Improving daily interactions of visually impaired people: towards an engaging vocal interface

Speech Processing and Recognition (101803) - University of Genoa, Spring 2020

Federico Minutoli[1]
4211286
fede97.minutoli@gmail.com

Matteo Ghirardelli[1]
4147398
matteoghirardelli01@gmail.com

Sofia Bagnato[2]
4320999
sofia.bagnato1998@gmail.com

Gianvito Losapio[2]
4803867
gvlosapio@gmail.com

***Abstract*—Text-to-speech synthesis (TTS) is the process of transforming texts written in natural language into synthetic speech. The use cases of the TTS technology are many, and they include, for instance, providing accurate speech descriptions of a scene to visually impaired people, or giving users with speech disabilities a way to communicate easily. Automatic speech recognition, which is also called speech-to-text (STT), is instead the opposite process which performs the recognition and translation of speech signals into texts written in natural language. This technology can be used for several purposes, one being the possibility of making visually impaired people able to use vocal commands in order to interact with scene description generators or with voice-to-text messaging applications. In this report, we present the Android implementation of a text-to-speech module and a speech-to-text module as the first steps towards the creation of a more ambitious application. In particular, after an introduction on the general frameworks of TTS and STT and on various techniques which have been adopted by such systems in the literature, we provide an overview of the state-of-the-art TTS and STT models used nowadays in the market. We then describe the architecture of our application, the Android APIs used in our project, and we list the main implementation details of the TTS and STT modules, along with the values chosen for the existing parameters. We then show how the application has been tested, providing screenshots for each functionality. In the end, we discuss a possible integration with a computer vision module in charge of producing high level image descriptions.**

***Index Terms*—speech processing, text-to-speech, speech recognition, Android, healthcare, visual-impairment**

I. Introduction

We proposed a joint project with the ambition of investigating easily accessible, state-of-the-art technologies in two different domains - Computer Vision and Speech Processing/Recognition - in order to conduct a feasibility study for the following use case:

> *"Providing a virtual assistant to visually-impaired people which can help making their daily interactions more engaging"*

More specifically, our main objective is the deployment of specific high-level cognitive functions on Android devices with low computational power.

[1] Project number 10: Text-to-speech app
[2] Project number 6: Speech recognition app

Applications targeted for visually impaired people are unfortunately lacking in quantity, which causes some people to depend on external help for day-to-day tasks. As for the current options on the market, the most relevant products are:

- *Envision glasses*, the new AI-powered smartglasses by Envision company, empowering the blind and visually impaired to be more independent, shown in Fig. 1 [1];
- *OrCam MyEye*, a tiny device with a smart camera that can be mounted on virtually any eyeglass frame which conveys the visual information audibly, in real-time and offline [2].

Fig. 1: Envision glasses

This report explains our initial focus on the vocal technologies - namely text-to-speech (TTS) and automatic speech recognition (ASR) - which are crucial for the success of a more ambitious application. TTS is a fundamental functionality allowing a visually impaired person to both catch useful insights about the world and consume digital material which would be, otherwise, unaccessible. On the other side, speech recognition is of paramount importance to control the application flow in a hands-free mode.

The report is organized as follows. Section II is devoted to text-to-speech systems and will comprise an overview of the problem, a short presentation of the state-of-the-art technologies as well as an introduction to the android APIs we chose to use. This structure is replicated in the section III on speech recognition systems. Section IV is dedicated to an in-depth analysis of our android application, divided in 4 subsections: (IV-A) with a focus on the general application workflow, providing a thorough description from the user point of view, both on how to navigate the app and on the single, implemented functionalities; (IV-B), (IV-C) containing an explanation of how the APIs described in the previous section were used; (IV-D) containing a description about our testing process. Section V contains conclusions and future works.

II. Text-to-speech systems

A. Overview

The process of transforming text written in natural language into synthetic speech, known as text-to-speech synthesis (TTS), should ideally yield human-like speech sounding as natural and intelligible as possible. Such a process is an essential component of applications in many different fields, such as speech-enabled devices, navigation systems and accessibility for visually-impaired people. A typical TTS system consists of two complex multi-stage processing pipelines: the natural language processing (NLP), or front-end, component, and the digital signal processing (DSP), or back-end, component. Intuitively, the NLP component represents the entry point to the system and is in charge of processing raw input text to extract useful prosodic parameters, such as duration, pitch and energy of phonemes, which express rhythm, intonation and loudness of speech, respectively. In order to do so, the NLP component has to go through three contiguous steps: Basic text processing, phonetic analysis and prosodic analysis. The DSP component, instead, converts the phonemes estimated from analyzed text, along with the prosodic parameters supplied by the NLP's, into an artificial sound. Historically, the latter has seen the deepest changes with variants belonging to three different families: Rule-based, concatenative or parametric synthesis.

Rule-based synthesis focuses sound generation on a complete (w.r.t. to the vocabulary $\mathcal{V}$, at hand) set of empirical rules for each class of phonemes (e. g., voiceless fricatives, sonorants, etc...). This technique was the first one being addressed as it doesn't need many resources and, as such, it was useful back in the day on limited-power hardware, like embedded systems. Unfortunately, requiring prior knowledge on the exact grammar underlying $\mathcal{V}$ doesn't scale well to large vocabularies (i. e., the English language) - moreso if they are in constant evolution.

Concatenative synthesis focuses sound generation on a large database of pre-recorded human audio signals from which new audible speech is produced by looking at the combination of diphones (i. e., half phones) out of all the available sequences, each of which is stored in many different facets. Sometimes longer sequences (units) of entire words are taken into account, instead. In order to choose the most appropriate unit to fill the sequence with in this scenario, a best-path search via Viterbi algorithm is performed to minimize the selection cost of adjacent units. Eventually, overlap-and-add (OLA) is used to concatenate previously selected units into a single output audio signal. The diphone's variant is super light-weight (a database of only $O(N^2)$ size is required, with N the number of diphones), but the quality may result in being robotic under stress. The units' variant, instead, achieves very good quality at the expense of a very large database, which limits its scalability to bigger vocabularies, as to change style of speech an entire new database with those audio voices would be needed.

Parametric synthesis focuses sound generation on a set of parameters that can be modified to change the voice as a function of prosodic and acoustic parameters. This technique makes use of powerful statistical models to handle such a processing, mainly hidden Markov models (HMMs), which operate in a similar fashion to what they do for acoustic modeling, albeit returning the HMM model itself that captured a given word in $\mathcal{V}$, as a probabilistic generator of future sounds.

Articulatory synthesis is a worth mentioning novel branch of TTS systems that tries to emulate mechanical and acoustic models of the vocal tract. It has shown promising experimental results thus far, but it's still way too costly.

As we have seen, TTS systems rely on hard-engineered features and heuristics: Due to this complexity developing new TTS systems' components can be very labor intensive and often requires extensive domain expertise with brittle design choices. Deep learning has consequently made its way through the complex pipeline of HMM-based parametric models, as deep neural networks (DNNs) have been introduced in place of them with a least- to a most-intensive manner, which has given birth to modern TTS systems that collapse NLP-DSP separation into a cohesive unified pipeline. Here on we will briefly comment on a handful of them from leading Tech-giants:

- WaveNet, by DeepMind, a (causal) convolutional neural network (CNN) for generating raw audio waves. It is a fully probabilistic and autoregressive model able to generate state-of-the-art results for English, where each audio sample is conditioned on the previous audio sample. Causal convolutional layers fulfill this purpose by preserving the ordering of the data, while also being faster to train than RNNs as they don't have recurrent connections, albeit posing a daunting computational problem due to the high-frequency autoregressive nature of WaveNet's model which hinders its capabilities of on premises real-time processing [5].

- Deep Voice, by SVA Baidu Lab, an ensemble of five building blocks of DNNs that specifically addresses real-time TTS' artificial generation. The choice of features - phonemes with prosodic parameters and stress annotation - is its main selling point, as it makes Deep Voice's system more readily applicable to new datasets, voices, and domains without any manual data annotation or additional feature engineering. More recently, two newer versions have been published (Deep Voice 2 and 3) that introduce multi-speaker TTS from a single model and attention-based character-to-spectrogram convolutional TTS, respectively [6].

- Tacotron, by Google, an end-to-end generative TTS model that synthesizes speech directly from text and audio pairs at frame-level and is, therefore, faster than sample-level autoregressive methods. Like Deep Voice, it is an attention-based sequence-to-sequence CNN model (robust to transfer learning), albeit it generates a raw Mel-scale spectrogram which is only then converted to waveforms. It has been the main motor behind Google Translate service ever since its release, as it has achieved a 3.82 mean opinion score (MOS) on US English. More recently, a second version was announced, Tacotron 2, which, with the addition of recurrent connections, seeks to take the best of the two worlds from the original WaveNet and Tacotron architectures - a stunning MOS of 4.53 is achieved [4].

B. *Android text-to-speech APIs*

In this section, we will provide a general overview of how a standard android TTS application can be built in java, and which APIs can be used to do so. Moreover, we will also show what are the most important parameters that should be taken into account when building a TTS application, w.r.t. the specific APIs being used.

For the purpose of implementing an Android text-to-speech application, we used the `TextToSpeech` java class [7]. This class allows us to use any already available implementation of a text-to-speech engine, provided that such engine is installed on the Android device. With the `TextToSpeech` class, we are able to specify directly which engine we will use by instantiating a `TextToSpeech` object. Before doing so, a standard Android TTS application usually checks that the Android system has the necessary resources installed in order to use the TTS engine effectively. Such resources are, specifically, voice data associated with specific languages: such data have in fact the format ("Voice type number 1", Locale). The process of checking such resources is usually executed by using the `Intent` class in Android[1], which is used in combination with the `startActivityForResult` method of the class `AppCompatActivity`. In short, the Intent describes the features of an action to be performed, whereas the `startActivityForResult` method allows us to trigger the callback `OnActivityResult` after the completion of the action that checks the resources. Such callback is used to execute further actions if the Intent fails or succeeds. In general, if the system doesn't have any TTS resources installed, then the usual practice is to redirect the user to an appropriate system application which has the purpose of choosing and installing such data; in the other case, instead, we start to create the `TextToSpeech` object. In order to create this object, we have also to specify an implementation of the `OnInit` method of an `OnInitListener` object. This listener calls the `OnInit` callback as soon as the TTS engine initialization has completed. The TTS initialization for the effective usage of speech synthesis from text is in fact triggered by the call to the constructor of the TextToSpeech class. In the `OnInit` method, we usually specify the configurations for the main language that will be adopted by the TTS engine. For instance, we could set the main language either to UK English or US English, if the corresponding voice data is available. This is not always the case, because the resources checked successfully when we use the Intent class may be different from the ones we set programmatically as languages in our application. Therefore, if they are not available, a common choice is to set the main language of the system to the one associated with the user's current locale. The whole process of setting the main languages for the system is done by using a Voice object. We can also modify the speech rate by using the method `setSpeechRate` of the `TextToSpeech` class, and the pitch by using the `setPitch` method from the same class. In order to use the `TextToSpeech` object configured so far, we simply have to call the method `speak` of the `TextToSpeech` class, which produces an audio signal given an input text. We can also modify programmatically the parameters of that particular spoken text. The most useful ones are, in this case, the following:

- `TextToSpeech.Engine.KEY_PARAM_PAN`: float between -1 and 1. Determines how much the audio is shifted to one of the two human ears. This can be noticed more effectively by using headphones.
- `TextToSpeech.Engine.KEY_PARAM_VOLUME`: float between 0 and 1.
- `TextToSpeech.Engine.KEY_PARAM_STREAM`: this parameter determines the expected general shape of the audio stream of the signal.

The provided values for such parameters are passed to the speak method by using a `Bundle` object, which is basically an Hashmap with (Key, Value) mapping. Another useful object to be used with the `TextToSpeech` class is the `UtteranceProgressListener`. This listener can be used in order to specify custom behaviors to be adopted when the speaking operation is in progress. An example of callback to be implemented in this listener is `onRangeStart`, which can be used to perform some actions each time a single word is read out loud, for instance.

More detailed information about how to set the most appropriate values for the parameters described so far will be provided in the section IV-A.

III. Speech Recognition systems

A. *Overview*

In the last few years, the ability to talk with different kind of devices (e.g. smartphones, cars, home assistants) is moving from the realm of science fiction into daily life. This is possible thanks to the recent advances in long vocabulary continuous

[1]In Android, an *Intent* is basically a passive data structure holding an abstract description of an action to be performed. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities. See https://developer.android.com/reference/android/content/Intent.

speech recognition (LVCSR) - i.e. the (real-time) translation of spoken words coming from a large vocabulary into text.

More precisely, the goal of automatic speech recognition (ASR) is to map an acoustic input sequence $X = \{x_1, \dots, x_T\}$ to a label sequence $L = \{l_1, \dots, l_N\}$, where:

- $x_t \in X$ is a d-dimensional feature vector extracted from the raw audio signal (such as the Mel Filter Bank or the auditory spectrogram) extracted from the t-th speech frame;
- $l_u \in L$ is an element of a given vocabulary $\mathcal{V}$, generally composed of the most common words in a language. As a consequence, L can be considered as an element of the set $\mathcal{V}^*$, i.e. the collection of all label sequences formed by labels in $\mathcal{V}$.

The ideal goal is thereby to find the most likely label sequence

$$\hat{L} = \underset{L \in \mathcal{V}^*}{\arg\max}\ p(L|X) \tag{1}$$

by establishing a model that can accurately compute the posterior distribution $p(L|X)$.

For a long time, Hidden Markov chains combined with mixture of Gaussians (HMM-GMM) have been the mainstream LVCSR framework [8], [9]. According to HMM-GMM, the (spoken) words in a sentence (or phonemes in a word) are estimated as the most likely sequence of hidden states $S = \{s_1, \dots, s_T\}$ with $s_t \in \{1, \dots, K\}, t = 1, \dots, T$ underlying the feature vector X. Generally, the following approximation is considered (assuming conditional independence):

$$\underset{L \in \mathcal{V}^*}{\arg\max}\ p(L|X) \simeq \underset{L \in \mathcal{V}^*}{\arg\max} \sum_S p(X|S)\, p(S|L)\, p(L) \tag{2}$$

where:

- $p(L)$ is called *language model* and represents acceptable (spoken) sentences in a specific language, represented by probability distributions over sequences of words. N-gram models learned on a large corpus along with grammar rules are typically used here.
- $p(S|L)$ is called *pronunciation model* or *dictionary*. Its role is to achieve the connection between acoustic sequence and phonetic spelling of each word through arbitrary levels of mapping, such as pronunciation to phone, phone to triphone.
- $p(X|S)$ is called *acoustic model* and maps the feature vector to distinct hidden states (e.g. phones). Generally, mixture of Gaussians are employed to compute this probability, giving the name to the model.

It is worth noting that each model plays an independent role and uses specific algorithms, for instance Baum-Welch to learn parameters of HMM models, forward-backwards to evaluate probabilities and Viterbi to solve the optimization problem (2), also referred as decoding phase [10].

Recently, the majority of industrially deployed systems are based on HMM in combination with deep learning techniques [8]. Typically, Long Short Term Memory (LSTM) networks and Convolutional Neural Networks (CNN) are trained on huge speech dataset such as AudioSet [11] to create an accurate acoustic model: the networks predict the phones corresponding to each frame and then the Viterbi decoding algorithm is used to compute the most likely phone sequence [9], [12].

Due to several advantages, acoustic and language models based on deep networks as well as end-to-end deep learning models are currently a very important research direction of speech recognition [8], [13], [14]. In particular, transfer and continuous learning on mobile devices with resource-efficient models is one of the main research interests of Google with the aim of improving speech representations and providing a personalized experience on Android devices [15], [16]. This is in accordance to the major technological trend in machine learning of rapidly moving closer to where data are collected - edge devices. A pictorial representation is provided in fig.2.

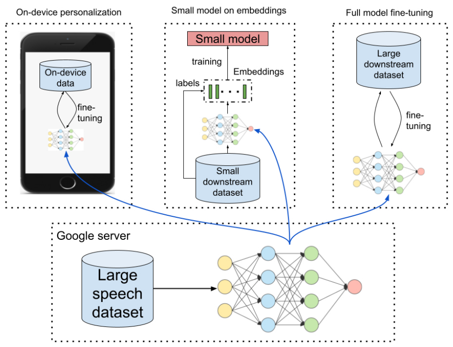


Fig. 2: Bottom:A large speech dataset is used to train a model, which is then rolled out to other environments. Top Left: On-device personalization — personalized, on-device models combine security and privacy. Top Middle: Small model on embeddings — general-use representations transform high-dimensional, few-example datasets to a lower dimension without sacrificing accuracy; smaller models train faster and are regularized. Top Right: Full model fine-tuning — large datasets can use the embedding model as pre-training to improve performance. Credits: Google research blog [15].

B. *Android speech recognition APIs*

We present the two main Android speech recognition APIs chosen for this project. The reasons behind their choices are detailed in the section IV-C.

Speech Recognizer. Android OS provides a proprietary high-level API for automatic speech recognition through the package `android.speech` [17]. The three main elements to consider when implementing a basic speech recognition system are:

- the class `SpeechRecognizer` which provides the main entry point to the speech recognition service;
- the interface `RecognitionListener`, used for receiving notifications when the recognition related events occur and consequently handling callbacks;

- the class RecognizerIntent, used to specify constants and parameters for supporting or tuning the speech recognition process.

A list of standard steps required to a developer follows:

1) Implementation of the RecognitionListener interface through the specification of its methods (not necessarily all of them) according to the developer's needs. Of main interest are:
   - onBeginningOfSpeech: called when the user has started to speak;
   - onEndOfSpeech: called after the user stops speaking;
   - onError: called when a network or recognition error occurred;
   - onPartialResults: called when partial recognition results are available as a Bundle of data;
   - onResults: called when recognition results are ready as a Bundle of data.

   In order to properly handle such callbacks, the following attributes of the SpeechRecognizer class can be used as keys to retrieve useful objects from the Bundle passed to the onResults and onPartialResults methods:
   - RESULTS_RECOGNITION: used to retrieve an ArrayList of strings containing the results of the speech recognition;
   - CONFIDENCE_SCORES: used to retrieve a float array containing confidence scores in a range 0-1 associated to each of the predicted results.
2) Initialization of a SpeechRecognizer object in the main application thread (e.g. in the onCreate method of an Activity) through the method createSpeechRecognizer passing the Context object in which the speech recognition service will be used (e.g. the Activity being implemented);
3) Association of the listener to the SpeechRecognizer object with the method setRecognitionListener;
4) Specification of parameters for the speech recognition process: an Intent object is instantiated on the action ACTION_RECOGNIZE_SPEECH specified in the RecognizerIntent class and related key-value pairs of strings are then passed to it through the method putExtra. For instance:
   - EXTRA_LANGUAGE: optional IETF language tag (as defined by BCP 47) to recognize speech in a language different from the default (e.g. "en-US");
   - EXTRA_LANGUAGE_MODEL: informs the recognizer which speech model to prefer in order to fine tune the results: LANGUAGE_MODEL_FREE_FORM, for a language model based on free-form speech recognition or LANGUAGE_MODEL_WEB_SEARCH for a language model based on web search terms. This extra is required.
   - EXTRA_MAX_RESULTS: optional limit on the maximum number of results to return, specified as an integer.
   - EXTRA_PARTIAL_RESULTS: optional boolean to indicate whether partial results should be returned by the recognizer as the user speaks (default is false but consider that server may ignore a request for partial results in some or all cases).
   - EXTRA_PREFER_OFFLINE: optional boolean to indicate whether to only use an offline speech recognition engine. The default is false, meaning that either network or offline recognition engines may be used. Depending on the recognizer implementation, these values may have no effect.
   - EXTRA_SPEECH_INPUT_POSSIBLY_COMPLETE_SILENCE_LENGTH_MILLIS: the amount of time that it should take after we stop hearing speech to consider the input possibly complete. This is used to prevent the endpointer cutting off during very short mid-speech pauses.
5) Starting of the speech recognition service by calling the method startListening on the SpeechRecognizer object with the Intent as argument.
6) Call of the methods cancel and destroy for a temporary or definitive deletion of the SpeechRecognizer object (e.g. in onStop and onDestroy methods).

CMUsphinx. It collects over 20 years of the research of the Sphinx Group at Carnegie Mellon University. It is a lightweight library originally created by Kai-Fu Lee, an american computer scientist. We used the *Pocketsphinx* version, which is targeted towards mobile devices. Documentation is unfortunately available only in the original C language, so we learned how to use it from various tutorials (mainly taken from CMUsphinx official webpage) and from the official github page [18]. Generally, CMUsphinx works with HMM-GMM models, as described in section III-A. The main class from *Pocketsphinx* is the SpeechRecognizer class, which allows access to decoder functionalities. The class SpeechRecognizerSetup is in charge of creating the recognizer and is the main object of the following discussion. The first thing to choose is the configuration (you can either call defaultSetup() or setupFromFile(File configFile), where the latter method takes a configuration file consisting of lines containing key-value pairs. Given a setup object you have to call the following methods:

- setAcousticModel: to specify a file containing the acoustic model;
- setDictionary: to specify a file containing the dictionary.

Other parameters can be specified via specific functions. For example, `setKeywordThreshold` sets the threshold for keyword recognition, useful to tune when the actual balance between false positives and negatives is too high. Once all the parameters have been added, then the method `SpeechRecognizerSetup.getRecognizer()` is called to create a `SpeechRecognizer` object. This whole process is a rather time-consuming one, hence it is usually performed in an async task. Analogously to the official Android APIs, an appropriate listener has to be set with the requirement of specifing one of the following search types:

- Keyphrase search: looking for a specific string;
- Keyword search: looking for a string among a list of possible strings;
- Grammar search: recognizing a text satisfying a specific grammar;
- N-gram search: recognizing a text with an n-gram model;

The search type can be specified with various methods, depending on the grammar type and whether you want to pass the path to the grammar file or only a string (grammar must, in any case, be in the JSGF format). Examples of such methods are:

- `addGrammarSearch(String name, File file)`
- `addNgramSearch(String name, File file)`
- `addKeyphraseSearch(String name, String phrase).`

Once all of this setup is done, we can call `startListening(String searchName)`, where the parameter indicates the search type. The listening process will be handled by the previously mentioned `RecognitionListener` object, which must implements six methods: `onBeginningOfSpeech`, `onEndOfSpeech`, `onPartialResult`, `onResult`, `onError`, `onTimeout`. Once the speech event ends, the `onEndOfSpeech` callback will fire, and similarly the `onBeginningOfSpeech` will be called when the speech starts; intermediate results, when available, will be received by `onPartialResult`. In order to stop the recognizer one can call either `stop()` or `cancel()`. Both will stop the recording, but `stop()` will cause the results to be passed to the `onResult` callback, whereas `cancel()` will discard them.

IV. Android application

The application consists of an Android Studio project containing a Java package named `it.unige.ai.speech`. Code is available at:

https://github.com/DiTo97/stunning-potato

Relevant to our project are the following directories:

- *speech/src/main/java/it/unige/ai/speech/* where Java classes are defined separately for recognition and synthesis tasks. The application logic is specified under the "Activities" directories.[2] The main thread is implemented in the the Java class *recognition/activities/CommandPickerActivity.java*.
- *speech/src/main/java/it/unige/ai/base/utils* which contains utility functions;

[2]the term Activity refers to an important Android building block which basically consists of a screen layout and the corresponding application logic.

The application resources such as graphic layout, images and strings are included in the "res" directory of the speech module.

A. Architecture overview

The overall workflow of our application is well described by the diagram reported in Fig. 3, which describes both the general workflow as well as the single activities' one.

Once the user taps on the app symbol to open it, a set-up process starts and the user is greeted by a loading screen, which then turns into a homepage with a list of buttons (six in total). The interaction can then be both vocal or visual: the user can either press on the button which leads to the desired functionality or he can say **"ok, potato"** followed by the name of the desired command (between the "ok, potato" and the activity name he has to wait for an acoustic signal, which notifies that the app is ready to recognize the command).

Then there are two possible scenarios: in the first one the command has not been recognized; user is notified with both a snackbar and an audio message, and the application starts again listening for "ok potato" after the snackbar has been dismissed. In the other case, if the functionality has been implemented the corresponding activity starts, whereas if that's not the case the app simply notifies the user that the activity is yes to be implemented.

The currently implemented activities are the text captioning and the speech recognition, which are responsible for handling the two parts of this SPR project. If the captioning activity is called, the user sees a gallery of images and has the option to click on one of the presented images. When he does the image is magnified with its description written below. The speech synthesis engine takes care of reading out loud the description, and once can choose to listen to it again by pressing on the refresh button which appears above once the first read has been done. While the engine is speaking, the text is highlighted in yellow as it is being read. The user can go back to the gallery and choose all the images that they want to see and listen to.

The other possible activity is the speech recognition one: once this activity starts the user is greeted with a rather simple view, consisting of a button which they must tap in order to speak and have their voice transcribed. Once the button has been pressed the speech recognition engine takes care of listening to the user speaking and, once it detects the end of the speech, it writes on screen what it has understood. This process can be repeated as many times as the user wants by pressing on the record button as many times as they want. Once the user is satisfied with the activity he has been doing, he can tap the "back" button to return to the main menu, where the whole "activity selection" starts again.

The following sections provide implementation details of both the text-to-speech and the speech recognition modules.

B. Text-to-speech module

The text-to-speech synthesis engine is used to implement an automatic reader of the image descriptions provided in the Google Conceptual Captions dataset[3]. By using this dataset, we ideally wanted to emulate the image captioning functionality of this application, which has still to be implemented, and will be added when developing the project for the Computational vision course. We imagined the following situation: each image description of the Google Conceptual captions dataset is supposed to be the output of the evaluation of some machine learning algorithm on the input images. This algorithm could be called, for instance, after taking a picture with Augmented Reality devices such as Smart Glasses. After this action, the visually impaired user will be provided with a realistic audio description of the photographed scene. Because of these assumptions on the problem, we decided to use an image gallery, where each image is clickable. In fact, we wanted to be flexible in testing the application as deeply as possible, therefore we decided that it was better to test the TTS module with a lot of different pairs (image, description) at the same time. In fact, the descriptions contained in the dataset can be either long, medium or short, can regard a lot of different subjects, and the intra-class variance of each type of subject in the images can be arbitrarily large. These are some properties of real life images. Under *speech/src/main/assets* there is a file *image_captioning_small.txt*, containing the image descriptions and links to the images used.

The automatic reading functionality is structured as reported in the diagram in Fig. 3, and it is the main functionality implemented in the TTS module of the Android application. This functionality has already been described in the section "Architecture overview". The TTS module is contained in *stunning-potato/speech/src/main/java/it/unige/ai/speech/synthesis*. This source path is considered with respect to the *project/repository* directory. From a technical point of view, the TTS module functionality which is in charge of implementing the automatic reading follows exactly the standard TTS Android implementation provided in the "Android API" section. For what concerns the choices that we made for the values of the parameters, we report them as a numbered list, along with the corresponding motivations:

1) To specify the language of choice, we used, as already introduced, a Voice object. To do so, we set the locale being used either to `Locale.UK` or `Locale.US` if available, because the Google Conceptual Captions image descriptions are written in english. We also provided an identification string existing in the specified locale which allows us to choose one of the installed voices in the android device. The most important parameter values are `Voice.QUALITY_VERY_HIGH` and `Voice.LATENCY_NORMAL`, which determine, respectively, the quality of the voice which reads the text and the latency between the current time and the first time that a phoneme can be play. We can understand clearly that `Voice.QUALITY_VERY_HIGH` is an obvious choice, since we want the visually impaired to understand clearly the contents of the scene, whereas the motivation for choosing `Voice.LATENCY_NORMAL` resides in the fact that after clicking on the image we may want that some time passes before reading the description to the user. This is done in order to improve user experience when testing the application that comprises the image gallery. We can also specify whether the TTS engine is network-based or not by using the boolean `requiresNetworkConnection`. This may also give us better performances if the TTS engine can be used in both online and offline mode. Since we wanted to use only in-device resources, then we set `requiresNetworkConnection` to false.
2) We used the class AudioAttributes to specify further properties of the audio produced by the speech engine. In particular, we used the `CONTENT_TYPE_SPEECH` constant to tell the TTS engine that the produced audio signal would contain speech (`CONTENT_TYPE_SPEECH` constant) and the `USAGE_ASSISTANT` constant to tell the TTS engine that the produced audio would be used as a response to user queries. The latter choice is motivated by the fact that we identified a general user query as either a click on an image when testing the application using the gallery or as a user request conveyed by a button when using Smart Glasses: in both cases, we answer a request from a user, therefore we supposed that such value for the parameter would make the spoken words more clear.
3) For what concerns the pitch of the TTS engine set through the `setPitch` method, it is usually better to keep the default value, which equals 1.0, because the values for the pitch of every phoneme are estimated directly by the engine from the text: changing the "overall" pitch of the TTS engine programmatically could in fact lead to worse performances for what concerns the quality of the sounds. For example, the TTS engine may end up sounding robotic when lowering too much the value for the parameter with female voices. Since we were not sure if there was a mapping of some kind between the "overall" pitch and the phoneme pitch, we decided to leave it unchanged.
4) For what concerns the parameter `TextToSpeech.Engine.KEY_PARAM_STREAM`, we used the constant `AudioManager.STREAM_VOICE_CALL`, because in our application we are mainly trying to read out loud image captions to a user, therefore we supposed that an audio stream with a voice call shape would the most appropriate for the task to be performed.
5) For what concerns the calls to the speak method, we used the parameter value `TextToSpeech.QUEUE_FLUSH`. This value is useful when another text-to-speech

[3]available at https://ai.google.com/research/ConceptualCaptions/

synthesis request is triggered before the current request in progress has finished. By using `TextToSpeech.QUEUE_FLUSH`, we interrupt the current description being spoken and we start immediately to produce the audio for the next one. This is an implementation choice used to improve user experience, both from the testing app point of view and the future app fully dedicated to the visually impaired.

6) When calling the `setSpeechRate` method, we used a value slightly lower than the default: 0.9 instead of 1.0. This value for the speech rate was adopted because of the properties of real life images already introduced in this section: in fact, the length and the contents of an image description depend on a wide variety of factors, and may affect how understandable the speech audio is. Therefore, we understood by performing experiments that a slightly lower value for the parameter was fine.
7) The chosen value for `TextToSpeech.Engine.KEY_PARAM_PAN` is the default one, which is 0. We decided not to shift the produced audio closer to one of the two human ears. We may decide, in the future, to tune this parameter in order to extend our application to acoustically impaired people, for instance to those that are not able to listen to speech very well from one of the two ears.
8) The chosen value for `TextToSpeech.Engine.KEY_PARAM_VOLUME` is the default one, which is 1. In order to facilitate the comprehension of the scene by the visually impaired, we wanted the volume to be as high as possible. In fact, this can help especially with long descriptions, because if they are read too fast then the user could lose the main meaning of the phrases. Moreover, we gain a further level of clarity in the produced speech, because the value for `TextToSpeech.Engine.KEY_PARAM_VOLUME` combines with the chosen value for the speech rate.

For what concerns the engine that we used in our application, we decided to employ the google TTS engine. Such choice was motivated by the following facts:

1) The google TTS engine is the default option for some Android devices and it is compatible with multiple already existing applications, therefore we thought that such an engine would have scaled nicely to the ideas of the application that we had in mind. Moreover, it is usually already installed in most Android devices, therefore the TTS functionalities are more likely to be available for a wider range of users.
2) Google TTS supports a large number of languages and is easy to set up and to use.
3) In its offline mode, the google TTS engine it's free to use.
4) The corresponding cloud-based alternative is built on Wavenet, therefore we supposed that the corresponding offline TTS version was a state of the art as well.
5) Even if we didn't find any evidence of the specific model used by the offline engine, we supposed that this engine was built on a similar model to Wavenet. This idea is supported by the fact that the Google TTS engine obliges to download the full language pack of the language of choice on premises.

C. Speech recognition module

The speech recognition engine is composed of two distinct instances of the APIs presented in the par. III-B. The use of two different recognizers can be justified by quoting Android `SpeechRecognizer` documentation verbatim: "The implementation of this API is likely to stream audio to remote servers to perform speech recognition. As such this API is not intended to be used for continuous recognition, which would consume a significant amount of battery and bandwidth." The class is presumably an entry point for an HMM-DMM model running on the cloud with a consequent delay in the response, depending on the complexity of the requested services and on the availability of the Google servers. This is the reason why we opted for multiple speech recognizers:

- Pocketsphinx is in charge of recognizing only the initial phrase;
- Android Speech Recognizer handles the other interactions: command recognition and speech transcript.

The folder named "aars" contains the file needed to install and run Pocketsphinx. The Pocketsphinx speech recognizer is initialized in the main application class, i.e. *CommandPickerActivity.java*, with the following specifications:

- default settings are used for the initialization through the method `defaultSetup()`. Default parameters are contained in the file *speech/assets/sync/en-us-ptm/feat.params*, for example[4]:
  - `lowerf 130`: lower edge of filters
  - `upperf 6800`: upper edge of filters
  - `nfilt 25`: number of filter banks
  - `transform dct`: transform type to calculate cepstra (legacy, dct, or htk)
  - `lifter 22`: length of sin-curve for liftering, or 0 for no liftering.
- the file `"cmudict-en-us.dict"` is provided as dictionary, which contains 133.420 entries of the form word-phonemes (e.g. `zurich Z UH R IH K`);
- the files contained in the directory *speech/assets/sync/en-us-ptm* are specified as acoustic model (e.g. transition matrices, mean and variances for the Gaussian mixture model)
- the keyphrase search option was specified with `ACTIVATION_keyphrase = "ok potato"`

As soon as the keyphrase is recognized, the method `stop()` is called to interrupt the Pocketsphinx service and the Android speech recognizer is started with the method

[4]a limited documentation is available at https://github.com/cmusphinx/pocketsphinx/blob/master/doc/pocketsphinx_continuous.1

`startListening()`. The settings used for the Android speech recognizer are the following:

- `EXTRA_LANGUAGE_MODEL` was set to `LANGUAGE_MODEL_FREE_FORM` as suggested by the official documentation;
- `EXTRA_SPEECH_INPUT_POSSIBLY_COMPLETE_SILENCE_LENGTH_MILLIS` was empirically set to 5000;
- `EXTRA_MAX_RESULTS` was set to 5.

The Android recognizer is also used with the same settings in the class *SpeechRecognitionActivity.java* for the speech-to-text interface.

D. Testing

The app was developed and tested with Android Studio 4.0. The debug mode was used on a physical Samsung Galaxy Tab S5e device with Android API 29. Several tests were performed to check the effectiveness of all the functionalities and the user experience (interface, time of interaction, permissions ...). The main issues were:

- Speech recognition module: (i) the initial keyphrase expression ”ok app” was not fairly recognized. Following the advice provided by a developer involved in the CMUsphinx project[5] we increased the lenght of the keyphrase. According to him, keyphrase expression searched by Pocketsphinx must contain prefereably 3 or 4 syllables. (ii) only a strict British pronounce is easily understood by the Android Speech recognizer; furthermore, single words in compact expression (e.g. ”pose detection”) have to be spelled separately.
- Text-to-speech module: sentences lacking punctuations cause the speech synthesis to be accelerated in a non natural way. As a consequence, the speech rate was tuned accordingly, as detailed in the par. III-A.

Screenshots from the app are shown in Fig. 4.

V. Conclusion and future works

In this project we implemented and tested vocal functionalities for a more ambitious Android application, intended to provide assistance to visually impaired people. With few tunings, both text-to-speech and speech recognition services were relatively easy to implement and seem to meet our requirements of providing efficient and engaging interactions to the user. Our application will be extended in a future project by adding a computer vision module. We would like to perform possibly real-time semantic analysis of specific scenes captured with an Android camera. For instance, one of the main functionalities that will be added in this module is the automatic real-time captioning functionality, which will trigger again the TTS functionalities to read out loud the generated image description. These descriptions ideally should result to be fluent and should have a fine level of detail: we don’t want to label each part of the scene with a single label, but with a whole phrase or sentence, possibly with punctuation. Because the result of the two operations will be transformed into speech, the goal is to not disturb the person with continuous feedback but rather provide a detailed scene description or text reading at the person’s request. Overall these functionalities are intended to replace some compromised visual cognitive tasks of daily utility.

[5] https://stackoverflow.com/questions/5184233/keyword-spotting-in-speech

Authors’ contributions

F.M. and M.G. worked on project n. 10 ”Text-to-speech app” and are the authors of par. II (Text-to-speech systems) and par. IV-B (Text-to-speech module). S.B and G.L worked on project n. 6 ”Speech Recognition app” and are the authors of par. III (Speech Recognition systems) and par. IV-C (”Speech reconition module”). All the students contributed equally to the other paragraphs and to the development of the Android application.

Acknowledgment

The students acknowledge Professor Zied Mnasri who allowed them working on this project and provided useful advices. F.M. would like to thank Aaron Zheng as a source of inspiration for the entire project.

References

[1] Envision glasses official webpage https://www.letsenvision.com/glasses, accessed on 24th June, 2020.

[2] OrCam official page https://www.orcam.com/en/myeye2/, accessed on 24th June, 2020.

[3] A 2019 Guide to Speech Synthesis with Deep Learning, https://heartbeat.fritz.ai/a-2019-guide-to-speech-synthesis-with-deep-learning-630afcafb9dd, accessed on 24th June, 2020.

[4] Wang, Y., Skerry-Ryan, R.J., Stanton, D., Wu, Y., Weiss, R.J., Jaitly, N., Yang, Z., Xiao, Y., Chen, Z., Bengio, S., Le, Q.V., Agiomyrgiannakis, Y., Clark, R., & Saurous, R.A. (2017). Tacotron: Towards End-to-End Speech Synthesis. INTERSPEECH.

[5] Oord, A. V. D., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., ... & Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. arXiv preprint arXiv:1609.03499.

[6] Arik, S.Ö., Chrzanowski, M., Coates, A., Diamos, G.F., Gibiansky, A., Kang, Y., Li, X., Miller, J., Ng, A., Raiman, J., Sengupta, S., & Shoeybi, M. (2017). Deep Voice: Real-time Neural Text-to-Speech. ArXiv, abs/1702.07825.

[7] Android TTS official documentation, https://developer.android.com/reference/android/speech/tts/TextToSpeech, accessed on 24th June, 2020.

[8] Wang, D., Wang, X., & Lv, S. (2019). An Overview of End-to-End Automatic Speech Recognition. Symmetry, 11(8), 1018.

[9] Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A. R., Jaitly, N., ... & Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal processing magazine, 29(6), 82-97.

[10] Murphy K., Machine learning: a probabilstic perspective, MIT Press, 2012

[11] Gemmeke, J. F., Ellis, D. P., Freedman, D., Jansen, A., Lawrence, W., Moore, R. C., ... & Ritter, M. (2017, March). Audio set: An ontology and human-labeled dataset for audio events. In 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP) (pp. 776-780). IEEE.

[12] Matlab official tutorial ”Speech Command Recognition Using Deep Learning”, https://it.mathworks.com/help/deeplearning/ug/deep-learning-speech-recognition.html, accessed on June 23rd, 2020.

[13] Nuance company R&D official blog, https://research.nuance.com/category/speech-recognition/, accessed on June 23rd, 2020.

[14] Mana, F., Weninger, F., Gemello, R., & Zhan, P. (2019, December). Online Batch Normalization Adaptation for Automatic Speech Recognition. In 2019 IEEE Automatic Speech Recognition and Understanding Workshop (ASRU) (pp. 875-880). IEEE.

[15] Improving Speech Representations and Personalized Models Using Self-Supervision, https://ai.googleblog.com/2020/06/improving-speech-representations-and.html, accessed on June 23rd, 2020.

[16] Large-Scale Multilingual Speech Recognition with a Streaming End-to-End Model, https://ai.googleblog.com/2019/09/large-scale-multilingual-speech.html, accessed on June 23rd, 2020.

[17] Android Speech package official documentation, https://developer.android.com/reference/android/speech/package-summary, accessed on June 23rd, 2020.

[18] Pocketsphinx official documentation, https://cmusphinx.github.io/doc/pocketsphinx/index.html, official website with Android tutorial https://cmusphinx.github.io/wiki/tutorialandroid/ accessed on June 23rd, 2020.

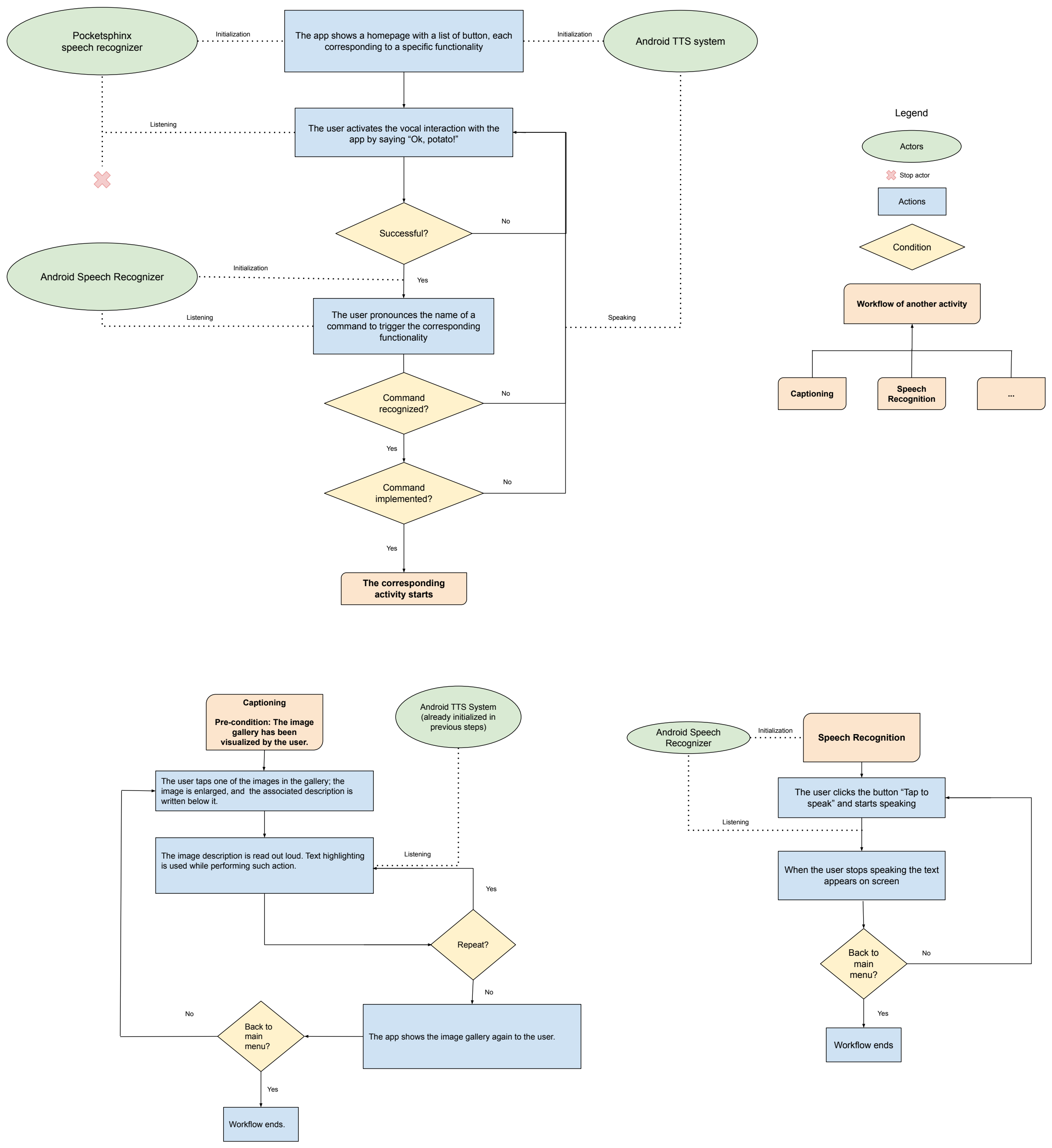
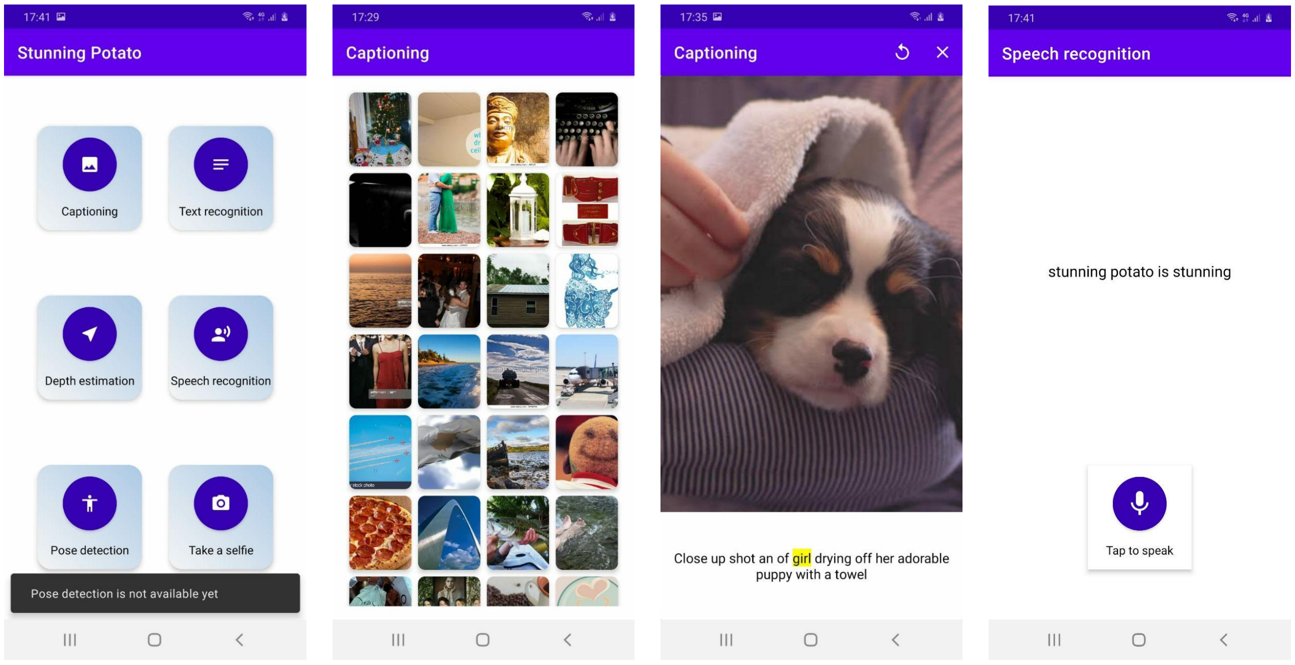

Pocketsphinx speech recognizer
Initialization
The app shows a homepage with a list of button, each corresponding to a specific functionality
Initialization
Android TTS system
Listening
The user activates the vocal interaction with the app by saying "Ok, potato!"
Successful?
No
Android Speech Recognizer
Initialization
Yes
Listening
The user pronounces the name of a command to trigger the corresponding functionality
Speaking
Command recognized?
No
Yes
Command implemented?
No
Yes
The corresponding activity starts
Legend
Actors
Stop actor
Actions
Condition
Workflow of another activity
Captioning
Speech Recognition
...
Captioning
Pre-condition: The image gallery has been visualized by the user.
Android TTS System (already initialized in previous steps)
The user taps one of the images in the gallery; the image is enlarged, and the associated description is written below it.
The image description is read out loud. Text highlighting is used while performing such action.
Listening
Yes
Repeat?
No
No
Back to main menu?
The app shows the image gallery again to the user.
Yes
Workflow ends.
Android Speech Recognizer
Initialization
Speech Recognition
The user clicks the button "Tap to speak" and starts speaking
Listening
When the user stops speaking the text appears on screen
Back to main menu?
No
Yes
Workflow ends

Fig. 3: Top: (left) flow chart of the Android app, (right) legend of the figure. Bottom: flow chart of the two functionalities implemented as distinct Activities: (left) image captioning with a text-to-speech system, (right) speech recognition and transcript.



Pose detection is not available yet

Close up shot an of girl drying off her adorable puppy with a towel

stunning potato is stunning

(a) (b) (c) (d)

Fig. 4: Screenshots from the app: (a) home screen with an interactive message to the user, (b) gallery for the image captioning, (c) captioning of a single image, (d) speech-to-text functionality