Université De Montpellier - Faculté Des Sciences
Année Universitaire 2020 - 2021
L3 informatique
TER HLIN601

Rapport de projet T.E.R. L3 :

Application Interactive Vocale

Étudiants :

Anh CAO
Ali DABACHIL
Marwan MASHRA
Merwane RAKKAOUI

Encadrant :

Abdelhak-Djamel SERIAI

Nous tenons à remercier notre encadrant M. Abdelhak-Djamel SERIAI
pour son accompagnement, sa bienveillance et ses conseils précieux tout au long de ce projet qui nous ont permis de le porter à son état actuel.

Sommaire

Partie 1

Introduction

1.1 Contexte général du projet

L'interface Homme-machine (IHM) se développe et les applications sont de plus en plus conçues pour être interactives. Les assistants vocaux tels que Google et Siri ont montré l'utilité de pouvoir interagir avec une application via des commandes vocales. Cela a ouvert les portes à de nouvelles approches dans le conception d'IHM où l'utilisateur peut communiquer avec l'application directement avec sa voix.

Dans le cadre du module HLIN601 TERL3, nous avons développé une application interactive vocale capable de réaliser de nombreuses opérations (comme la reconnaissance des émotions, faciale ainsi que des recherches sur le Web etc.). Comme indiqué précédemment, l'interaction entre l'utilisateur et l'application se fait entièrement de manière vocale.

1.2 Objectif et cahier des charges

L'objectif de ce projet est d'utiliser les différentes APIs open source disponibles pour développer une application en Python dont l'objectif est de :

— Utiliser la Webcam pour recueillir le visage d'une personne.
— Reconnaître l'émotion d'une personne grâce à des APIs open source.
— Reconnaître le visage d'une personne grâce à des APIs open source.
— Réaliser des recherches sur le Web.
— Permettre les interactions vocales grâce à des APIs open source pour la synthèse vocale et la reconnaissance de la parole.
— Assurer que toute interaction se fait vocalement depuis le lancement de l'application jusqu'à son arrêt.
— Utiliser le Machine Learning pour développer notre propre modèle de reconnaissance des émotions.

Partie 2

Technologies utilisées

2.1 Langage utilisé

Nous avons décidé de développer entièrement l'application en **Python**. Nous l'avons choisi pour les nombreux avantages qu'il nous offre pour le développement de notre application. C'est un langage de programmation très riche et complet contentant toutes les APIs dont nous aurons besoin pour le développement de notre application. Il est aussi très populaire pour faire du Machine Learning dont nous avons besoin pour la dernière partie de l'application. Sans oublier que c'est un langage simple et facile ce qui nous permet de nous concentrer sur le fonctionnement de l'application et d'oublier la syntaxe.

2.2 APIs utilisées

De nombreuses APIs ont été utilisées pour le développement de notre application. Ceci afin d'implémenter des fonctionnalités essentielles dont les suivantes :

1. **La reconnaissance faciale :**
   Nous avons utilisé ici l'API ***face_recognition*** [1] permettant de comparer deux photos pour identifier le visage d'une personne avec une précision de 99.38%.
   Exemple :

```
from face_recognition import load_image_file,face_encodings,compare_faces

known_image = load_image_file("biden.jpg")
unknown_image = load_image_file("unknown.jpg")

known_encoding = face_encodings(known_image)[0]
unknown_encoding = face_encodings(unknown_image)[0]

results = compare_faces([known_encoding], unknown_encoding)
```

1. https://pypi.org/project/face-recognition/

2. **Synthèse vocale :**
   Nous avons utilisé ici l'API ***gTTS*** (Google Text-to-Speech)[2] permettant de transformer du texte en fichier audio. Exemple :

```
from gtts import gTTS

myobj = gTTS(text="Bonjour tout le monde", lang='fr', slow=False)
myobj.save("bonjour.mp3")
```

3. **La reconnaissance de la parole :**
   Nous avons utilisé ici l'API ***speech_recognition***[3] permettant de capturer la parole via le microphone et de le convertir en texte.

```
import speech_recognition as sr

r = sr.Recognizer()
with sr.Microphone() as source:
    audio = r.listen(source)
    speech = r.recognize_google(audio, language="fr-FR")
    print(speech)
```

4. **La reconnaissance des émotions :**

   Nous utilisons ici le projet ***realtime-facial-emotion-analyzer***[4] réalisé en 2018 par ***Susanta Biswas*** et ***Sagnik Chatterjee*** et dont le code source se trouve libre de droits sur GitHub. Il permet de détecter l'émotion à partir d'une photo donnée avec une précision de 63%. Les émotions reconnues sont la joie, La tristesse, la surprise, la peur, le dégoût, la colère et l'indifférence.

5. **La création d'un modèle prédictif :**

   Nous utilisons ici ***TensorFlow***[5] qui est un outil d'apprentissage automatique développé par Google. Il permet d'utiliser l'apprentissage automatique (Machine Learning) pour développer des modèles prédictifs avec un haut niveau d'abstraction. Il intègre également ***Keras***[6] qui est une API d'apprentissage automatique en Python très populaire.

2. https://pypi.org/project/gTTS/
3. https://pypi.org/project/SpeechRecognition/
4. https://github.com/susantabiswas/realtime-facial-emotion-analyzer
5. https://www.tensorflow.org/?hl=fr
6. https://keras.io/

Partie 3

Développement de l'application

3.1 Conception et implémentation des composants

Pour le développement de notre application, nous avons utilisé la structure de Micro-services ce qui signifie que l'application intègre un ensemble de composants (services). Chaque composant fonctionne de manière indépendante et assure une fonctionnalité précise de l'application. L'application intègre principalement quatre composants qui sont :

1. **emotion_recognition :**
   Ce composant est chargé de faire de la reconnaissance d'émotions à partir d'une image donnée. Il a été réalisé à partir du projet *realtime-facial-emotion-analyzer* disponible en open source sur GitHub. Il utilise un modèle d'un réseau neuronal convolutif avec une précision de 63%.

2. **face_recognizer :**
   Ce composant est chargé de faire de la reconnaissance faciale. En utilisant l'API face_recognition, il reconnaît un visage à partir d'une image donnée contenu dans un dossier regroupant des fichiers de visages. En annexe, un exemple montrant la structure de ce fichier.

3. **text_to_speech :**
   Ce composant est chargé de faire de la synthèse vocale. Il utilise l'API gTTs pour transformer le contenu d'un fichier texte en fichier audio. Il demande ensuite le lancement de ce fichier audio avant de le supprimer.

4. **speech_to_text :**
   Ce composant est chargé de faire de la reconnaissance de la parole. Il enregistre la voix de l'utilisateur via le microphone et puis le transforme en texte à l'aide de l'API speech_recognition.

Chaque composant a été empaqueté avec toutes ses dépendances dans un conteneur virtuel à l'aide de *Docker* [1]. Cela permet d'assurer le bon fonctionnement de ces différents composants et d'éviter les potentiels problèmes de compatibilité ou des dépendances. Toutefois, isoler les composants dans des environnements virtuels séparés rend plus difficile l'accès aux périphériques. Comme certains composants ont besoin d'accéder à des périphériques, nous avons mis en place une solution basée sur l'architecture Client-Serveur.

Nous avons mis en place un serveur avec *Flask* [2] et nous l'avons déployé avec *WSGI* [3]. Les composants envoient des requêtes GET au serveur pour lui demander de réaliser des tâches précises nécessitant un accès aux périphériques. Par exemple, le composant *text_to_speech* envoie une requête pour demander le lancement d'un fichier audio et le composant *speech_to_text* envoie une requête pour demander l'enregistrement via le microphone.

Une image est construite pour chaque composant grâce à un fichier *docker-compose.yml*. Ce dernier construit les images à partir d'un fichier *Dockerfile* se trouvant dans chaque composant. Ces images permettent de créer des conteneurs qui sont des processus lancés dans des environnements virtuels pour réaliser la tâche associée au composant. En annexe, le fichier *docker-compose.yml* ainsi qu'un exemple d'un fichier *Dockerfile*.

3.2 Intégration

L'intégrateur de notre application utilise les quatre composants pour réaliser les tâches nécessaires pour son fonctionnement. Bien que ces composants peuvent être utilisés via la ligne de commande, nous avons préféré de les utiliser avec le module Python *docker*. Des conteneurs sont donc créés et auto-détruits au fur et à mesure de l'exécution de notre application. Toutes les communications internes se font par des écritures et lectures dans des fichiers intermédiaires. D'ailleurs, comme les conteneurs sont lancés dans des environnements virtuels, ils n'ont pas accès à l'espace de stockage de la machine. Nous avons donc créé un volume (dossier) partagé entre les conteneurs.

D'autres responsabilités sont aussi accordées à l'intégrateur. Par exemple, il est chargé de démarrer/arrêter le serveur comme un processus détaché, d'accéder à la Webcam pour prendre des photos et aussi d'ouvrir le navigateur à la demande de l'utilisateur.

1. (https://www.docker.com/) **Docker** est un logiciel libre permettant de lancer des applications dans des conteneurs virtuels.
2. (https://flask.palletsprojects.com/en/1.1.x/) **Flask** est un framework de développement web en Python pour créer des applications Web.
3. (https://docs.python.org/3/library/wsgiref.html) **WSGI** (Web Server Gateway Interface) est un module Python pour créer des interfaces entre des serveurs et des applications web.

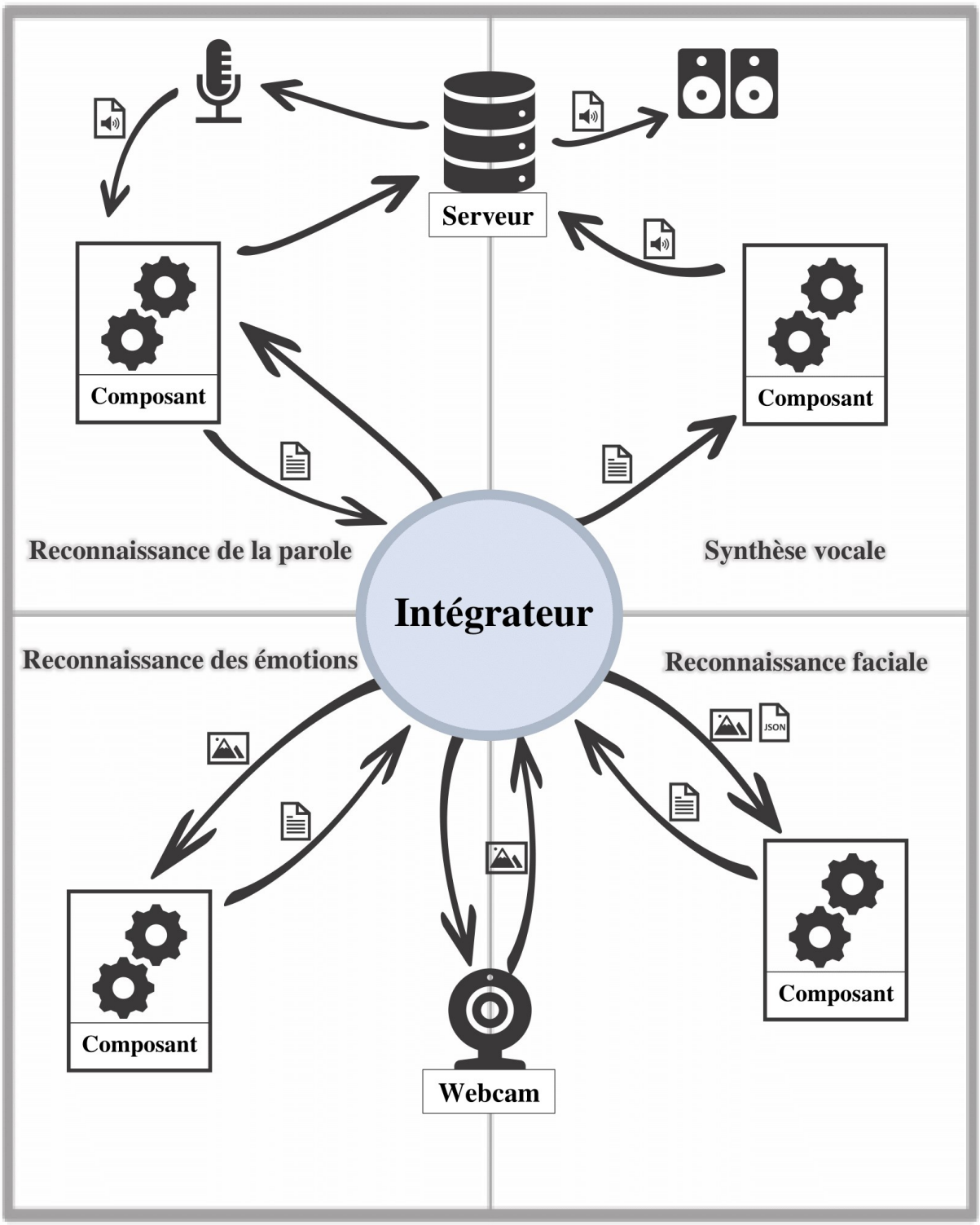


FIGURE 3.1 – schéma fonctionnel

3.3 Interaction et scénarios d'usage

Toute interaction avec l'application depuis le lancement jusqu'à la fin de l'exécution se fait vocalement. Il n'existe donc pas d'interface Homme-machine graphique. Après le démarrage, l'application se met en mode d'écoute. La commande vocale "Ok Assistant" permet d'appeler l'assistant pour qu'il commence à recevoir les ordres de l'utilisateur. À part cette commande, il n'existe pas de commande vocale spécifique à connaître pour communiquer avec l'application. L'utilisateur peut parler librement et l'application essayera de répondre au mieux aux demandes de l'utilisateur.

Traitement des demandes

Les demandes vocales de l'utilisateur sont converties en texte avant de passer par plusieurs étapes de traitement de d'analyse pour décider l'action à réaliser. Ces étapes peuvent se résumer en deux parties :

1. **Analyse syntaxique :**

   L'analyse syntaxique consiste à mettre en évidence la structure d'un texte afin de ne plus le voir comme une chaîne de caractères mais comme une phrase écrite dans une langue naturelle et décomposée en nom, verbe, adjectif, proposition...etc. Ici nous utilisons *TreeTagger* [4] pour remettre les mots à leur origine (nom singulier, verbe à l'infinitif...etc) et ensuite les trier par ordre d'importance (les noms propres, les noms, les verbes, les adjectives et ensuite le reste). Ces mots sont par la suite utilisés pour la prise de décision.

2. **Prise de décision :**

   Ici nous utilisons les mots fournis par l'analyse syntaxique pour trouver des mots clés permettant de décider l'action qui répond au mieux à la demande de l'utilisateur. Plusieurs actions ont été implémentées comme reconnaître l'utilisateur par la reconnaissance faciale, enregistrer le visage d'un nouvel utilisateur, reconnaître l'émotion de l'utilisateur, ouvrir le navigateur pour réaliser une recherche sur Google ou sur YouTube.

   Cependant, la décision ne peut pas être prise en se basant uniquement sur la dernière demande de l'utilisateur. Conserver le contexte de la demande est indispensable car des simples phrases comme "Oui" ou "Non" n'ont pas de sens sans leur contexte. Il faut donc que l'application soit capable de tenir des conversations entières avec l'utilisateur afin qu'elle puisse prendre les bonnes décisions. Pour ce faire, des scénarios ont été créés et implémentés dans une structure prenant la forme d'un *Arbre de décision*.

4. (`https://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/`) **TreeTagger** est un outil qui permet d'annoter un texte avec des informations sur les parties du discours (genre de mots : noms, verbes, infinitifs et particules) et des informations de lemmatisation.

3.4 Réalisation de notre propre modèle

La dernière partie de développement de l'application consistait à créer notre propre modèle prédictif de reconnaissance des émotions. Pour ce faire, nous avons utilisé l'API Keras intégrée dans TensorFlow.

En apprentissage automatique, un réseau de neurones artificiels (ANN) est un système qui permet de créer et de développer des modèles capables de faire des prédictions. Sa conception est inspirée du fonctionnement des neurones biologiques. Il existe plusieurs types de réseau de neurones artificiels servant à résoudre des problèmes variés comme le réseau de neurones récurrents (RNN) utilisé pour la reconnaissance de la parole ou le réseau neuronal convolutif (CNN) utilisé pour la vision par ordinateur. Comme nous voulons que notre modèle détecte l'émotion à partir d'une image donnée, nous utiliserons un réseau neuronal convolutif.

Un réseau neuronal convolutif est composé d'une succession de couches dont chacune prend ses entrées sur les sorties de la précédente (bibliographie [1] et [2]). Il y a trois types de couches :

1. **Input :** Les neurones contiennent les valeurs d'entrée du programme. Dans le cas de notre modèle, les valeurs d'entrée sont les pixels d'une image en noir et blanc de taille 48 x 48 x 1.

2. **Hidden :** Les neurones reçoivent un signal (valeur) des neurones précédents, y appliquent un certain traitement et transmettent le signal aux neurones successeurs. Souvent un CNN possède plusieurs couches de type Hidden comme c'est la cas de notre modèle.

3. **Output :** Le nombre de neurones de cette couche dépend du problème que nous cherchons à résoudre. Dans le cas de notre modèle, cette couche possède 7 neurones correspondant aux probabilités des 7 émotions à détecter.

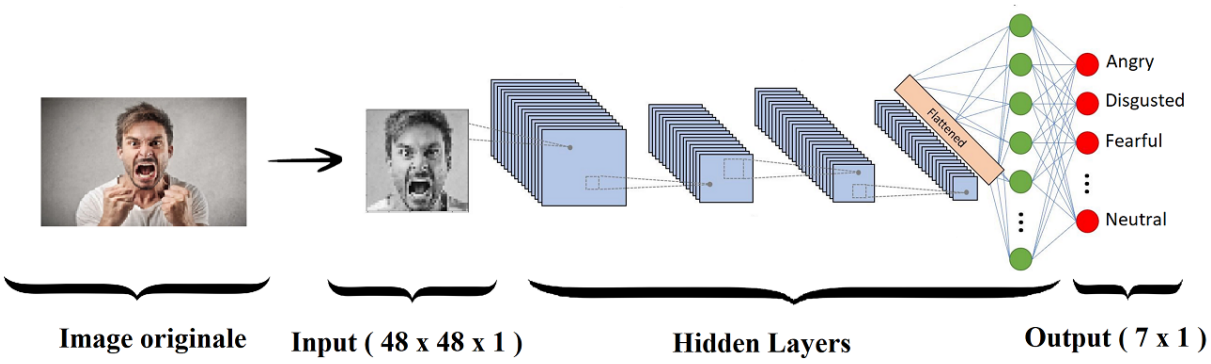


FIGURE 3.2 – Architecture d'un modèle CNN

Dans le développement d'un modèle, trouver l'architecture permettant d'obtenir les meilleurs résultats à un problème donné est souvent une étape dure et qui demande beaucoup d'études et de recherches (bibliographie [3] et [4]). Nous avons donc décidé de reprendre la même architecture utilisée dans le projet *realtime-facial-emotion-analyzer*. En annexe, l'architecture ainsi que le code source que nous avons utilisé pour créer le modèle.

L'entraînement d'un modèle est le processus visant à trouver les meilleurs valeurs des paramètres (poids associés aux neurones) du modèle afin de minimaliser la différence entre la prédiction faite et le résultat voulu. Ce processus nécessite un grand ensemble de données dans le but d'avoir une meilleure précision . Pour l'entraînement de note modèle, nous avons utilisé un ensemble de données mis en ligne sur *Kaggle* [5] dans le cadre de *Facial Expression Recognition Challenge* [6] en 2013. Après l'entraînement, notre modèle a atteint une précision de 0.627 ce qui reste très correct.

5. (`https://www.kaggle.com/`) **Kaggle** est une plateforme web organisant des compétitions en science des données.

6. `https://www.kaggle.com/debanga/facial-expression-recognition-challenge?rvi=1`

Partie 4

Algorithmes et Structures de Données

Arbre de décision

Un arbre de décision[1] est une structure de données représentant un ensemble de choix sous la forme d'un arbre. Dans notre arbre, les noeuds représentent les suites possibles de la conversation entre l'application et l'utilisateur. Chaque nœud possède les propriétés suivantes :

- **tag :** l'identifiant unique des nœuds de l'arbre.
- **keywords :** la liste des mots clés décrivant le nœud actuel.
- **text :** une liste contentant des parties de la conversation avec l'utilisateur. chaque partie est une liste des paroles possibles parmi lesquelles l'application fera un choix aléatoire.
- **context :** une variable vide par défaut. Elle permet de passer des informations du contexte au nœud suivant comme l'émotion détectée, le nom de l'utilisateur dont le visage a été détecté...etc. Il est également possible de l'utiliser dans l'attribut text. Par exemple, dans la phrase "Bonjour {context}", {context} sera remplacé par le contenu de l'attribut context.
- **action :** l'action à réaliser. Plusieurs actions ont été implémentées, nous en citons les suivantes :
  - listen : pour recevoir une requête de l'utilisateur.
  - google : pour réaliser une recherche sur Google.
  - emotion_recognition : pour reconnaître l'émotion de l'utilisateur.
  - face_recognizer : pour reconnaître le visage de l'utilisateur.
  - face_register : pour enregistrer le visage de l'utilisateur.
- **next :** une liste des nœuds suivants parmi lesquels l'application choisira selon les mots clés retrouvés dans la demande de l'utilisateur. Les nœuds peuvent soient être écrits explicitement, soient être représentés par leur tag.

En annexe, un exemple extrait de notre arbre de décision.

1. `https://fr.wikipedia.org/wiki/Arbre_de_décision`

Algorithme récursif

Nous avons mis en place un algorithme récursif capable de gérer les interactions avec l'utilisateur à partir des scénarios implémentés dans l'arbre de décision.

Algorithm 1: Algo (Node : le noeud actuel ou son tag)

```
1  if isTag(Node) then
2  |  Node := getNodeByTag(Node) ;        // remplacer le tag par le noeud
3  end
4  say := "";
5  foreach choices in Node['text'] do
6  |  choice := randomChoice(choices) ;      // choisir une phrase aléatoire
7  |  choice := replaceContext(choice) ;              // remplacer {context}
8  |  say += choice;
9  end
10 text_to_speech(say);
11 response := doAction(Node['action']) ;                // réaliser l'action
12 key_words := analyse_syntaxique(response);
13 foreach node in Node['next'] do
14 |  if isTag(node) then
15 |  |  node := getNodeByTag(node);
16 |  end
17 |  foreach word in key_words do
18 |  |  if word in node['keywords'] then
19 |  |  |  Algo(node);
20 |  |  |  return
21 |  |  end
22 |  end
23 end
```

Partie 5

Analyse des résultats

Tous les composants que nous avons implémentés pendant le développement de l'application fonctionnent correctement et réalisent les tâches demandées. L'intégrateur arrive à utiliser les différents composants pour assurer le bon fonctionnement de l'application. Les interactions avec l'application depuis son lancement jusqu'à son arrêt sont entièrement vocales comme prévu. Les actions et les scénarios implémentés couvrent toutes les fonctionnalités précédemment citées dans le cahier des charges. De plus, la structure d'arbre que nous avons mis en place est complète et rend l'ajout de nouveaux scénarios très facile.

Quant à notre modèle de reconnaissance des émotions, sa précision est de 0.627 ce qui reste très raisonnable. Nous l'avons testé et nous avons obtenu des performances correctes. L'émotion est souvent correctement détectée quand elle apparaît clairement sur l'image. En revanche, certaines émotions (comme la joie ou la colère) sont beaucoup mieux détectées que d'autres (comme la surprise) ce qui peut être dû aux données d'entraînement que nous avions utilisées. En effet, les données d'entraînement disponibles pour chaque émotion varient en quantité entre 500 et 8000 images.

Partie 6

Gestion du projet

6.1 Outils du travail en collaboration

➢ **GitHub :** un service web d'hébergement basé sur le logiciel de gestion de versions Git. Nous l'avons utilisé pour l'hébergement de notre code source.

➢ **Jira :** un outil de gestion de projet permettant de faciliter le suivi des tâches, d'identifier les blocages et de partager des informations entre les membres d'une équipe. Nous l'avons utilisé pour suivre l'avancement des tâches hebdomadaires.

➢ **Overleaf :** un éditeur LaTeX en ligne permettant de travailler sur les documents de manière collaborative et en temps réel. Nous l'avons utilisé pour la rédaction des documents hebdomadaires (compte-rendu,bilan) ainsi que le rapport.

➢ **Google Drive :** un service de stockage et de partage de fichiers dans le cloud lancé par la société Google. Nous l'avons utilisé pour le partage des documents ainsi que les vidéos de démonstration.

6.2 Répartition du travail dans le temps

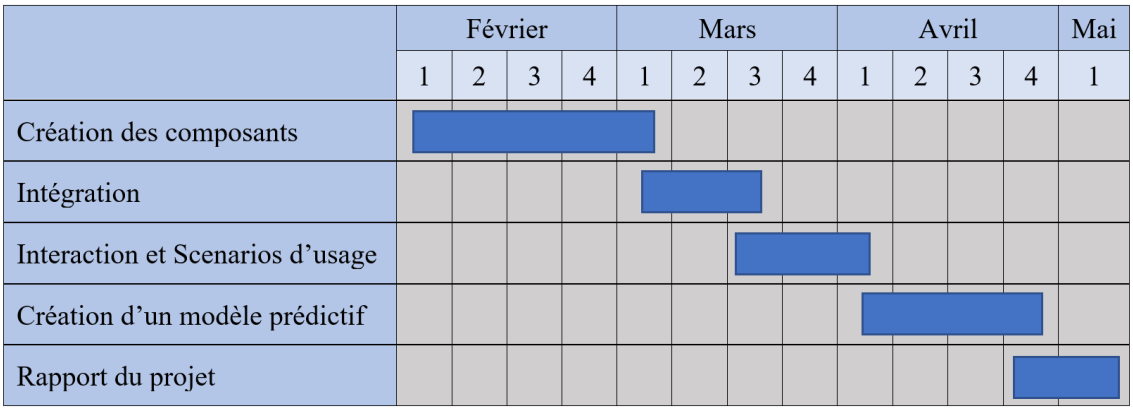


FIGURE 6.1 – Diagramme de Gantt

Partie 7

Conclusion et perspective

Finalement, nous considérons que le cahier des charges a été bien respecté. Toutes les fonctionnalités prévues dans l'application ont bien été implémentées. L'interaction vocale avec l'application est très réussie grâce aux nombreux scénarios implémentés. De plus, l'ajout de nouveaux scénarios ne nécessite pas de modification dans le code source à part pour ajouter une nouvelle action. Nous pouvons donc ajouter facilement des scénarios et des actions comme lancer une musique, ouvrir une photo, réaliser des recherches sur certaines sites Web ou encore d'autres actions internes ou externes. Pour aller plus loin dans le développement de l'application, l'implémentation d'un Chabot peut même être envisagé.

Partie 8

Bibliographie

[1] TowardsAI - *Explication générale du CNN* [Online ; accessed 16-April-2021]

[2] TowardsDataScience - *Explication plus précise du CNN* [Online ; accessed 16-April-2021]

[3] Simplilearn - *Conception d'un modèle et explications* [Online ; accessed 16-April-2021]

[4] DeepLearningAI - *Convolutional Neural Networks (Course 4 of the Deep Learning Specialization)* [Online ; accessed 16-April-2021]

Partie 9

Annexe

9.1 Fichier de sauvegarde de visages

Un exemple montrant la structure utilisée dans le fichier Json contentant la collection des visages sauvegardés.

```
{
    "faces": [
        {
            "name": "Anh CAO",
            "path_img": "faces/Anh_CAO.jpg"
        },
        {
            "name": "Ali DABACHIL",
            "path_img": "faces/Ali_DABACHIL.jpg"
        },
        {
            "name": "Marwan MASHRA",
            "path_img": "faces/Marwan_MASHRA.jpg"
        },
        {
            "name": "Merwane RAKKAOUI",
            "path_img": "faces/Merwane_RAKKAOUI.jpg"
        }
    ]
}
```

9.2 Dockerfile et docker-compose

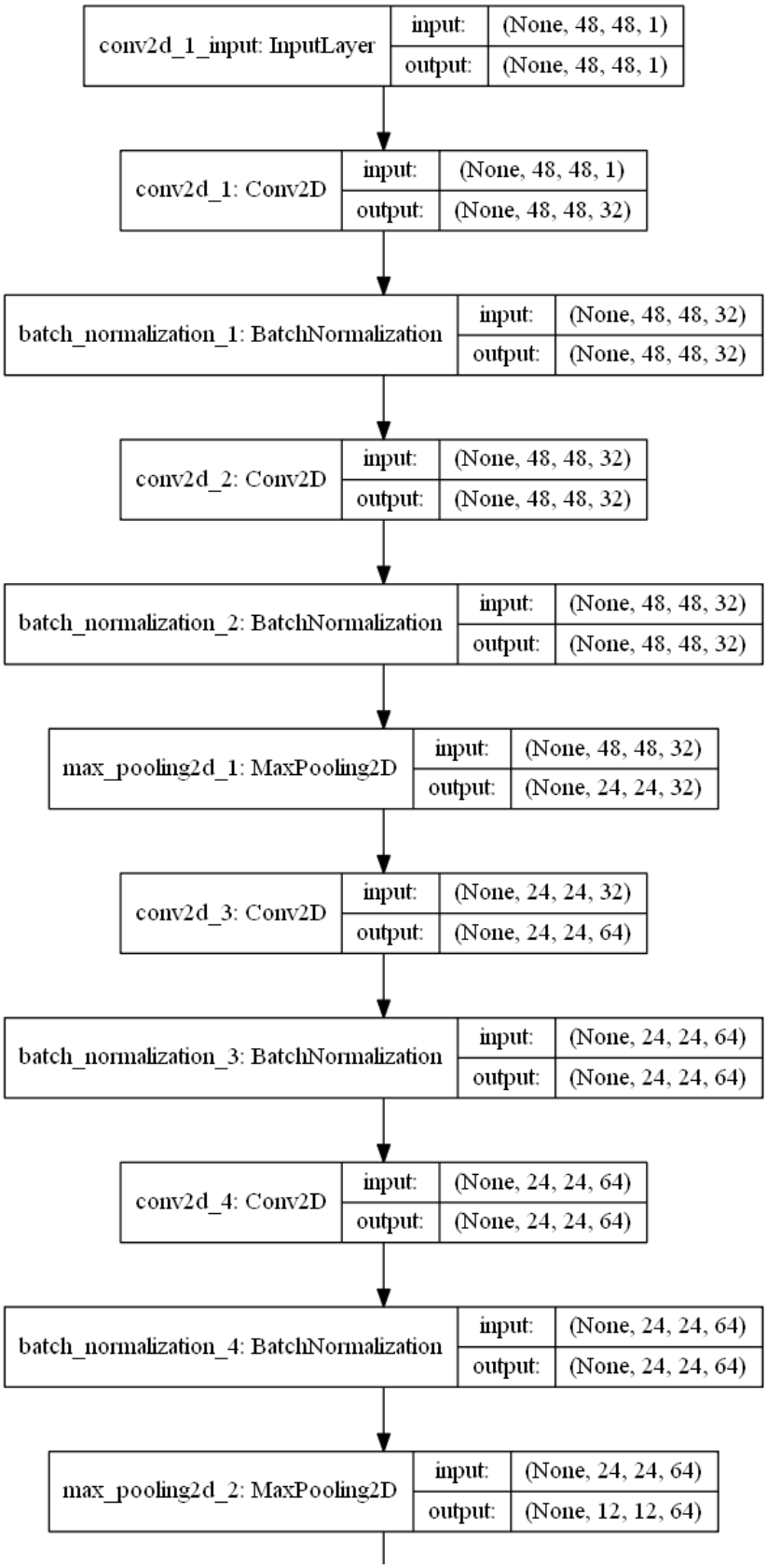
Le fichier docker-compose.yml que nous utilisons pour créer les images des composants.

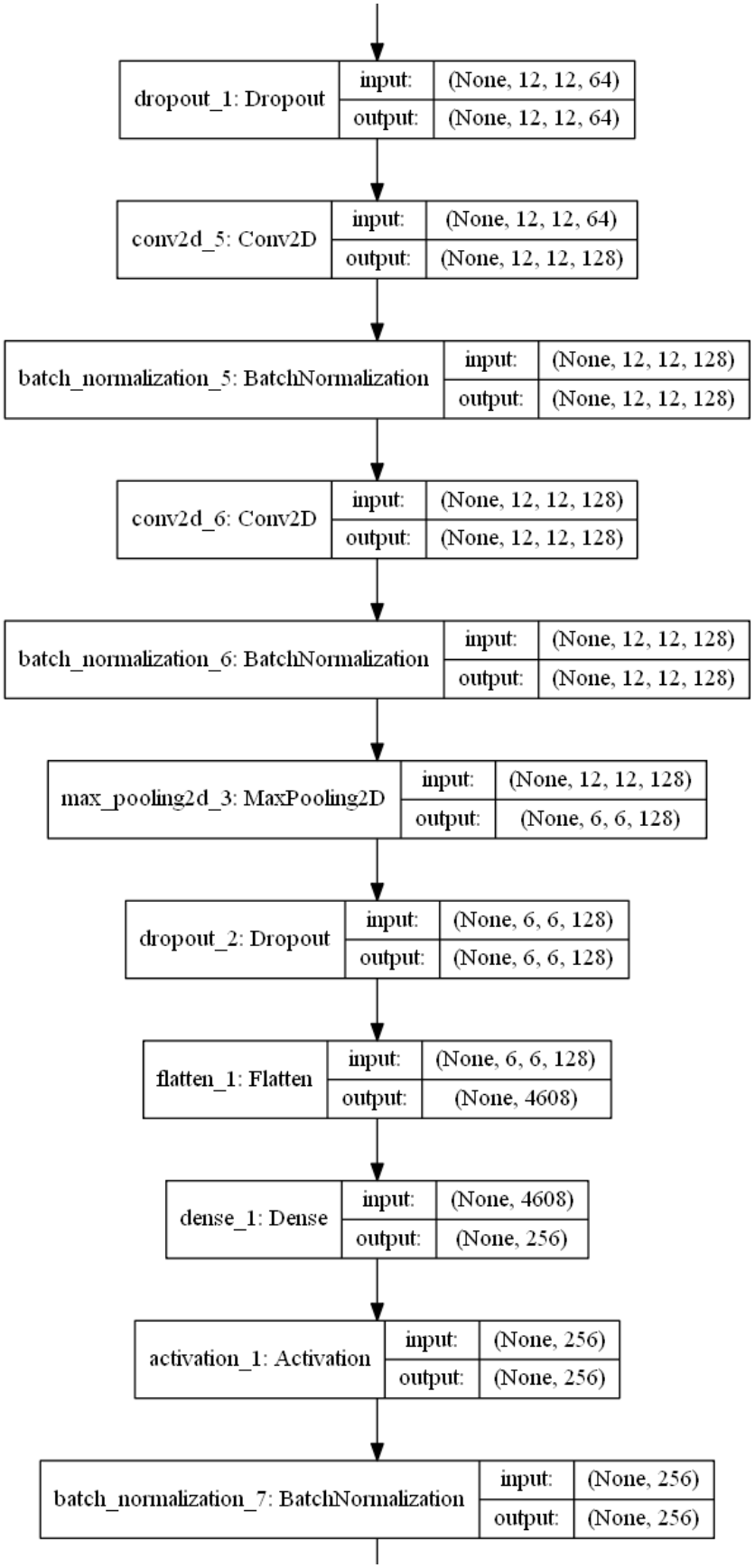
```yaml

  version: '3.8'
  services:
    emotion_recognition:
      build:
        context: ./emotion_recognition
        target: emotion_recognition-docker
      container_name: emotion_recognition
    text_to_speech:
        build:
          context: ./text_to_speech
          target: text_to_speech-docker
        container_name: text_to_speech
    speech_to_text:
      build:
        context: ./speech_to_text
        target: speech_to_text-docker
      container_name: speech_to_text
    face_recognizer:
      build:
        context: ./face_recognizer
        target: face_recognizer-docker
      container_name: face_recognizer

```

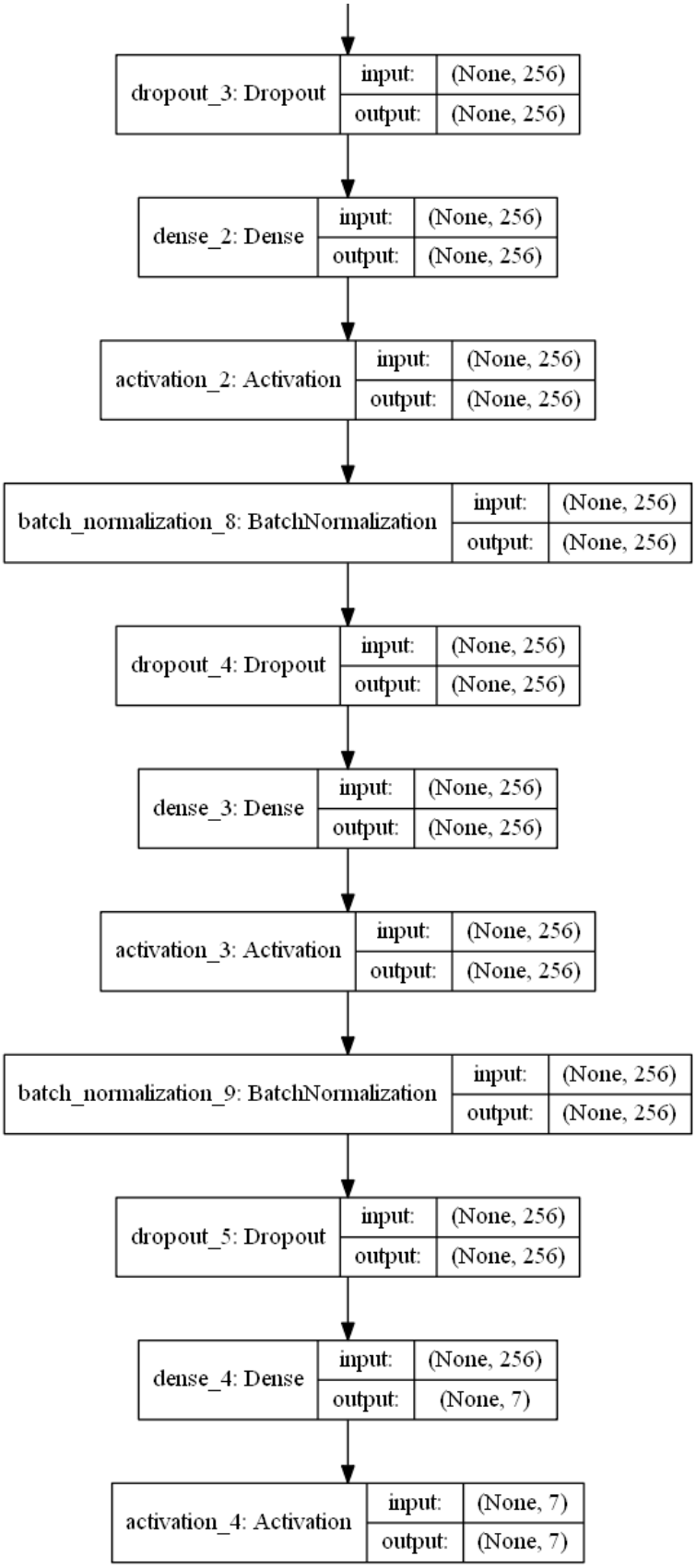
Le fichier Dockerfile du composant text_to_speech.

```dockerfile

# set base image (host OS)
FROM python:3.8 as text_to_speech-docker
# set the working directory in the container
WORKDIR /
# install dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt
# copy the content of the local src directory to the working directory
COPY * ./
# command to run on container start
ENTRYPOINT ["python","./__init__.py"]

```

9.3 Architecture du modèle




dropout_1: Dropout
input: (None, 12, 12, 64)
output: (None, 12, 12, 64)
conv2d_5: Conv2D
input: (None, 12, 12, 64)
output: (None, 12, 12, 128)
batch_normalization_5: BatchNormalization
input: (None, 12, 12, 128)
output: (None, 12, 12, 128)
conv2d_6: Conv2D
input: (None, 12, 12, 128)
output: (None, 12, 12, 128)
batch_normalization_6: BatchNormalization
input: (None, 12, 12, 128)
output: (None, 12, 12, 128)
max_pooling2d_3: MaxPooling2D
input: (None, 12, 12, 128)
output: (None, 6, 6, 128)
dropout_2: Dropout
input: (None, 6, 6, 128)
output: (None, 6, 6, 128)
flatten_1: Flatten
input: (None, 6, 6, 128)
output: (None, 4608)
dense_1: Dense
input: (None, 4608)
output: (None, 256)
activation_1: Activation
input: (None, 256)
output: (None, 256)
batch_normalization_7: BatchNormalization
input: (None, 256)
output: (None, 256)


dropout_3: Dropout
input: (None, 256)
output: (None, 256)
dense_2: Dense
input: (None, 256)
output: (None, 256)
activation_2: Activation
input: (None, 256)
output: (None, 256)
batch_normalization_8: BatchNormalization
input: (None, 256)
output: (None, 256)
dropout_4: Dropout
input: (None, 256)
output: (None, 256)
dense_3: Dense
input: (None, 256)
output: (None, 256)
activation_3: Activation
input: (None, 256)
output: (None, 256)
batch_normalization_9: BatchNormalization
input: (None, 256)
output: (None, 256)
dropout_5: Dropout
input: (None, 256)
output: (None, 256)
dense_4: Dense
input: (None, 256)
output: (None, 7)
activation_4: Activation
input: (None, 7)
output: (None, 7)

9.4 Code source du modèle

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Dropout, Flatten
from tensorflow.keras.layers import BatchNormalization, Conv2D, MaxPool2D

model = Sequential(
    [
        Conv2D(32, 3, input_shape=(48, 48, 1), padding='SAME', activation='relu'),
        BatchNormalization(),
        Conv2D(32, 3, padding='SAME',activation='relu'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2), strides=2),

        Conv2D(64, 3, padding='SAME',activation='relu'),
        BatchNormalization(),
        Conv2D(64, 3, padding='SAME',activation='relu'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2), strides=2),
        Dropout(0.25),

        Conv2D(128, 3, padding='SAME',activation='relu'),
        BatchNormalization(),
        Conv2D(128, 3, padding='SAME',activation='relu'),
        BatchNormalization(),
        MaxPool2D(pool_size=(2, 2), strides=2),
        Dropout(0.25),

        Flatten(),

        Dense(256,activation='relu'),
        BatchNormalization(),
        Dropout(0.5),

        Dense(256,activation='relu'),
        BatchNormalization(),
        Dropout(0.5),

        Dense(256,activation='relu'),
        BatchNormalization(),
        Dropout(0.5),

        Dense(7,activation='softmax')
    ]
)
```

9.5 Arbre de décision

Un extrait de l'arbre de décision.

```
{
    "tag": "start",
    "keywords":[],
    "text":[
        ["Bonjour","Salut","Coucou","Hi"],
        ["Comment puis-je vous aider ?","Avez-vous besoin de quelque chose ?"]
    ],
    "action":"listen",
    "context":"",
    "next":[
        {
            "tag": "start>google",
            "keywords":["chercher","google","rechercher"],
            "text":[
                ["Que voulez-vous chercher ?","Ok, qu'est-ce que je dois chercher ?"]
            ],
            "action":"google",
            "context":"",


            "next":[
                "start"
            ]
        }
    ]
}


```