Integration of an Artificial Intelligence Model into a Smartphone Flutter Application to Solve a Live Image Classification Problem

Sergio Tasso(✉), Simone Calò, Osvaldo Gervasi, and Damiano Perri

Department of Mathematics and Computer Science, University of Perugia, via Vanvitelli, 1, 06123 Perugia, Italy
{sergio.tasso,osvaldo.gervasi,damiano.perri}@unipg.it

Abstract. The object of study consists of solving a live image classification problem in the mobile environment and the subsequent implementation of mechanisms to integrate an image classification model, leveraging TensorFlow Lite, within a smartphone application built with Google's Flutter framework and written in Dart code. The functionality has been implemented within a cultural outreach application and allows the user, during an art visit, to use the camera on his or her device to frame the artwork and get its name on the screen in real time. Each captured frame is in fact sent as input to the classifier, which, after performing the classification, returns as output the associated label (the name of the artwork) that is finally displayed on the screen. After briefly introducing the preliminary knowledge necessary for the complete understanding of the object of study (implementation of the camera in the app and conversion of the classification model into Tensorflow Lite format), the tools used for the implementation of the functionality are described, which exploit the main plugin used, namely tflite_flutter. Finally, to ensure that the performance of the feature is optimal in devices, it is presented the concept of Isolate in the flutter framework and described the mechanisms for creating a new Isolate within which to perform the classification operation and how to communicate between them in order to display in the application user interface the output returned by the image classifier.

Keywords: Mobile Developing · Image Classification · Artificial Intelligence · Convolutional Neural Network

1 Introduction

The idea behind this study is to use technology intelligently to encourage the dissemination of art and culture. In fact, technology can be used not only to experiment with new artistic possibilities and forms, but also as a means of dissemination and a tool for popularising all those pre-existing traditional cultural and artistic forms [1,2].

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
O. Gervasi et al. (Eds.): ICCSA 2024 Workshops, LNCS 14815, pp. 39–55, 2024.
https://doi.org/10.1007/978-3-031-65154-0_3

Technologies using virtual and augmented reality quickly and affordably allow users to fully immerse themselves [3–6] in the works of art they are viewing in real time during an artistic visit. In fact, using an AR application on their smartphones, viewers can explore artworks in a three-dimensional way, often with interactive elements that enrich the experience, or by viewing information or trivia about the framed artwork in a contextual way [7]. The visit then becomes interactive, viewers are actively involved, and as they contemplate the artwork, they can view contextual information that enriches and embellishes the experience as if they were part of the artwork itself [8–10].

Another example are apps dedicated to art and culture [11], such as complementary museum apps, with which viewers can experience features such as camera recognition of artworks and view detailed information about the artworks, stories, and trivia about the artists on their smartphone screen, all in real time as they continue their art visit.

In other words, technology can thus be used to culturally enrich and supplement the art visit [12,13], in real time and without sacrificing contact with reality [14]. This is where the *Lions Umbria Art*[1] cultural outreach app and its live image classification feature, which is the subject of the following article, was born.

2 Mobile Developing

To implement this functionality, it is first necessary to create a flutter design of a smartphone app and create an image classifier.

2.1 Development of the Classification Model

A common Convolutional Neural Network can be used to develop an image classification model [15]. First, it is necessary to create an appropriate dataset with which to train the model later. In this case, the dataset consists of a set of images depicting some artistic paintings dating back to the early/late Middle Ages in the Italian region of Umbria.

For the definition of the Convolutional Neural Network, Tensorflow's Keras API[2] of the Python programming language and the *Sequential* model were used, to which the appropriate convolutional and downsampling layers were added [16]. It should be noted that the input model receives 3 maps of size 256×256 (one for each of the 3 RGB colors) whose values are normalised. This is an important step since, then, within the *dart* code of the application, the image captured by the camera must first also be normalised. In output, however, the model returns an array of decimal values, each of which expresses the probability of belonging to that specific class (each value is then between 0 and 1, and the length of the output array is equal to the number of classes with which the model is trained).

[1] https://www.lions108l.com/.
[2] https://keras.io/api/.

After compiling the model, we continue with its training from the dataset appropriately created earlier [17,18]. At the end of this phase, you have obtained a Keras model of image classification ready to make possible predictions.

Before it can be used in the mobile environment, the newly created Keras model must be converted to the Tensorflow Lite format [19].

2.2 Conversion of Image Classification Model to Tensorflow Lite Format

In fact, converting the model to the *.tflite* format is necessary because Tensorflow Lite[3] guarantees excellent model execution performance within mobile environments [20].

To convert a classification model, it is first necessary to identify the format with which it was stored. In this case, the newly created model is a Keras model, so the corresponding Python API for[4] conversion is used, as shown in Listing 1.1.

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

Listing 1.1. Conversion by *from_ keras_ model()*.

After storing the newly converted model with the appropriate function shown in Listing 1.2, we obtain the file *.tflite* that coincides with the classification model that will be used by the flutter application.

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

Listing 1.2. Storage of the *.tflite* model.

2.3 Creating a Flutter App that Uses the Camera

This Section briefly describes the creation of a simple Flutter[5] application that, when opened, opens the device's camera.

First, a Flutter project must be created [21]. In this case, the development environment is Windows 11 and the project was created via Android Studio, as described in depth in the official documentation[6].

Within the **directory *root*** of the **project**, the elements that are most involved are as follows:

- **directory *assets***
  - This directory is used to store all those assets that the application needs, such as static files like images, fonts, and JSON data. Usually the contents of the assets directory can be organised into subdirectories to have more sorting by file types.

[3] https://www.tensorflow.org/lite.
[4] https://www.tensorflow.org/lite/models/convert.
[5] https://docs.flutter.dev/.
[6] https://docs.flutter.dev/get-started/install/windows/mobile.

  - In this directory are stored:
    * *model.tflite*, which is the model of image classifications.
    * *labels.txt*, which is the file containing the list of labels involved in the classification problem (each line contains one label).
- **directory *lib***
  - It is the heart of a project developed with Flutter as it contains all the *.dart* files that allow the definition of the application's Dart code[7]. In fact, this directory contains the *main.dart* file with which you can launch the execution of the application. All classes, widgets and graphical objects will be defined within this directory. In other words, the source code written by the developer resides entirely in this folder.
- **file *pubspec.yaml***
  - The file *pubspec.yaml* is the second most important file within the project. It is located in the project's root directory and is used to manage the project's *dependencies*, define metadata and application assets. It has **3 key elements** of the *pubspec.yaml* file:
    * ***dependencies***: within this section are declared all the external packages that the application uses. Flutter packages can be obtained from the official website *pub.dev*[8], where examples, versions, and instructions for installation and use can be found for each package.
    * ***assets***: this section declares the resources used by the application, such as images, icons, fonts. Paths to subdirectories or files within the *assets* folder are then specified.

Presented in Listing 1.3, the first file involved is ***main.dart***, which contains the ***main()*** function, whose job is to launch the execution of the app.

In the context of Flutter, central is the concept of **widget**[9]. The **widgets**, in fact, are the **elements** of a Flutter **application**, and similar to the bricks forming a wall, the widgets are **aggregated inside each other** creating a **structure** called *Widget Tree* with which the user interface is identified. Thus, the design of the **user interface** in Flutter involves the **assembly** and/or creation of several **widgets**, building more complex widgets by combining many simpler ones. In Flutter, moreover, a **widget** represents an immutable **description** of the **user interface**: graphics, text, shapes, animations, and in general **any graphic object** that appears on the screen are created using **widgets**.

To actually define the content of a custom widget (i.e., a custom class that extends *StatelessWidget* or *StatefulWidget*), must be executed *override* of the inherited **build()** method, the return of which is what will be displayed. In fact, the ***build()*** method is responsible for building the widget and displaying it.

So in this case the largest container is the custom widget *MyFlutterApp* that is launched by the function *main()*. In fact, its method *build()* returns the widget *MaterialApp* which has as its child the custom widget *CameraScreen*, described in Sect. 3.2.

[7] https://dart.dev/guides/.

[8] At the following link https://pub.dev/packages the entire library of Dart packages that can be used with the Fluter framework can be consulted.

[9] https://docs.flutter.dev/.

```
import 'package:flutter/material.dart';

void main() async {
  runApp(const MyFlutterApp());
}

class MyFlutterApp extends StatelessWidget {
  const MyFlutterApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter App',
      debugShowCheckedModeBanner: false,
      theme: ThemeData(
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.green),
        useMaterial3: true,
      ),
      home: const CameraScreen(),
    );
  }
}
```

Listing 1.3. *main()* function and *MyFlutterApp* class.

Figure 1 shows the widget tree of the entire application.

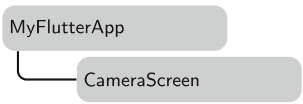

Fig. 1. Application Widget Tree.

3 Architecture of the System

In order for them to be effectively used by the classifier, frames captured by the camera must be converted into objects of **class *Image***[10]. In order to use the latter class, it is necessary to import the **package *image*** within the app.

3.1 Class *ImageUtils*

Within the class ***ImageUtils*** are then defined the **functions** that allow the conversion of **objects** of **class *CameraImage*** (i.e., frames captured by the ***camera***) to **objects** of the **class *Image***. Depending on the device used (Android or iOS), the *CameraImage* class object coincides with two different types of images: bgra8888 for iOS and YUV420 for Android.

Therefore, the functions to be implemented are as follows:

- *convertBGRA8888ToImage()*
- *convertYUV420ToImage()*

[10] https://pub.dev/documentation/image/latest/.

3.2 Class *CameraScreen*

This custom widget takes care of camera management through the use of the package ***camera***[11].

In fact, the *camera* package was imported to be able to use the ***CameraPreview*** widget, which allows the display of a screen where the camera is active. The two main objects that were used are the class ***controller*** and the function ***availableCameras()***, which is needed to get the list of all the cameras in the device. In addition, the following functions were also used:

- ***getMinZoomLevel()*** and ***getMaxZoomLevel()***, which allow capturing the minimum and maximum zoom value of the camera used, respectively.
- ***getMinExposureOffset()*** and ***getMaxExposureOffset()***, which respectively allow capturing the minimum and maximum exposure value of the camera used.
- ***takePicture()***, which allows you to take a picture.

The screen interface, whose widget tree is shown in Fig. 2, is made possible from the widget ***Stack***, which allows multiple widgets to be stacked on top of each other.

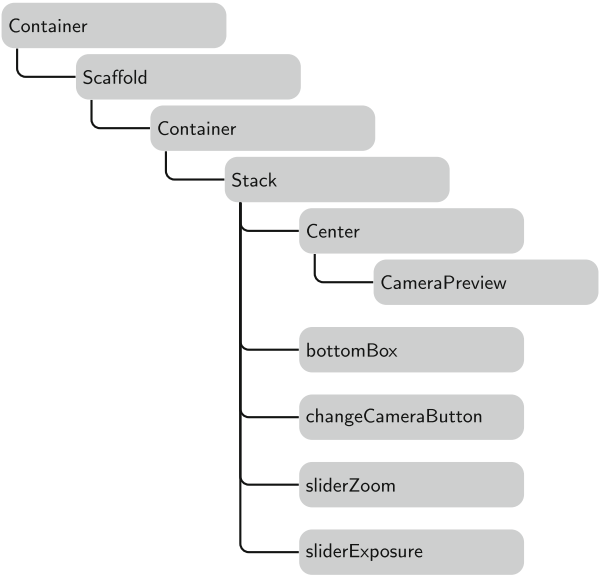

Fig. 2. *CameraScreen* Widget Tree.

[11] https://pub.dev/documentation/camera/latest/.

In fact, the following custom widgets are placed above the main widget:

- widget returned by the function ***bottomBox()***, positioned **at the bottom of the screen** and within which appear the **information** obtained through the **classification** of the **image framed** by the user.
- widget returned by the function ***sliderZoom()***, which realizes a slider to increase or decrease the zoom.
  - The slider is made with the widget ***Slider***, at whose **field** ***onChanged*** is assigned an **anonymous function** that **updates** the **current zoom value** with the one set by the user by calling the ***setState()*** function to update the widget display.
- widget returned by the function ***sliderExposure()***, which makes a slider to increase or decrease the display.
  - The slider is made with the widget ***Slider***, at whose **field** ***onChanged*** is assigned an **anonymous function** that **updates** the **current display value** with the one set by the user by calling the function ***setState()*** to update the display of the widget.

The function *initCamera*

The function ***initCamera()***, shown in Listing 1.4 has the simple task of initializing the widget ***CameraView***, by performing the following operations:

- **captures** all the **cameras** available in the **device** via the ***availableCameras()*** function of the **package camera**, assigning them to a list of class ***List<CameraDescription>***.
- calls the function ***initController()***, described in the next Sect. 3.2.

```
void initCamera() async {
    if (camerasList.isEmpty) {
      camerasList = await availableCameras();
    }
    for (var i = 0; i < camerasList.length; i++) {
      if (listaCamere[i].lensDirection == widget.initialCameraLensDirection)
          {
        currentCameraIndex = i;
        break;
      }
    }
    if (currentCameraIndex != -1) {
      initController();
    }
  }
```

Listing 1.4. *initCamera* function of *CameraScreen*.

It should be noted that the function ***initCamera()*** is **recalled** by the function ***initState()*** of the widget ***CameraScreen***.

The function *initController*

The function ***initController()*** deals with the **creation** of the **controller** class ***CameraController***, to which are given as input, for example, one of the **cameras** class ***CameraDescription*** captured earlier in the *initCamera()* function, or the resolution with which it will be used and disabling audio while using the camera as it is not necessary.

After **istanced** the **controller**, you perform the following operations, as shown in the code excerpt contained in Listing 1.5:

1. You start the procedure that allows the execution of the function ***imageAnalysis()*** with each captured frame.
2. You **create** the class object ***ImageClassificationHelper*** to create the classifier.
3. You execute ***initHelper()*** to actually load the classification model.
4. The operations of capturing the maximum and mini zoom and exposure values with the appropriate functions provided by **package *camera*** are executed.

```
Future initController() async {
        final camera = cameraList[currentCamera];
        controller = CameraController(
        camera,
        ResolutionPreset.high,
        enableAudio: false,
        imageFormatGroup: Platform.isAndroid ? ImageFormatGroup.yuv420 :
            ImageFormatGroup.bgra8888,
        );
        controller?.initialize().then((_) {

                //1
                controller!.startImageStream(imageAnalysis);


                //2
                imageClassificationHelper = ImageClassificationHelper();

                //3
                imageClassificationHelper.initHelper();

                ...
        });
}
```

Listing 1.5. Excerpt from the function *initController* of *CameraScreen*.

The function *imageAnalysis()*

The ***imageAnalysis()*** function, presented in Listing 1.6, is the most important function of the *CameraScreen* widget since it is executed for each frame that the *CameraView* widget captures when the camera is active and if the analysis operation is not currently active. The function ***imageAnalysis()*** calls the function ***inferenceCameraFrame()*** on the object of class ***ImageClassificationHelper***, passing to it as the **parameter** the **frame** of class ***CameraImage***.

```
Future<void> imageAnalysis(CameraImage cameraImage) async {
        if (_isProcessing) {
                return;
        }
        _isProcessing = true;

        try {
                classification = await imageClassificationHelper.
                    inferenceCameraFrame(cameraImage);
        } catch (e) {
                print(
                ''Late Inizialization Exception.  imageClassificationHelper
                    not yet initialized.'');
        }
```

```
        _isProcessing = false;
        if (mounted) {
                setState(() {});
        }
}
```

Listing 1.6. *imageAnalysis* function of *CameraScreen*.

3.3 Creation of a Second *Isolate*

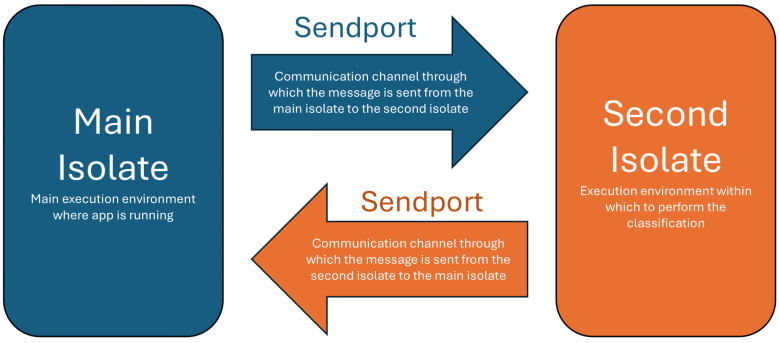

Fig. 3. Scheme of operation of the *Isolates* and the communication mechanism between them.

***Isolates* e communication mechanism**
The classification operation takes a lot of resources to perform, so it makes sense to separate it from the one environment currently running that manages the entire application. The *main()* method launches the execution of the app within the ***MainIsolate***[12], which basically consists of an isolated execution environment similar to a java thread. To separate the classification operation from the rest, one must then initialize a new *Isolate*, within which one must execute everything necessary to classify the frame captured by the camera.

In order then for the output of the classifier to be shown on the screen at the end of the classification operation, it is necessary for the second *Isolate* to communicate the label to the *MainIsolate*. As shown schematically in Fig. 3, one must then implement the unique **mechanism** of **communication between *Isolates*** that initiates via **message exchange**. In particular, the main objects with which a communication channel is realized are *ReceivePort* and *SendPort*:

– ***ReceivePort*** is the object from which an *Isolate* receives messages, listening to the corresponding *SendPort*.

[12] https://docs.flutter.dev/perf/isolates.

- ***SendPort***, on the other hand, represents the actual communication channel and is created with the *sendPort()* method of ***ReceivePort***.
  - A ***SendPort*** is then always instantiated starting with the creation of a ***ReceivePort*** via the *send()* method.

Therefore, the creation of the class ***IsolateInference*** is indispensable.

***IsolateInference* class**
The class ***IsolateInference*** is basically identified with the second *Isolate* within which to perform the classification operation and has associated **3 fields** that are extremely important:

- the **2 objects** of class ***ReceivePort*** and ***SendPort*** for the **communication** with the *MainIsolate*.
- the **object** class ***Isolate*** that coincides with the actual **second execution environment**.

In addition, it is essential to define the following **functions**:

1. ***start()***
   - This function has the simple of:
     (a) **create** the **istance** of class ***Isolate*** with the appropriate function ***spawn()***, which allows the creation of the second execution environment.
     (b) listens to the second *Isolate* to receive the message from the *MainIsolate*.
2. ***close()***
   - This function has the simple of:
     (a) **destroy** the **istance** of class *Isolate* with the appropriate function ***kill()*** called on the class object *Isolate*.
     (b) **close** also the **communication channel** by means of the ***close()*** function called on the ***ReceivePort***.
3. ***entryPoint()***
   - This function, shown in Listing 1.7, is executed whenever the ***Isolate*** receives a message from the ***MainIsolate*** and is responsible for the actual classification of the frame. The ***message*** that it receives is the ***InferenceModel*** class object[13] which is appropriately **created with each frame captured by the camera** and which contains everything needed to perform the classification.
     (a) After converting the *CameraFrame* to an object of class *Image* and appropriately generating the 3-dimensional matrix (remember that, in the example, the model receives a matrix with normalised values, thus divided by 255) and appropriately setting the *shape* of the model input and output, the function loads the classification model and performs the classification, obtaining the output.

[13] The class ***InferenceModel*** is described in more detail in Sect. 3.4.

(b) Finally, after appropriately transforming the output of the model into a map of the type ***label: value*** (where *value* is a decimal number from 0 to 1 expressing the probability of belonging to that class), the function sends the result to the *MainIsolate* exploiting the *SendPort* associated with the *InferenceModel*.

```
static void entryPoint(SendPort sendPort) async {
    final port = ReceivePort();
    sendPort.send(port.sendPort);

    await for (final InferenceModel isolateModel in port) {
      image_lib.Image? img;

      img = ImageUtils.convertCameraImage(isolateModel.cameraImage!);

      image_lib.Image imageInput = image_lib.copyResize(
        img!,
        width: isolateModel.inputShape[1],
        height: isolateModel.inputShape[2],
      );

      imageInput = image_lib.copyRotate(imageInput, angle: 90);

      final imageMatrix = List.generate(
        imageInput.height,
        (y) => List.generate(
          imageInput.width,
          (x) {
            final pixel = imageInput.getPixel(x, y);
            return [pixel.r / 255, pixel.g / 255, pixel.b / 255];
          },
        ),
      );

      final input = [imageMatrix];

      final output = [List<double>.filled(isolateModel.outputShape[1], 0.0)];

      Interpreter interpreter = Interpreter.fromAddress(isolateModel.
          interpreterAddress);
      interpreter.run(input, output);

      final result = output.first;

      double maxScore = result.reduce((a, b) => a + b);

      var classification = <String, double>{};
      for (var i = 0; i < result.length; i++) {
        if (result[i] != 0) {
          classification[isolateModel.labels[i]] = result[i].toDouble() /
              maxScore.toDouble();
        }
      }
      isolateModel.responsePort.send(classification);
    }
  }
```

Listing 1.7. *entryPoint* function of *ImageClassificationHelper*.

3.4 *InferenceModel* Class

The class ***InferenceModel*** has the sole purpose of enabling the **communication** between the *MainIsolate* and the second *Isolate*. As already specified, the

MainIsolate class object is in fact the **content** of the **message** that the *MainIsolate* sends to the second *Isolate* whenever the device's camera captures a frame. This class, in fact, contains only a **set** of **fields necessary** for **classification**:

- class object *CameraImage* captured by the camera.
- object of class *Image*.
- integer value specifying the address of class object *Interpreter* (the classifier).
- list of type *List<String>* that contains the labels involved in the classification.
- list of type *List<int>* that contains the *shape* of the classifier input (in this case it is [1, 256, 256, 3], i.e., a vector containing 3 256×256 matrices).
- list of type *List<int>* that contains the *shape* of the classifier's output (in this case it is [1, 4], i.e., a vector containing 4 decimal values from 0 to 1, each associated with its respective label).
- object of class *SendPort* for communication between the *MainIsolate* and the second *Isolate*.

3.5 *ImageClassificationHelper* Class

Within this class are defined the **functions** that deal with the **management** of the **model *.tflite***, among which we emphasize:

- ***loadModel()***
  - It is responsible for loading the model (***.tflite*** stored in the **folder *assets***) into the class variable ***Interpreter*** (the use of which is allowed through the package ***tflite_flutter***[14]).
- ***loadLabels()***
  - It provides the loading of labels (*labels*) stored in the corresponding text file *.txt*.
- ***initHelper()***
  - It takes care of initiating the procedures for the creation of the second *Isolate*, in particular it provides:
    1. create the class object ***IsolateInference***.
    2. call the function ***start()*** on the newly created class object ***IsolateInference***, which, as already described in Sect. 3.3, listens for the message containing the class object ***InferenceModel***.
- ***inferenceCameraFrame()***
  - As visible in Listing 1.8, it takes care of creating the class object ***InferenceModel*** which is the object that the *MainIsolate* exchanges in the communication channel with the other *Isolate*. Specifically, it performs the following operations:
    1. first creates the class object ***ReceivePort***, the only means of communication (along with ***SendPort***) between ***Isolate***.

[14] https://www.tensorflow.org/lite.

2. connects the ***SendPort*** class objects of the ***MainIsolate*** and the second ***Isolate***.
3. opens a *listener* such that it waits for the **response** of the second *Isolate*.
4. returns as a result the response sent by the second *Isolate* to the *MainIsolate*, that is, the label that the classifier assigned to the frame captured by the device camera.

– ***close()***
  - Instead, it deals with the closure of the second *Isolate*, thus executing the function ***close()*** on the class object ***IsolateInference***.
    * It is pointed out that the function ***close()*** of the class *ImageClassificationHelper* is ***dispose()*** by the function ***dispose()*** of the widget ***CameraScreen***. When the ***CameraScreen*** widget is closed, the second execution space reserved for the classification operation is then also closed.

```
Future<Map<String, double>> inferenceCameraFrame(CameraImage cameraImage)
    async {


    var inferenceModel = InferenceModel(cameraImage, null, interpreter.
        address, labels, inputTensor.shape, outputTensor.shape);

    ReceivePort responsePort = ReceivePort();
    isolateInference.sendPort.send(inferenceModel..responsePort =
        responsePort.sendPort);

    var results = await responsePort.first;
    return results;
  }
```

Listing 1.8. *inferenceCameraFrame* function of *ImageClassificationHelper*.

4 Feature Applications: *Lions Umbria Art*, Cultural Outreach App

The functionality analysed and developed in this paper was realised for the cultural outreach application *Lions Umbria Art*. As visible in Fig. 4, the user can open the camera of his or her device whenever he or she wants to take pictures of the works he or she is viewing during the art visit. Taking a cue from cultural museum applications and simple basic virtual reality features, the user, while taking the photo, has the possibility in real time to read on the screen some information associated with the work he or she is framing, such as its name and author. In this way, as anticipated within the Introduction, the user actively participates in the artistic visit, intelligently using the smartphone enriching the experience without, however, sacrificing contact with reality.



Fig. 4. Screenshot that allows the user to read the associated information in real time as they are framing the work with their smartphone.

5 Conclusions

The use of development frameworks such as Flutter and programming languages highly expressive such as Dart has made it possible to ascertain the feasibility practicality of developing sufficiently complex and feature-rich software integrating an artificial intelligence model.

First, it was indeed possible to see the enormous advantage of writing of the only source code, thus halving the entire app development. The management and integration of the classification model is uniquely implementable for both target platforms (iOS and Android), with rare exceptions of single management (such as the conversion of the *CameraImage* object, which depending on the device is in bgra8888 format for iOS and YUV420 for Android and thus requires writing 2 conversion functions).

Second, the excellent execution performance of the classification model could be ascertained. Indeed, the separate handling of the operation within the second *Isolate* ensures that there is no loss of smoothness in the user interface. In addition, the Flutter framework together with the Tensorflow Lite format guarantees excellent performance, especially from the point of view of classification timing. In fact, the latter is substantially immediate, ensuring maximum responsiveness and ease of use by the user. It is also pointed out that the quality of the classification is obviously not taken into account, since the latter depends on the goodness of the construction of the classification model and the appropriate training of the latter, providing an adequate dataset in order to avoid overfitting or underfitting. Also significant is the mechanism for handling Isolates provided by the Flutter framework, which allows the application (both the simplified version just described and *Lions Umbria Art*) to be extremely responsive, both on devices equipped with high-end hardware and software optimizations such as Google Pixel 8, and on more modest devices such as Nothing Phone (1). Finally, also not to be forgotten is the final size of the application, which, despite the packages used and the presence of a classification model, is nevertheless sufficiently contained without therefore being a real disadvantage for users who will wish to download it to their smartphones.

Acronyms

The following acronyms are used in this manuscript:

API	Application Programming Interface
AR	Augmented Reality
iOS	iPhone Operating System
JS	JavaScript
JSON	JavaScript Object Notation
RGB	Red Green e Blue

References

1. Pellas, N., Dengel, A., Christopoulos, A.: A scoping review of immersive virtual reality in STEM education. IEEE Trans. Learn. Technol. **13**(4), pp. 748–761 (2020). https://doi.org/10.1109/TLT.2020.3019405
2. Buragohain, D., Deng, C., Sharma, A., Chaudhary, S.: The impact of immersive learning on teacher effectiveness: a systematic study. IEEE Access **12**, 35924–35933 (2024). https://doi.org/10.1109/ACCESS.2024.3373541
3. Zhang, X.: The college english teaching reform supported by multimedia teaching technology and immersive virtual reality technology. In: 2019 International Conference on Virtual Reality and Intelligent Systems (ICVRIS), pp. 77–80 (2019). https://doi.org/10.1109/ICVRIS.2019.00028
4. Han, H., Lu, A., Wells, U.: Under the movement of head: evaluating visual attention in immersive virtual reality environment. In: 2017 International Conference on Virtual Reality and Visualization (ICVRV), pp. 294–295 (2017). https://doi.org/10.1109/ICVRV.2017.00067
5. Perri, D., Fortunelli, M., Simonetti, M., Magni, R., Carloni, J., Gervasi, O.: Rapid prototyping of virtual reality cognitive exercises in a tele–rehabilitation context. In: Electron. (Switz.) **10**(4) (2021). Cited by: 6; All Open Access, Gold Open Access, Green Open Access, pp. 1–16. https://doi.org/10.3390/electronics10040457
6. Rajasagi, D.P.: Immersive health data visualization in virtual reality. In: 2023 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW), pp. 971–972 (2023). https://doi.org/10.1109/VRW58643.2023.00328
7. Gervasi, O., Perri, D., Simonetti, M.: Empowering knowledge with virtual and augmented reality. IEEE Access **11**, 144649–144662 (2023). https://doi.org/10.1109/ACCESS.2023.3342116
8. Qu, M., Sun, Y., Feng, Y.: Digital media and VR art creation for metaverse. In: 2022 2nd Asia Conference on Information Engineering (ACIE), pp. 48–51 (2022). https://doi.org/10.1109/ACIE55485.2022.00018
9. Gervasi, O., Perri, D., Simonetti, M., Tasso, S.: Strategies for the digitalization of cultural heritage. In: Gervasi, O., Murgante, B., Misra, S., Rocha, A.M.A.C., Garau, C. (eds.) Computational Science and Its Applications – ICCSA 2022 Workshops. ICCSA 2022. LNCS, vol. 13382. Springer, Cham. https://doi.org/10.1007/978-3-031-10592-0_35
10. Hong, R., He, H.: Interference and consultation in virtual public space: the practice of intermedia art in metaverse. In: 2021 17th International Conference on Mobility, Sensing and Networking (MSN), pp. 792–797 (2021). https://doi.org/10.1109/MSN53354.2021.00124
11. Perri, D., Simonetti, M., Tasso, S., Gervasi, O.: Open metaverse with open software. In: Gervasi, O., et al. Computational Science and Its Applications – ICCSA 2023 Workshops. ICCSA 2023. LNCS, vol. 14111. Springer, Cham (2023). https://doi.org/10.1007/978-3-031-37126-4_37
12. Li, X., Li, K., Zhao, H., Lou, Y., Geng, X., Huang, W.: Virtual reality custom interactive art space. In: 2020 International Conference on Virtual Reality and Visualization (ICVRV), pp. 365–366 (2020). https://doi.org/10.1109/ICVRV51359.2020.00104
13. Liu, Q.: The virtual reality technology in art design. In: 2012 2nd International Conference on Consumer Electronics, Communications and Networks (CECNet), pp. 2226–2228 (2012). https://doi.org/10.1109/CECNet.2012.6201444

14. Wu, X., Yuan, X.: Innovation system of film art creation based on virtual reality technology. In: 2021 6th International Conference on Smart Grid and Electrical Automation (ICSGEA), pp. 231–234 (2021). https://doi.org/10.1109/ICSGEA53208.2021.00057
15. Kumar, G., Kumar, P., Kumar, D.: Brain tumor detection using convolutional neural network. In: 2021 IEEE International Conference on Mobile Networks and Wireless Communications (ICMNWC), pp. 1–6 (2021). https://doi.org/10.1109/ICMNWC52512.2021.9688460
16. Perri, D., Simonetti, M., Lombardi, A., Faginas-Lago, N., Gervasi, O.: Binary classification of proteins by a machine learning approach. In: Gervasi, O., et al (eds.). Computational Science and Its Applications – ICCSA 2020. ICCSA 2020, vol. 12255. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-58820-5_41
17. Tseng, C.C., Lee, S.L.: Design of digital differentiator using supervised learning on keras framework. In: 2019 IEEE 8th Global Conference on Consumer Electronics (GCCE), pp. 162–163 (2019). https://doi.org/10.1109/GCCE46687.2019.9014634
18. Joshi, S., Owens, J.A., Shah, S., Munasinghe, T.: Analysis of preprocessing techniques, keras tuner, and transfer learning on cloud street image data. In: 2021 IEEE International Conference on Big Data (Big Data), pp. 4165–4168 (2021). https://doi.org/10.1109/BigData52589.2021.9671878
19. Konaite, M., et al.: Smart hat for the blind with real-time object detection using raspberry pi and tensor- flow lite. In: Proceedings of the International Conference on Artificial Intelligence and Its Applications. icARTi '21. Virtual Event, Mauritius: Association for Computing Machinery, pp. 1–6 (2021). isbn: 9781450385756. https://doi.org/10.1145/3487923.3487929
20. Ahmed, S., Bons, M.: Edge computed NILM: a phone-based implementation using MobileNet compressed by Tensorflow Lite. In: Proceedings of the 5th International Workshop on Non-Intrusive Load Monitoring. NILM'20. Virtual Event, Japan: Association for Computing Machinery, 2020, pp. 44–48. isbn: 9781450381918. https://doi.org/10.1145/3427771.3427852. url: https://doi.org/10.1145/3427771.3427852
21. Lohani, D.: Taking Flutter to the Web: Learn How to Build Crossplatform UIs For Web and Mobile Platforms Using Flutter For Web. Packt Publishing (2022). isbn: 1801817715