worker.stop()
if __name__ == "__main__":
    main()
```

Ključni elementi:

• **MainWindow**: Glavni prozor aplikacije koji upravlja prikazom i kontrolom audio streama.

• **AudioStreamWidget**: Widget koji prikazuje vizualizaciju audio signala u stvarnom vremenu.

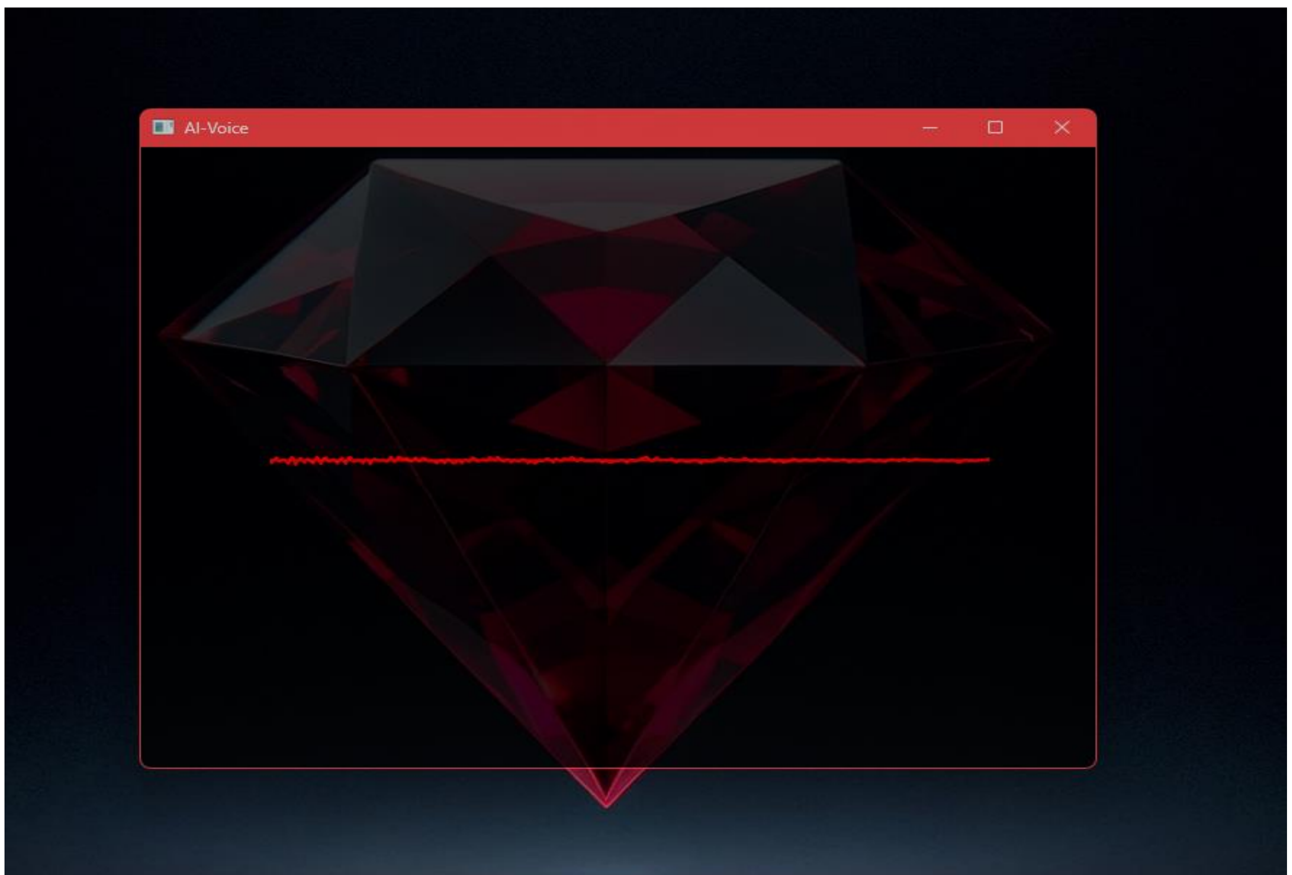
Integracija modela za prepoznavanje i klasifikaciju glasovnih naredbi u Windows okruženje omogućila je razvoj aplikacije koja omogućuje korisnicima upravljanje sustavom putem glasovnih naredbi. Korištenje unaprijed treniranih modela poput Wav2Vec2 i DistilBERT omogućilo je precizno prepoznavanje i klasifikaciju naredbi, dok su alati poput PyAudio i PyAutoGUI osigurali interakciju s operativnim sustavom na način koji je intuitivan i učinkovit.

Konačni rezultati

1.17. Prikaz rezultata

Slika prikazuje sučelje Python Windows aplikacije nazvane "AI-Voice". Aplikacija sadrži grafički prikaz waveform-a (valnog oblika zvuka) u sredini prozora. Ovaj waveform služi kao vizualni prikaz zvuka koji aplikacija registrira u stvarnom vremenu. Valni oblik je prikazan kao tanka crvena linija koja oscilira duž sredine prozora, što ilustrira trenutnu razinu zvuka ili glasovne naredbe unutar aplikacije.

Waveform nije statičan on se animira u skladu s ulaznim zvukom, pružajući korisniku vizualnu povratnu informaciju o tome da aplikacija aktivno sluša i obrađuje zvučne signale. Sveukupno, sučelje je jednostavno, ali efektno, kombinirajući funkcionalnost i estetski privlačan dizajn.



Slika 1 Windows aplikacija "AI-Voice"

Slika prikazuje rezultat Python aplikacije u terminalu nakon što je korisnik izgovorio glasovnu naredbu "Open Web Browser" u mikrofon.

Aplikacija pomoću Wav2Vec2 modela pretvara audio signal u tekstualni oblik. Nakon transkripcije, prikazana je predikcija modela: "penwebbrowser", što je blizu izgovorenoj naredbi, ali s malom greškom u transkripciji zbog izazova u točnosti prepoznavanja. Ispod transkripcije prikazana je vjerojatnost koju model dodjeljuje ovoj predikciji: 0.9999997615814209, što ukazuje na visoki stupanj točnosti u rezultat. Na kraju, aplikacija koristi DistilBERT model kako bi klasificirala prepoznatu naredbu i mapirala je na pravu akciju. Klasificirana naredba prikazana je kao: "OpenWebBrowser", što je ispravna komanda. Ova komanda bi sada pokrenula odgovarajuću akciju, kao što je otvaranje web preglednika u Windows operativnom sustavu.

```
C:\Users\winte\AppData\Local\Temp\ipykernel_9800\1594319731.py:71: UserWarning: To copy construct from
  audio_tensor = torch.tensor(audio_data, dtype=torch.float32).unsqueeze(0).to(self.device)
c:\Users\winte\AppData\Local\Programs\Python\Python312\Lib\site-packages\transformers\models\wav2vec2\
  attn_output = torch.nn.functional.scaled_dot_product_attention(
Prediction: penwebbrowser

0.9999997615814209
Classified command : OpenWebBrowser
```

Slika 2 Rezultat izgovora glasovne komande Open Web Browser

Slika prikazuje izlaz Python aplikacije u terminalu nakon što je korisnik izgovorio neprepoznatljivu glasovnu naredbu, koja se ne može mapirati na jednu od predefiniranih akcija. U izlazu možemo vidjeti da je model Wav2Vec2 pokušao transkribirati audio unos i rezultat transkripcije je prikazan kao "ou". Ova transkripcija očito ne odgovara nijednoj od očekivanih naredbi.

Sljedeći korak je klasifikacija te transkripcije pomoću DistilBERT modela. Model je klasificirao ovu naredbu kao "BadCommand", što znači da je prepoznao da se radi o nevažećoj naredbi. Vjerojatnost dodijeljena ovoj klasifikaciji iznosi 0.956025242805481, što ukazuje na relativno visoko točnost modela u ovu klasifikaciju. Na kraju, aplikacija generira izlaz "Not valid command" kako bi korisnika obavijestila da izgovorena naredba nije valjana i da se neće izvršiti nijedna akcija u sustavu. Ova funkcionalnost je ključna za rukovanje neprepoznatljivim unosima, čime se osigurava stabilnost aplikacije prilikom interakcije s korisnikom.

```
start_stream = True
if __name__ == "__main__":
    main()
```

[20] 2m 12.3s Python

```
Prediction: ou

0.956025242805481
Classified command : BadCommand
Not valid command
```

Slika 3 Rezultat izgovora nevaljane glasovne komande

```
start_stream = True
if __name__ == "__main__":
    main()
```

[18] Python

```
C:\Users\winte\AppData\Local\Temp\ipykernel_1412\1594319731.py:71: UserWarning: To copy construct from
  audio_tensor = torch.tensor(audio_data, dtype=torch.float32).unsqueeze(0).to(self.device)
c:\Users\winte\AppData\Local\Programs\Python\Python312\Lib\site-packages\transformers\models\wav2vec2\
  attn_output = torch.nn.functional.scaled_dot_product_attention(
Prediction: tsurfingonline

0.9999998807907104
Classified command : OpenWebBrowser
C:\Users\winte\AppData\Local\Temp\ipykernel_1412\1594319731.py:71: UserWarning: To copy construct from
  audio_tensor = torch.tensor(audio_data, dtype=torch.float32).unsqueeze(0).to(self.device)
Prediction: search

0.9999779462814331
Classified command : Search
Prediction: indowsmenu

0.9999998807907104
Classified command : WinMenu
```

Slika 4 Rezultat izgovora glasovnih naredbi u nizu

Zaključak

Tijekom ovog projekta razvijena je aplikacija za glasovno upravljanje Windows operativnim sustavom, koja koristi najnovije tehnologije strojnog učenja učenja za prepoznavanje i klasifikaciju glasovnih naredbi. Projekt je obuhvatio sve ključne korake potrebne za izgradnju pouzdanog sustava, od prikupljanja podataka i preprocesiranja, preko razvoja i fine-tuninga modela, do konačne integracije s Windows okruženjem.

1. Skupljanje i priprema podataka:
Prvi izazov bio je prikupljanje kvalitetnih podataka za treniranje modela. Korištenje raznolike i bogate baze podataka s različitim verzijama istih naredbi, uključujući sintetičke greške, omogućilo je izgradnju modela otpornog na varijacije u izgovoru i manje greške u transkripciji.

2. Razvoj i testiranje vlastitih modela:
Početni pokušaji izgradnje vlastitog RNN modela pokazali su da je klasična arhitektura, iako sposobna za prepoznavanje obrazaca u sekvencijalnim podacima, imala poteškoća s generalizacijom u stvarnim slučajevima. Ova spoznaja dovela je do potrebe za istraživanjem drugih pristupa.

3. Fine-tuning postojećih modela:
Korištenje unaprijed istreniranih modela, kao što su Wav2Vec2 i DistilBERT, pokazalo se kao iznimno učinkovito rješenje. Fine-tuning ovih modela na specifične skupove podataka omogućio je visoku točnost u prepoznavanju i klasifikaciji glasovnih naredbi, čak i u situacijama gdje su unosi bili blago netočni ili nejasni.

4. Integracija s Windows operativnim sustavom:
Konačna faza projekta bila je integracija s Windows okruženjem, gdje su prepoznate naredbe korištene za pokretanje konkretnih akcija poput otvaranja web preglednika ili pretraživanja u Windows izborniku. Implementacija ove funkcionalnosti omogućila je korisnicima da učinkovito upravljaju računalom putem glasa, čineći cijeli sustav intuitivnim i korisnički orijentiranim.

5. Izazovi i rješenja:
Tijekom razvoja sustava, suočio sam se s nekoliko izazova, uključujući problem generalizacije i preciznosti prepoznavanja. Prvi neuspjeli pokušaji s vlastitim modelom, kao i problemi s prekomjernim sažimanjem podataka na samo tri klase, pokazali su važnost pravilnog označavanja i korištenja naprednih modela za fine-tuning. Ova iskustva su istaknula potrebu

za pažljivim dizajnom i strukturiranjem modela te pripremom podataka kako bi se postigla visoka razina točnosti i pouzdanosti u stvarnim aplikacijama.

6. Budući rad:

Projekt predstavlja solidnu osnovu za daljnji razvoj i nadogradnju sustava. Potencijalne nadogradnje uključuju proširenje baze podataka za treniranje, optimizaciju modela za još veću točnost, te integraciju dodatnih funkcionalnosti koje bi omogućile još veću fleksibilnost i primjenjivost aplikacije u različitim kontekstima.

Ovaj rad ne samo da je omogućio praktičnu primjenu naprednih tehnika dubokog učenja u stvarnom okruženju, već je također pokazao važnost kombinacije teorijskog znanja, eksperimentiranja i prilagodbe kako bi se postigli najbolji rezultati. Aplikacija razvijena kroz ovaj projekt ima potencijal da unaprijedi način na koji korisnici komuniciraju sa svojim računalima, omogućujući prirodniju, efikasniju i pristupačniju interakciju.

Popis slika

Slika 1 Windows aplikacija "AI-Voice" 53
Slika 2 Rezultat izgovora glasovne komande Open Web Browser 54
Slika 3 Rezultat izgovora nevaljane glasovne komande 55
Slika 4 Rezultat izgovora glasovnih naredbi u nizu 55

Popis literature

1. Hugging Face. (n.d.). *DistilBERT Model Documentation*. Hugging Face. Preuzeto sa https://huggingface.co/docs/transformers/en/model_doc/distilbert
2. Hugging Face. (n.d.). *Wav2Vec2 Model Documentation*. Hugging Face. Preuzeto sa https://huggingface.co/docs/transformers/en/model_doc/wav2vec2
3. PyTorch. (n.d.). *PyTorch Documentation*. Preuzeto sa https://pytorch.org/docs/stable/index.html
4. TensorFlow Hub. (n.d.). *YAMNet Tutorial*. TensorFlow. Preuzeto sa https://www.tensorflow.org/hub/tutorials/yamnet
5. Ayo, F. (2020, July 7). *Audio Deep Learning Made Simple: Sound Classification Step-by-Step*. Towards Data Science. Preuzeto sa https://towardsdatascience.com/audio-deep-learning-made-simple-sound-classification-step-by-step-cebc936bbe5
6. IBM. (n.d.). Recurrent Neural Networks (RNNs). IBM. Preuzeto sa https://www.ibm.com/topics/recurrent-neural-networks
7. Amidi, S., & Amidi, A. (n.d.). CS 230 - Recurrent Neural Networks Cheatsheet. Stanford University. Preuzeto sa https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks
8. TensorFlow. (n.d.). Working with RNNs. TensorFlow. Preuzeto sa https://www.tensorflow.org/guide/keras/working_with_rnns
9. Hugging Face. (n.d.). Training. Hugging Face. Preuzeto sa https://huggingface.co/docs/transformers/en/training
10. DataCamp. (n.d.). Fine-Tuning Large Language Models. DataCamp. Preuzeto sa https://www.datacamp.com/tutorial/fine-tuning-large-language-models