Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

DeepReality: An open source framework to develop AI-based augmented reality applications

Roberto Pierdicca [a,*], Flavio Tonetto [b], Marina Paolanti [c], Marco Mameli [d], Riccardo Rosati [d], Primo Zingaretti [d]

[a] *Dipartimento di Ingegneria Civile, Edile e dell'Architettura (DICEA), Università Politecnica delle Marche, Via Brecce Bianche, 12, 60131 Ancona, Italy*
[b] *Sinergia, Viale Goffredo Mameli, 44, Pesaro, 61121, Italy*
[c] *Department of Political Sciences, Communication and International Relations, University of Macerata, Via Don Minzoni 22A, Macerata, 62100, Italy*
[d] *VRAI - Vision Robotics and Artificial Intelligence Lab, Dipartimento di Ingegneria dell'Informazione, Università Politecnica delle Marche, via Brecce Bianche 12, Ancona, 60131, Italy*

ARTICLE INFO

Keywords:
Deep learning
Augmented reality
Unity
AR foundation
Barracuda

ABSTRACT

Augmented reality (AR) and Artificial Intelligence (AI) are technologies pioneers in innovation and alteration in several domains. AR allows the creation of an entirely new and interactive experience for users. However, there are several drawbacks in developing AR applications, such as the marker identification process and the creation of content itself. These are very time-consuming procedures and require ad-hoc development. The advantages of using AI to solve AR limitations have recently been explored in literature. Motivated by these findings, in this paper it is proposed DeepReality, a software toolkit plug-in for Unity 3D. It is conceived for allowing developers to integrate any Deep Learning (DL) models into Unity, through AR Foundation and Barracuda inference engine. DeepReality is aimed at simplifying and streamlining the usage of DL models in conjunction with AR. As such, users skilled in Unity and DL can easily create mobile applications (iOS and Android) to: extract visual features of real-world objects (framed with the device camera) via DL; Show on-screen content on top of those real-world objects, via AR. DeepReality performs object semantic processing within the scene, and extended semantic effects for incongruent objects, overcoming the environmental tracking, which is feature-based. In order to test DeepReality usability, experiments have been performed on the execution time and memory usage data, demonstrating the feasibility and possibility of integrating and using DNNs models in mobile applications for AR. The complexity analysis confirms that DeepReality can be completely executed on mobile devices. DeepReality is also open-source and it is freely available in the Unity asset store. By fostering accessible AI-AR integration, DeepReality addresses key shortcomings in existing approaches, encapsulating contributions such as versatile DL integration, open-source accessibility, operational validation, and comprehensive metrics analysis. DeepReality empowers developers to transcend boundaries, enriching AR applications with AI's transformative potential. Our proposed framework fosters benchmarking, comparison, and a future harmonised by AR-AI synergy.

1. Introduction

The increasing number of emerging immersive technologies are being considered to improve and enhance procedural execution of applications in many domains (e.g., manufacturing, education, retail, tourism, etc.) (Stanney et al., 2022). Thanks to technological advances, companies and research centres are investing and acting to spread these applications, whilst the end users are growing their interest given the irrefutable usefulness to facilitate daily tasks (Sereno, Wang, Besançon, McGuffin, & Isenberg, 2020). The literature assesses the adoption of virtual, mixed, and augmented reality (VR/MR/AR) solutions, and in particular the coalescing of all three in eXtended Reality (XR), as consolidated ways to convey contents in different scenarios (Banfi, Brumana, & Stanga, 2019; Vaswani et al., 2017).

AR is a notable example of such technologies, which has grown up as one of the most prominent in the present era. AR can be described as an extended variant of the physical world overlaid with digital contents bridging the real and virtual environments (Ghasemi, Jeong, Choi, Park, & Lee, 2022). Up to few years ago, the exploitation of AR had the

* Correspondence to: Via Brecce Bianche 12, Ancona, 60131, Italy.
E-mail addresses: r.pierdicca@staff.univpm.it (R. Pierdicca), ftonetto@sinergia.it (F. Tonetto), marina.paolanti@unimc.it (M. Paolanti), m.mameli@pm.univpm.it.it (M. Mameli), r.rosati@pm.univpm.it.it (R. Rosati), p.zingaretti@univpm.it (P. Zingaretti).

https://doi.org/10.1016/j.eswa.2024.123530
Received 6 July 2023; Received in revised form 13 February 2024; Accepted 16 February 2024
Available online 28 February 2024

main aim of providing the so-called "wow effect" to the public. Nevertheless, the more the technology becomes widespread, the more the expectation increases. Broadly speaking, the technological limitations hampering the diffusion of AR applications are going to be overcome, thanks to the growing computational capabilities of smartphones and tablets. However, there are still many challenging issues that are waiting to be discovered and improved in AR-related fields. One of the major difficulties is that there are several AR markers on the market, each with its own unique encoded information algorithm (Nguyen, Tran, Le, & Yan, 2017). They usually require the users to modify their original material contents in some way, either partially or completely. Another problem is the marker identification process, which utilises the standard computer vision-based feature extraction approaches, such as scale-invariant feature transformations or histograms of oriented gradients (Lowe, 1999), for classification tasks. These mathematical methods are vulnerable to unanticipated real-world lighting, marker orientation, and unexpected noises (Gammeter, Gassmann, Bossard, Quack, & Van Gool, 2010).

Another great bottleneck, identified in the literature (see Section 2) is the creation of contents itself. In fact, it is a very time-consuming process and requires ad-hoc development. In other words, to create an AR experience, it is necessary to choose the points of interest, create digital content, and then develop the mobile application. If this is affordable for very specific requirements or tasks, it would be impracticable for a real generalisation of the process. Many studies have shown that virtual contents, which are displayed as additional layers, must be conveyed according to both the choices of the domain experts and according to the preferences of the users (Naspetti et al., 2016; Pierdicca et al., 2018), ad-hoc generating suggestions (Moreno-Armendáriz et al., 2022), and supporting the navigation (Lin, Chung, Chou, Chen, & Tsai, 2018). This aspect represents a great limitation: it requires that the contents displayed in the application are created a priori by using software external to the application. The request for working outside of the application context by inexperienced users in content creation for AR applications has hampered its horizontal exploitation in the broader market.

Artificial Intelligence (AI) algorithms, particularly Deep learning (DL) approaches, demonstrate outstanding performance in several applications, especially image recognition through Convolutional Neural Networks (CNNs). Recently, research papers propose the use of CNNs as useful tools to overcome the standard computer vision difficulties in the AR marker identification process (Lalonde, 2018; Park, Kim, Choi, & Lee, 2020; Rao, Qiao, Ren, Wang, & Du, 2017). By the availability of large-scale datasets and computational power, DL approaches have been used in visual recognition tasks (Lampropoulos, Keramopoulos, & Diamantaras, 2020; Zhao, Zheng, Xu, & Wu, 2019). Therefore, to facilitate the development of AR experiences by overwhelming the above-mentioned limitations, the following assumption has been conceived: albeit being two distinct technologies, AR and AI are disruptive technologies that can be combined to obtain unique and immersive experiences. Starting from these premises, this work focuses on integrating DL models into the pipeline of AR development. Our approach resulted in the integration of pre-trained neural networks to increase the performances of both object detection and classification in AR environment. DeepReality, a proof-of-concept system that allows developers to integrate DL models into Unity3D, is developed to demonstrate the feasibility of our methodology. At a glance, DeepReality is aimed at simplifying and streamlining the usage of DL models in conjunction with AR. As such, users skilled in Unity and DL can easily create mobile applications (iOS and Android) to:

- Extract visual feature of real-world objects (framed with the device camera) via DL;
- Show on screen content on top of those real-world objects, via AR.

Due to its data-driven nature, DeepReality can be generalised to any DL model that the user might integrate within the AR application, overcoming the current limitation given by the software development kit (SDK) providers. The developed software toolkit plugin for Unity 3D is distributed in open-source mode and documented with the sequences and classes necessary for its implementation. Tests were conducted with several off-the-shelves devices to prove that the framework can be adopted in operational environments. Besides, important metrics and statistics have been collected, in order to highlight pros and cons of the proposed approach. The main concept is schematically depicted in Fig. 1, which highlights the important contributions of this research. In summary, this paper, aside from extending the system and the analysis presented in Pierdicca, Tonetto, Mameli, Rosati, and Zingaretti (2022), aims to fulfil the gap in AR applications development by (i) the developing and deploying of a Unity plug-in to enable the combination of AI in AR mobile Apps, ii) the integration of pre-trained Deep Neural Networks (DNNs) models within Unity, which will serve as a benchmark for future comparison and implementation, iii) a real-environment test to demonstrate the effectiveness of our approach and iv) the definition of a lower bound computational capacity of a mobile device to support the execution of the neural network. The source code and demos can be found at https://github.com/SinergiaGit/DeepReality and https://assetstore.unity.com/packages/tools/integration/deepreality-201076.

DeepReality stands as a technological achievement, powered by its robust data-driven architecture. Its defining strength lies in the seamless integration of a diverse range of deep learning models into augmented reality (AR) applications. This innovative approach allows developers to overcome the constraints of conventional software development kits (SDKs), enabling them to craft AR experiences enriched with a versatile palette of deep learning capabilities. A pivotal stride in DeepReality's journey is the creation of an open-source Unity 3D plugin, meticulously designed to amplify accessibility for developers. This plugin serves as the foundational building blocks for effortless framework implementation, empowering developers to seamlessly incorporate the DeepReality framework into their projects. Rooted in openness, this approach fosters collaboration and spurs continuous refinement, nurturing a dynamic community-driven culture that ensures sustained accessibility and adaptability. Moreover, the tangible value of DeepReality is substantiated through rigorous validation. Rigorous testing across a diverse range of commonly available devices underscores its practicality in real-world scenarios. This validation underscores its adaptability across various hardware configurations and everyday contexts, establishing it as a robust solution poised for widespread adoption. Deep at its core, DeepReality embraces a metrics-driven philosophy. By meticulously collecting and analysing performance metrics, the framework uncovers insightful revelations about its strengths and potential areas for enhancement. This data-centric approach empowers developers with informed decision-making capabilities, enhancing the tangible benefits of integrating DeepReality into their projects. In essence, DeepReality serves as a bridge that seamlessly connects the realms of AR and AI. Its contributions span a wide spectrum, encompassing adaptable data-driven integration, open accessibility, real-world validation, and meticulous metrics analysis. This framework empowers developers to transcend conventional boundaries, infusing AR applications with the transformative potential of AI. As it achieves this, DeepReality sets the stage for benchmarking, comparison, and a future where AR and AI harmoniously converge.

The paper is structured as follows: after the state of art presenting different papers that use DL in the development of AR applications (Section 2), the methodology is described in Section 3, which is the core of DeepReality. Section 4 presents the experiments conducted followed by the Complexity Analysis (Section 5). Finally, in Section 7, conclusions and discussion about future directions for this field of research are drawn.

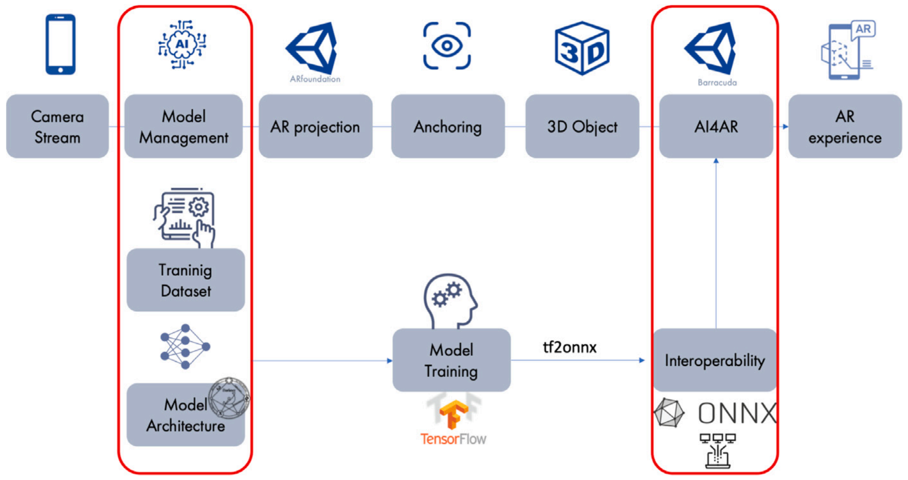

Fig. 1. General workflow. In red, the main contribution of DeepReality in the well-established AR pipeline.

2. Related work

After briefly going through AR, this section presents and analyses related applications and systems which combine AI with these AR technologies. In the context of AR and its synergy with AI, it is evident that current approaches are not without their limitations. Marker identification, a crucial facet of AR, often involves cumbersome and time-consuming processes, hampering real-time interaction. Content creation for AR applications remains intricate, requiring specialised skills and resources that limit accessibility. Additionally, environmental tracking, a cornerstone of successful AR experiences, faces challenges with incongruent objects and complex backgrounds. Furthermore, achieving optimal performance on mobile devices for resource-intensive tasks like deep learning inference poses a persistent obstacle. These limitations underscore the need for innovative solutions that can bridge these gaps and unlock the full potential of AR-AI integration. Companies, enterprises, governments, and academies have invested in AR research to the value and the potential that its applications promise to offer. As AR diffusion becomes more widespread, several software development platforms, such as Vuforia,[1] ARCore,[2] ARkit,[3] WikiTude[4] have been designed to expedite the development of AR applications. The comparison between the aforementioned SDKs have been analysed, since the platform-specific APIs have de facto revolutionised the process of developing AR applications (Amin & Govilkar, 2015; Nowacki & Woda, 2020).

According to Tanskanen, Martinez, Blasco, and Sipiä (2019), AR is a technology that has the ability to close the gap between physical and virtual worlds; while AI can influence machines in understanding the world. In literature, several works have been proposed especially in medicine with the main goal to assist clinicians in surgical operations or to facilitate dynamic clinical decision (Shen et al., 2023).

The integration of DL into AR can indeed provide a lot of interactive, real-time, user-friendly, and user-centred applications which focus on enhancing the quality of experience. It can be assumed that applications and systems that adopt these technologies are progressively being developed. Object recognition using DL-based techniques guarantees excellent performance from the point of view of accuracy and processing of data in real-time, facilitating the combination of AR and DL (Cheng, Zhang, Bo, Chen, & Zhang, 2020; Le, Nguyen, Yan, & Nguyen, 2021; Tan, Pang, & Le, 2020).

In industry, there are different examples of AR applications that integrate DL algorithms (Devagiri, Paheding, Niyaz, Yang, & Smith, 2022). The work proposed in Subakti and Jiang (2018) aimed to design, develop and implement a fast mobile increased reality system without marker to provide recording, display, and interaction with machines in indoor intelligent factories with industry vision 4.0. They used a DL algorithm (MobileNet (Howard et al., 2017)) to process images and AR by superimposing additional information on the image of the portion of the machine shown on the display. In the same context, it is settled the work of Kim, Choi, Park, and Lee (2021) where the authors proposed a hybrid approach that included a segmentation algorithm based on DL by considering RGB images acquired by AR camera. Simultaneously, a depth prediction method was applied to the AR image to estimate the depth map as a 3D point cloud for the detected object. Based on the segmented data of the 3D point cloud, 3D spatial relationships between physical objects were determined to solve problems of occlusion and visual mismatch. In another work (Kästner, Frasineanu, & Lambrecht, 2020), the authors considered mobile robots used in the industry. The authors presented an approach that performed calibration based on DL of AR devices using data from 3D depth sensors. They considered an approach markerless that used a modified version of VoteNet (Ding, Han, & Niethammer, 2019) architecture that directly processed the raw point cloud input.

Wang et al. exploited a two-step knowledge-based DL algorithm that automatically detected damages in real time using AR smart glasses. The first phase of the algorithm involved the recognition of the areas prone to damage in the region of interest. During the second phase, automatic damage detection was performed independently within each of the identified areas beginning with the one most subject to damage. This approach significantly improves the likelihood of detection when dealing with structures with complex geometric characteristics (Wang, Zargar, & Yuan, 2021).

Another interesting paper was presented in Svensson and Atles (2018), where AI is used to perform the object detection task. The authors develop an iOS application, which combines AR with object detection and recognition tasks. The purpose was to verify if the combination of these fields is possible with the modern tools available. The application must be an alternative option to traditional furniture assembly manuals.

In urban environment domain, noteworthy is the work proposed by Alhaija, Mustikovela, Mescheder, Geiger, and Rother (2017). They proposed a method that combines real and synthetic data for learning segmentation models of semantic instances. They augmented the real-world images with virtual objects of the target category represented by cars. During the experimental phase, to demonstrate the effectiveness of

[1] https://developer.vuforia.com/
[2] https://developers.google.com/ar
[3] https://developer.apple.com/augmented-reality/
[4] https://www.wikitude.com/

the proposed approach, they used a DL algorithm for the segmentation of the semantic instance.

In the same domain, the work of Polap, Kesik, Ksiazek, and Wozniak (2017) developed a real-time reality-based inspection of the environment, to inform the user of any impending obstacles. The proposed solution employed a CNN architecture to provide as much input information as possible. AR was applied to give in a real-time safety alerts. A broadly similar approach was proposed in Abdi and Meddeb (2018). They presented a real-time object detection method based on DL to define and recognise the types of road obstacles, and also to analyse and predict complex traffic situations. They introduced an AR approach to creating interactive real-time traffic animations considering different types of obstacles, positioning, and visibility rules, and projecting these in a vehicle display device.

Considering workplace safety, the work of Bhattarai, Jensen-Curtis, and Martínez-Ramón (2020) aimed to assist firefighters during their rescue operations. They proposed an integrated system prototype that used data transmitted by cameras integrated into firefighters' personal protective equipment to acquire thermal, RGB colour, and depth images and then use DL models to analyse in real time the input data. The system processed images via wireless streaming and relayed to the firefighters through an AR platform which allowed to display the results and focused attention on the objects of interest, otherwise invisible from smoke and flames. Another important point of interest when dealing with AR is the use of Adversarial Networks (Lim, Kim, & Ra, 2018), which are able to generate content in real-time forecasting the users' behaviours (Kim, Lim, & Ro, 2019).

With respect to the above-mentioned state-of-the-art works, DeepReality allows the integration within Unity AR module of any DL model, through AR Foundation and Barracuda inference engine. Thusly, the tracking is not marker-based, but network-inference based, ensuing paramount to support more accurate and robust environmental tracking-based in AR. The proficiency of DeepReality is to perform object semantic processing within the scene, and extended semantic effects for incongruent objects, surmounting the environmental tracking, which is feature-based.

3. Methodology

This section outlines the components of the plug-in development by highlighting the technical issues that are nowadays hampering the exploitation of AI within AR applications. As stated in Introduction and schematically depicted in Fig. 1, DeepReality pipeline relies on Unity and AR Foundation, a package that acts as an interface between Unity and platform-specific AR libraries (ARCore and ARKit), giving a unified way to access their common functionality. The AR Foundation components particularly demand AR Session, AR session origin, AR Raycast Manager (Unity, 0000a). Firstly, data are acquired and passed to a training module, where a new neural network model is created. The next step is deploying this model to a smartphone with an application able of running neural networks on its camera input. The object detection is executed on Unity Barracuda with pre-trained models such as tiny YOLOv2 and MobileNetV2, as in Pierdicca et al. (2022). However, the user can develop its own models and can also customise standard CNN architectures using nodes and activation functions supported by Barracuda (Unity, 0000b). Barracuda is a neural network inference library, it uses platform-specific API (Vulkan for Android and Metal for iOS) to make the computation required by model (Unity, 0000b). In order to work with Barracuda, DL models must be converted in ONNX standard, which allows easy interoperability between different frameworks. Tensorflow (Abadi et al., 2016) does not present the possibility to export models in ONNX; it requires the use of tf2onnx, a package that converts TensorFlow (tf-1.x or tf-2.x), tf.keras and tflite models to ONNX via command line or python api. PyTorch (Paszke et al., 2019), instead, does not require additional tools to extract the ONNX representation of a model, allowing a direct export.

The system is designed to integrate state-of-the-art DNNs within the components to allow the generation of dynamic AR content. It was also necessary to draw up guidelines for the integration of these networks, which serve as a guide for the integration or implementation of forthcoming DNNs.

The overall sequence diagram, including the main class developed, is depicted in Fig. 2. The general functionality of DeepReality classes is described by the following steps:

1. When the Unity scene is loaded, the Start method is called by Unity itself. This method starts the initialisation procedure, mainly calling the initialisation method of the *ModelExecutor*.
2. When a specific amount of time has passed, a frame from the device's camera is captured.
3. This frame becomes the input of the *IPreProcessor* class that generates the input for the DL model based on the data captured from the camera;
4. The Output Tensor of the *IPreProcessor* moves on the *ModelExecutor*, which returns a Dictionary that contains Tensors representing elements found with the DL model.
5. The Output of the model is given to the *IPostProcessor* to obtain an output that can be visualised.
6. The Model Outputs are gone to the *ARProjector*, which tries to convert them in world space.
7. The outputs of the previous step are mixed with the ones already presented from previous iterations.
8. If a new Output is obtained, an anchor (an empty *GameObject*) is generated at the centre of its area. A copy of a prefab specified by the user is also generated as a child of that anchor. This prefab should have a component that implements the *IARObject* interface. This prefab is generally what the end user will see in AR on top of every detected element.
9. If output is too old (i.e., the corresponding element has not been detected for time) it is deleted together with its corresponding anchor (and subsequently its *ARObject*).
10. Repeat from step 2.

The filtering functionality described in steps 5 and 7 is extremely important for AR experience: duplicate or former elements viewed in AR could be extremely confusing for the end user. To filter out duplicate elements the world space area of a newly detected element is compared to the ones from currently detected elements. If an area that has similar characteristics is found, it is assumed that it represents the same object, and thus it is discarded in support of the newly acquired one. If an already present area is updated, its anchor and *ARObject* is also updated to represent the new information. Every output considered valid is stored in a *SessionOutput* instance, accompanied by its anchor, its *ARObject*, and its age (the time that has passed from its last detection). The age is used to determine if it should be discarded (as stated in step 7).

On a functional level, DeepReality is divided into three main areas which are detailed in the following SubSections: Session manager (Sub Section 3.1), Model Handling (Sub Section 3.2), AR Projection (Sub Section 3.3).

3.1. Session manager

The *Session Manager* class is responsible for managing the whole process, and in particular, fulfils the following tasks:

- It initialises the execution of the application.
- it periodically acquires an image from the camera.
- It sends the image to the *Model Handling* and gets back the objects found by the model.
- It sends those elements to the *AR Projection* and receives back the 3D space coordinates.

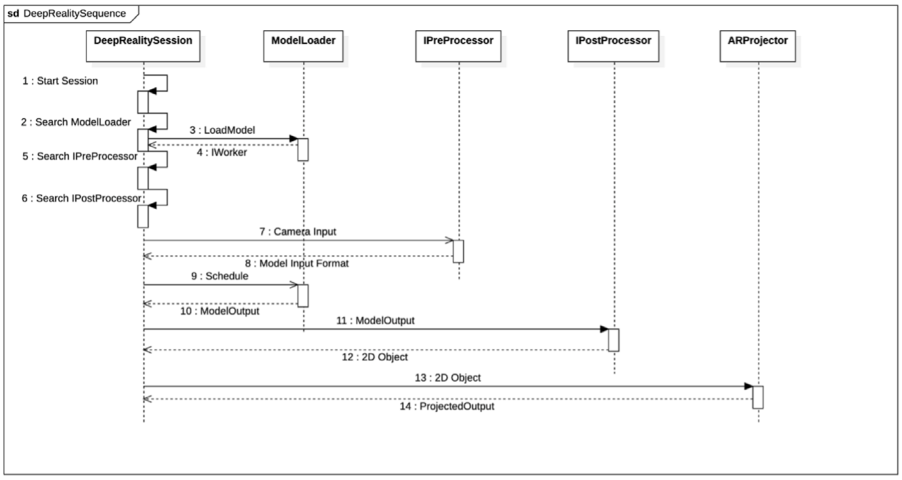

Fig. 2. Sequence Diagram on the execution of the standard phases of a DeepReality application.

- It places the objects on those coordinates.

This class communicates with the other classes of DeepReality, as well as the underlying Unity Engine. It also performs important filtering functionality on the detected elements, trying to avoid duplicate representations of the same element. In fact, before placing new 3D objects on top of newly identified elements, some filtering actions are necessary. When DL model is executed and screen areas are identified as elements of interest, there is no intrinsic notion of previous executions. This will cause any element to be identified (and then projected) multiple times, but the pertinent 3D object must be placed only once, otherwise, the scene will get extremely cluttered with duplicate objects. To solve this problem, we pursue to understand if any newly identified element is “really” new or if it has been identified before. This is done by comparing the newly found coordinates with those of the objects already identified, assuming that those that are too close together should actually be corresponding to the same object.

The general schema of the session manager class is reported in Fig. 3.

Session Manager has also several attributes, such as:

- The time interval between the executions of the DL model;
- A list of the currently detected and tracked elements;
- The prefab that is generated when a new element is detected;
- The time that must elapse before an element is considered expired and removed.

Besides, it has several events and the other classes can subscribe to perform actions when elements are detected, changed, or removed.

DeepReality has also the goal of instantiating user specified *GameObjects* where something was recognised by an object detection DL model. This is all handled by the DeepReality session. To specify the *GameObject* to be instantiated, one should set the ARObjectPrefab attribute of the Session Manager. Whatever is specified there will be instantiated in the world position where something was recognised. The basic position and rotation are handled automatically. To obtain the *GameObject* response on the recognised object, it should be created some components (MonoBehaviours) that implement the *IARObject* interface. When the ARObjectPrefab is instantiated, the UpdateData method is called on every component that implements the IARObject interface. In the *ProjectedOutput* instance that is passed to the method all the data can be found as:

- pose: World space pose. The object is automatically positioned according to this value, but it can be used to perform any additional actions required.
- description: Description string of what was recognised. It should generally be the class label of the recognised object.
- confidence: Reported confidence of the recognition.
- data: Additional data relevant to the recognition. Its value strongly depends on the DL model (it can also simply be null if not needed).

3.2. Model handling

The *Model Handling* class is responsible of loading and using the DL model. The model to be loaded must comply with the Barracuda specifications that can be found in the Unity documentation (Unity, 0000b). At the core of the processing of a frame by the DL Model there are 3 different interfaces (see Fig. 4):

- IModelLoader;
- IPreProcessor;
- IPostProcessor.

In a scene where a DeepReality session should be in execution, there must be a component that implements each interface (obviously a component can implement more than one at a time). The implementation strictly depends on the architecture of the DL Models that are chosen and tailored to their own needs. Once a model is loaded, it is used to process the images acquired from the system camera, with the aim to detect the objects of interest. When the object is detected, a data structure is created that contains the screen coordinates where the object is located and other information (e.g., description). Different models require different inputs and have different outputs. To perfectly integrate them with the system is necessary a pre-processing and post-processing phases. For proprietary models, there is a cloud platform that provides pre-processing and post-processing functions; while for custom models, the user must write these functions.

3.2.1. Model executor

This is the main class of the *Model Handling* functionality. It holds references to instances of classes that implement *IModelLoader*, *IPreProcessor* and *IPostProcessor*. It has an initialisation method that mainly uses the *IModelLoader* instance to load and initialise a Barracuda-compatible DL model, getting an *IWorker* instance (interface from the Barracuda package) that will be used to execute the model. It also has a method that, given an image as input, returns a list *ModelOutput* instances, each one holding the screen coordinates and data of an element recognised by the model. The recognition is done by executing the loaded model with Barracuda. To adapt the input image to the required input of the model the *IPreProcess* instance is used. To convert the output of the model to the required list of *ModelOutput*, the *IPostProcess* instance is used.

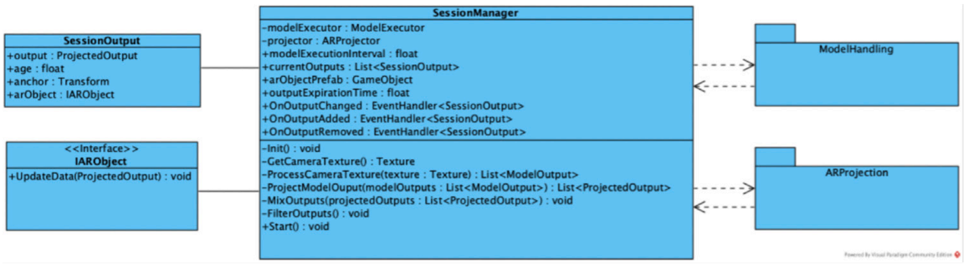
SessionOutput
+output : ProjectedOutput
+age : float
+anchor : Transform
+arObject : IARObject
<<Interface>>
IARObject
+UpdateData(ProjectedOutput) : void
SessionManager
-modelExecutor : ModelExecutor
-projector : ARProjector
+modelExecutionInterval : float
+currentOutputs : List<SessionOutput>
+arObjectPrefab : GameObject
+outputExpirationTime : float
+OnOutputChanged : EventHandler<SessionOutput>
+OnOutputAdded : EventHandler<SessionOutput>
+OnOutputRemoved : EventHandler<SessionOutput>
-Init() : void
-GetCameraTexture() : Texture
-ProcessCameraTexture(texture : Texture) : List<ModelOutput>
-ProjectModelOuput(modelOutputs : List<ModelOutput>) : List<ProjectedOutput>
-MixOutputs(projectedOutputs : List<ProjectedOutput>) : void
-FilterOutputs() : void
+Start() : void
ModelHandling
ARProjection

Fig. 3. Session Manager Class Diagram.

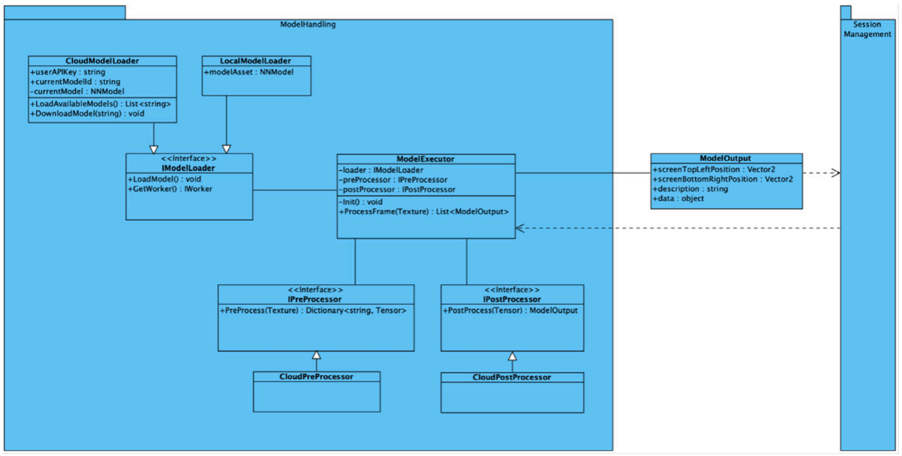

Fig. 4. Model Handling Class Diagram.

3.2.2. *Imodel loader*

The *IModel* Loader interface defines the basic structure that must be respected in order to read the model from the ONNX file and load it into memory. The basic requirements of this interface are the implementation of the *LoadModel* and *GetWorker* methods. *LoadModel* is the method that accesses the model inside the application, whether downloaded from the cloud or manually inserted during the design phase from external sources and loads it into the memory of the device, representing it in a special Barracuda data structure called Model. Being an interface, each instance may correspond to different ways of loading the model into the memory. This phase is fundamental for understanding whether or not the model can be used by Barracuda and the hardware on which the application is being tested. *GetWorker* is the method that, once the model has been opened, creates a class that executes the model using the hardware available. The *IWorker* structure resulting from the call is the representation in an object of the hardware dedicated to the model, ready to carry out the computation necessary for the call to obtain the result from it. This method gives the representation of the model and the hardware on which it will be executed during use. The *IModel Loader* is an interface that must be implemented by the classes responsible of loading and initialising the DL model. Different implementations can of course load models in different ways, for example by referencing a model in the unity project's assets. This interface contains two methods, one for loading the model and one that returns an *IWorker* ready to be used. The loading method must be called before generating a worker.

3.2.3. *IModelPreProcessor*

The *IModelPreprocessor* is an interface that must be implemented by the classes responsible of converting the image that is passed to the *ModelExecutor* in a list of Tensor instances (class from the Barracuda package) that will be used as input for the model. This conversion must be implemented knowing exactly what the used model expects as input. Different models will most likely require different implementations of the *IPreProcessor* instance. This interface contains only one method that performs the conversion. This information, available outside the representation through the implementation of the methods "RequiredFrameWidth" and "RequiredFrameHeight" will be used by *DeepRealitySession* to generate a frame of the expected size to lighten, therefore, the workload of a developer who will only have to add, if present, other phases of pre-processing before generating the Tensor that represents the input image. To obtain the input tensor for the *DeepRealitySession* model it will call the PreProcess method. This method will use the current frame, apply the pre-processing required by the network and insert the result inside a Barracuda Tensor of the same size of the image. Finally, it will assign this tensor to the key of the dictionary that represents the name of the input layer of the model inside the *IWorker*.

3.2.4. *IModelPostProcessor*

The *IModelPostProcessor* is the interface that represents the operations necessary to obtain, from the output of the DL model, a correct representation that is comprehensible, and therefore visible, for the user. In fact, there are two methods necessary to achieve this result:

- *RequiredOutput* that will contain a list of strings useful to identify the layers from which to extract the result of computation of the model contained in the *IWorker*;
- *PostProcess* which is the main method that takes as input the Dictionary containing the Tensor output of the model and the string that identifies the layer from which it was obtained. Based on the key the developer can then perform different tensor processing to describe the output within the *ModelOutput* representation (explained later). Since several outputs can be obtained simultaneously from an DL model, the implementation of this method requires that it returns a list of *ModelOutputs*.

The *IModelPostProcessor* is an interface that must be implemented by the classes responsible of converting the output Tensor of the model in a List of *ModelOutput* instances. Each *ModelOutput* represents a single recognised element, with its screen coordinates and any eventual additional data. If nothing is recognised an empty List will be returned. This conversion must be implemented knowing exactly what the used model outputs. Different models will most likely require different implementations of the *IPreProcessor* instance. This interface contains only one method that performs the conversion.

3.2.5. ModelOutput

The *ModelOutput* class describes the data structure to be used to represent the output of the processing of a DL model. It contains the following data structure:

- ScreenRect, which represents a rectangle on the screen, i.e., the bounding box in which to insert the result of the neural network execution,
- Description, which will contain a string describing what the model has recognised,
- Data that is an additional data structure that may contain additional information or data structures obtained from different models,
- Confidence that is a measure of the goodness of the result achieved by the network.

This class is useful for representing data with a defined structure using ARProjector integrated within DeepReality. It is possible to use the output of the model in a different way, by implementing classes to manage the representation and projection in the real world. This is an interface that must be implemented by the classes responsible of converting the output Tensor of the model in a List of *ModelOutput* instances. Each *ModelOutput* represents a single recognised element, with its screen coordinates and any eventual additional data. If nothing is recognised an empty List is returned. This conversion must be implemented knowing exactly what the used model outputs. Different models likely require different implementations of the *IPreProcessor* instance. This interface contains only one method that performs the conversion.

3.3. AR projection

The *AR projection* module (Fig. 5) is responsible for taking areas of the screen and projecting them in the 3D world using AR Foundation. The purpose is to convert the screen-space coordinates identified by the model in 3D coordinates for 3D objects. It used advanced AR frameworks such as ARKit (for iOS) and ARCore (for Android) since they offer great functionality and are well-integrated with their respective system. By using those frameworks, we can raycast a point on the screen which is projected from a 2D space to a 3D space (the real world). The result of the projection is a ranking list, ordered by decreasing robustness, of possible anchor points in the real world. The first four points and the average between them are then used to make the pin more robust in reality, allowing one to place objects in a more precise position. Furthermore, they can track the movement of the device in the real world, allowing the execution of the DL model more sporadically while still tracking identified areas in real-time: once an object is placed in the 3D world (as a result of something being identified through DL), it remains in the same physical position even as the device moves around.

4. Experiments and analysis

In this section, two different DL tasks are presented to test DeepReality plug-in functionality. The DNNs architectures and related datasets on which the models were trained are described. Moreover, a general framework is provided to develop proper models to be integrated into the proposed solution.

Table 1
Number of parameters and weight storage of state-of-the-art DNNs for DL classification task.

Classification models	Parameter (Million)	Weight storage (MB)
DenseNet	8.1	33
VGG-16	138.3	528
AlexNet	60	220
InceptionV3	23.8	92
ResNet-101	44	171
MobileNetV2	3.5	14

Table 2
Frame per second (FPS) performed on a Pascal Titan X and weight storage of principal state-of-the-art object detection models.

Object detection models	FPS	Weight storage (MB)
Faster R-CNN	4	168
Retinanet-101	11	228
SSD500	19	77
YOLOv1	45	190
Tiny YOLOv1	155	58
YOLOv2	40	202
Tiny YOLOv2	244	43
YOLOv3	35	237
Tiny YOLOv3	220	34

4.1. Classification task

The model used for classification task is MobileNetV2 (Sandler, Howard, Zhu, Zhmoginov, & Chen, 2018), which presents a lightweight CNN architecture. Thanks to its small size, MobileNetV2 is widely used due to the low computing power needs for real-time inferences. Therefore, it has become a benchmark model for mobile devices and embedded systems. The architecture is characterised by depthwise separable convolutions, which significantly reduce the number of parameters compared to CNNs with regular convolutions with the same depth. Also included in MobileNetV2 are two additional basic structures, namely the linear bottle-neck layer and the reverse residual structure. Both of these features contribute to accelerating model convergence and preventing gradient vanishing. The number of parameters and weight storage consumption of the MobilenetV2 model compared to other state-of-the-art classification methods are reported in Table 1.

The employed MobileNetV2 model is trained on the ImageNet benchmark dataset (Deng et al., 2009), which spans 1000 object classes. The network takes as input a tensor with shape $[224, 224, 3]$ (corresponding to height and width of the image and RGB channels, respectively), while the output is a tensor with shape $[1, 1, 1000]$ containing the probability values for each of the 1000 classes.

4.2. Object detection task

The state-of-the-art object detection approaches include objectness detection (OD), salient object detection (SOD) and category-specific object detection (COD) (Han, Zhang, Cheng, Liu, & Xu, 2018). In DeepReality, COD is performed, where multiple objects can be detected in the same image based on predefined categories. The process consists of identifying both the image regions that may contain the object of interest as well as the specific object category for each region. The model chosen for object detection task is the Tiny YOLOv2 (Redmon & Farhadi, 2017), which belongs to the family of regression-based one stage detectors. For embedded real-time applications, one-stage detectors are the most promising COD direction since they are highly efficient: the entire detection process runs in real-time with little memory and storage demand (Liu et al., 2020; Zhao et al., 2019). The comparisons of speed, in frames per second [FPS], and storage consumption on COCO (Lin et al., 2014) test set for state-of-the-art COD models are shown in Table 2.

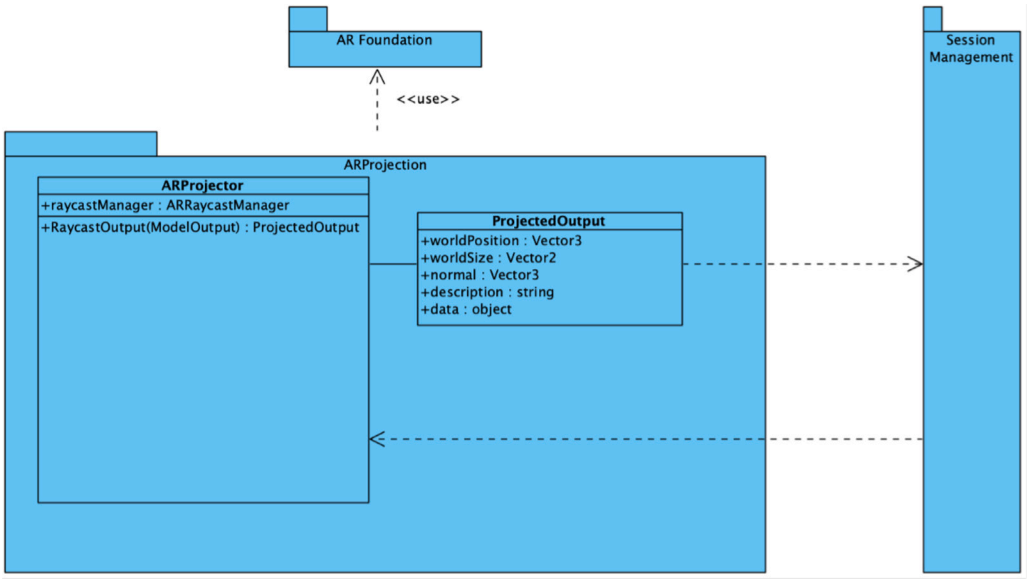

Fig. 5. AR Projection Class Diagram.

The Tiny YOLOv2 architecture is constituted by convolutional layers with a 3×3 kernel and max-pooling layers with a 2×2 kernel. Respect to the traditional YOLOv2 architecture, the number of layers is reduced in the Tiny version, presenting only 9 convolutional layers and 6 pooling layers. We use Tiny YOLOv2 trained on COCO benchmark dataset (Matsuda, Hoashi, & Yanai, 2012), which contains 80 general classes. The input layer expects a tensor with shape $[3, 416, 416]$, while the output layer generates a $[525, 13, 13]$ tensor. The output divides the image into a 13×13 grid, with each cell in the grid containing 525 values. A cell includes 5 potential bounding boxes, and each one is represented by the following 105 elements:

- x: the x position of the bounding box centre relative to the grid cell it is associated with;
- y: the y position of the bounding box centre relative to the grid cell it is associated with;
- w: the width of the bounding box;
- h: the height of the bounding box;
- o: the confidence value that an object exists within the bounding box, also known as “objectness score”;
- $p1$-$p100$: class probabilities for each of the 100 classes predicted by the model for the considered dataset.

Besides the performance tests, a mobile execution test was conducted. Inference examples are shown in Fig. 6. For the Tiny YOLOv2 model, the result displays the class of the recognised object and the bounding box projected in reality through the use of Unity’s ARFoundation. Moreover, an example of MobileNetV2 execution is shown. The model correctly recognises the object, i.e. a mouse, and it projects in reality the related name in the centre of the object.

5. Complexity analysis

The complexity analysis allows to evaluate efficiency and computational problems according to their inherent difficulty. In this case, the complexity analysis is performed to evaluate whether DeepReality can be completely executed on mobile devices. Before analysing the computational complexity, a definition of the key values involved is required. In particular, here following are listed as the key values:

- N is the computational complexity of the input image, it is also indicated with the values: W for Width and H for Height;
- C is the number of analysed classes;
- c is the number of characters in a string
- B is the number of bounding boxes analysed in output to the detection model;
- R is the number of rows for the cell division of the image for the application of detection;
- P is the number of columns for splitting the image for the application of detection;
- f is the counting features for detection;
- F is the number of filters;
- D is the filter size
- I is the number of neurons in the connected dense layer;
- L number of network blocks;
- K number of key-points extracted from the image;
- E is the number of bits for the BRIEF descriptor.

In particular, it is analysed each component of DeepReality (Section 3): Session manager, model handling, AR projection. The complexity analysis deepens four development phase:

- *Model Loader*: which takes care of loading the model in memory and its execution. This component has a complexity that depends on the model loaded;
- *Pre-Processor*: responsible for obtaining the input structure suitable for the model in use;
- *Post-Processor*: which processes the output of the model and makes it suitable to be projected into reality;
- *AR Projection*: which projects virtual objects onto reality.

5.1. Model executor

As mentioned above, the model is loaded and executed in this phase. In this case, the time complexity of the DL model depends on the configuration adopted. For this component, the complexity analysis is based on the definition in Vaswani et al. (2017). First of all, the networks considered, i.e. MobileNet and Yolo, have two types of layers:

- *Convolutional*: characterised by a series of filters of different sizes and depths that are applied by means of a series of dot products.
- *Fully-Connected*: characterised by a series of neurons following each other to determine a result given by the sum of all activation preceding the output neuron(s).

Within the MobileNet architecture, there is a block, called BottleNeck. The complexity of this block can be defined as follows:

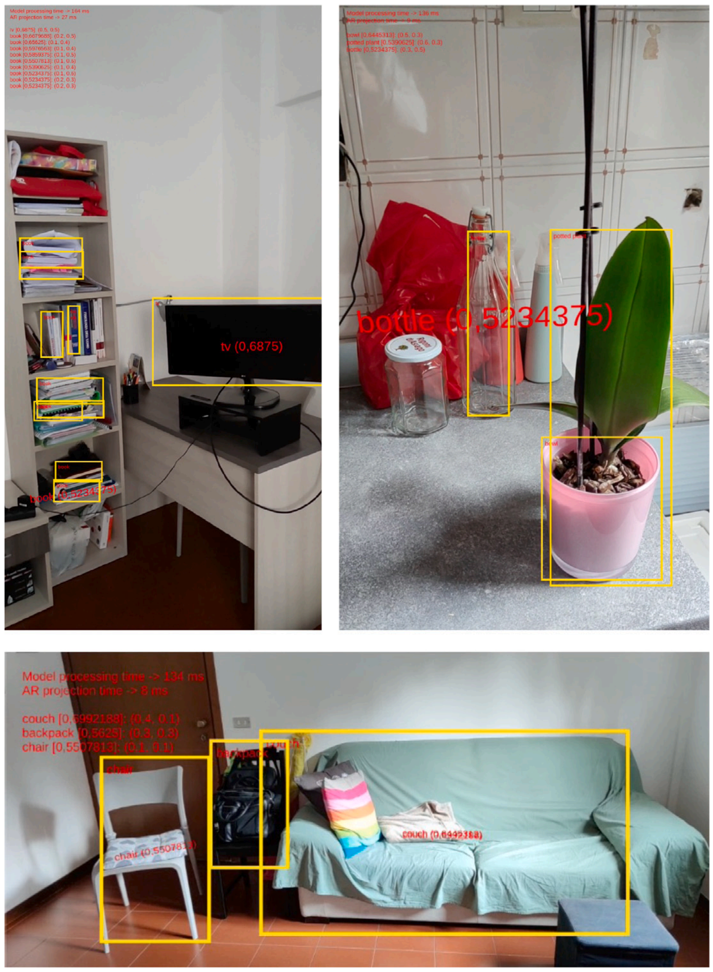
tv (0,6875)
bottle (0,5234375)
Model processing time -> 134 ms
AR projection time -> 8 ms
couch [0.6992188]: (0.4, 0.1)
backpack [0.5625]: (0.3, 0.3)
chair [0.5507813]: (0.1, 0.1)
couch (0.6949388)
chair (0.5507813)

Fig. 6. Real-time execution examples of Tiny YOLOv2 model.

- 1×1 CONV2D: this operation considers the convolution applied with filters of size 1 so the complexity of this layer is given by the input image and the number of filters applied, thus the total complexity is $O(NF)$.
- 3×3 DEPTHWISECONV: it applies the convolution operation by considering 3-dimensional filters that are applied to the image by considering all its channels at once, thus the total complexity is $O(NFD^3)$.
- CONV2D: convolutional layer that applies F convolutional filters of dimension D to the input image, so its total complexity is $O(NFD)$.

For the fully connected layer, it is applied in the final phase of the network. In particular, it has as input the output of the last convolutional layer that is represented by a single array with I neurons. Therefore, the computational complexity is $O(NI)$. The total complexity is:

$$(L * O(NFD^3)) + O(NI) = O(LNFD^3) + O(NI) \tag{1}$$

YOLO, instead, is a Fully Convolutional model, thus by applying F filters of dimension D on the input of dimension N and repeating them L times, we obtain that the convolutional complexity of the network is $O(LNFD^2)$. To this we must add the complexity of the pooling layers, applied during the execution of the model to reduce the dimension of the input image and to individuate features at different depths. We can consider such operation as a matrix operation applied on the image, with a complexity $O(ND^2)$. The total complexity of the model is:

$$O(LNFD^2) + O(LND^2) \tag{2}$$

Table 3 reports the complexity of this phase considering the components involved.

5.2. Pre-processor

Considering that the models implement computer vision algorithms and that the expected inputs are images, the procedures here analysed consist of matrix operations on the images. The following steps are implemented:

1. acquiring the image from the camera;
2. resizing the image to fit the expected input from the model;
3. transforming the image into a byte-array;
4. creating the input tensor.

The acquisition step relies on resolution camera, i.e. it is atomic, thus it is considered the worst case notation, assuming that is executable in $O(WH)[O(N)]$. The transformation step, instead, can be divided into memory space created for the new image (respecting the final size of the transformation) and an anti-aliasing algorithm for resizing applications.

Considering the time, the execution complexity is O(1) because its complexity resides in memory, in fact, it will require $O(N)[O(WH)]$ memory (the size refers to the expected input from the model). The anti-aliasing algorithm is applied to avoid undesirable visualisation effects. It is based exactly on the use of convolutional filters to calculate the value of the colours at the position of the new image considering the neighbours of the image. In such a manner, a filter is applied and $K = 1$, with dimension D^2, for which the computational complexity

Table 3
Summary of Time complexity analysis for Model Executor.

Components	Time complexity
1 × 1 CONV2D	$O(Nf)$
3 × 3 DEPTHWISECONV	$O(NFD^3)$
CONV2D	$O(NFD)$
Pooling layer	(ND^2)
Fully connected layers	$O(NI)$
Total Model Executor for YOLO	$O(LNFD^2) + O(LND^2)$
Total Model Executor for MobileNetV2	$O(LNFD^3) + O(NI)$

Table 4
Summary of Time complexity analysis for Pre-Processor.

Components	Time complexity
Memory creation	$O(1)$
Anti-aliasing algorithm	$O(ND^2)$
Tensor creation	$O(1)$
Memory insertion	$O(N)$
Total Pre-Processor	$O(ND^2)$

is $O(ND^2)$. The result of the processing is inserted into the memory location created in the first step of the procedure. In this case, the insertion is done by the positions knowledge where the data is to be inserted for the complexity of the single insertion operation is $O(1)$ which repeated for the size of the image reaches the complexity $O(N)$.

In the Pre-Processor, the transformation in a byte array of the input image is performed as an atomic operation because this modification both in memory and in execution does not require any intermediate operation or use of new memory. This operation requires a time equal to $O(1)$.

Two alternatives could be considered for the creation of the input tensor:

- Creation of a new memory location that has the following characteristics: size greater than the size of the input image, since both its insertion in this location and the string containing the name of the input layer of the network to which the image must be assigned as input must be considered. In this case, we can examine: *time complexity* of $O(1)$ for the creation of new memory locations, $O(N)$ for transferring the image within the new location; $O(1)$ for inserting the string into the memory location; *Memory complexity* equals to $O(N)$ for the image and $O(c)$ for the string.
- Creation of a memory location for the string containing the name of the input layer and link to the location where the image is. In this case, the memory complexity becomes $O(c)$ for the string only. While the time complexity becomes $O(1)$ since the string insertion occurs in $O(1)$ and the addressing of the memory location where the image resides is an $O(1)$ operation since the address of the first memory location must be entered.

Summing up, the total time complexity is: $O(N) + O(N) + O(1) + O(N) = O(N)$ and the total memory complexity is: $O(N) + O(c) = O(N + c)$. Table 4 summarises the time complexity for each component of Pre-Processor phase analysed.

5.3. Post-processing

The post-processing phase is relevant to evaluate the two DL models under consideration. For MobileNet DL model, the output processing selects the maximum probability of the one-dimensional tensor that is obtained as output from the model. It can be considered as an array search and when the worst case occurs the maximum is in the last position of the one-dimensional tensor. For example, if C is the number of classes expected from the model as output and if we consider the execution complexity as an ordered array since each index represents a class, for the search of the maximum we obtain $O(C)$. In this way, obtaining the index of the maximum of the array containing the probabilities of the classes in output from the MobileNet model, we know the index of the memory location in which resides, within an array of strings, the name of the class for which the access to this location can be carried out with time $O(1)$. In this process, the memory complexity for the post-processing requires $O(cC)$ to store the strings, where c is the maximum number of characters that are used to describe a class and C the number of classes, to which we add $O(C)$ for the memory location of the one-dimensional tensor that contains the probabilities of each class and is obtained as output from the MobileNet model. In total the time complexity will be: $O(1) + O(C) = O(C)$. The memory complexity is: $O(cC) + O(C) = O(cC)$

For YOLO DL model, the output processing has two main components: Parsing Output (Algorithm 1) and Filtering Output.

```
forall $Column$ do
    forall $Row$ do
        forall $Feature$ do
            Extract Box Dimension
            Get Box Confidence
            Extract Classes
            Get Top result for class
            Map box to cell
        end
    end
end
```

Algorithm 1: Parsing output Pseudocode

The iterations listed in Algorithm 1 can be detailed as follows:

- *Extract Box Dimension*: it is a procedure which extracts the tensor information based on the knowledge of the locations where and therefore the time required is $O(1)$.
- *Get Box Confidence*: it extracts, once again, the information from the tensor based on the knowledge of the memory locations in which they are contained and for each extraction it applies the SIGMOID function, this function consists of a series of mathematical functions for which it can be considered applicable in $O(1)$ time and once again the access to the memory location, knowing the address, occurs in $O(1)$.
- *Extract Classes*: it returns the class that can be assigned to the BBOX. This procedure applies SOFTMAX on the output of the DNN and the complexity of SOFTMAX depends on classes number, hence the time complexity is $O(C)$. This operation is repeated for the number of bounding boxes that have been obtained in output. The total is $O(C) * B = O(BC)$.
- *Get Top result for class*: for each class, it reorders the BBOXes obtained by the DNN by decreasing and selecting only the first element of the array obtained for each bounding box. The complexity of the single operation is based on the number of classes C, applying the MergeSort algorithm to reorder an array, the complexity of the single operation of reordering is $O(ClogC)$, to which is added complexity equal to $O(1)$ that is the access to the position of memory in which is contained the best bounding box (that being ordered in a decreasing way) is in position 1 resulting $O(1)$. This operation is performed for each bounding box for a total of: $B(O(ClogC) + O(1)) = O(BClogC)$.
- *Map box to cell*: it applies the mapping procedure of the bounding boxes found by YOLO to the cell of the image considered. This operation involves the use of mathematical operations of mapping from the relative reference of the bounding box to the reference with respect to the cell, i.e. centring it in the cell and representing its size with respect to it, resulting equal to $O(1)$.

The total execution time of this step is given by repeating the previous steps for the number of features, the number of cells, and the number of rows:

$$f * P * R * (O(1) + O(BC) + O(BClogC) + \\ + O(1)) = O(fPRBClogC) \tag{3}$$

The filtering phase is necessary to reconstruct the global boxes on the image, i.e. to bring together the individual bounding boxes found in each cell considered and make them into a single bounding box. The algorithm 2 shows the Box filtering operations.

```
Order Box-Array
for ba in Box-Array do
    for bb in Box-Array do
        if IOU(ba,bb) > threshold then
            Exclude bb
        end
    end
end
```

Algorithm 2: Box filtering pseudocode

In this case, an array of size equal to the number of bounding boxes is used so that the sorting is equal, again considering the *MergeSort* algorithm, to $O(B\log B)$. Intersection Over Union (IoU) evaluation can be considered as a constant operation since it is always applied to the same amount of information and therefore, we can identify it as computational complexity $O(1)$, to which we can add the boolean operation, which is still $O(1)$, and the IF check equal to $O(1)$ and finally the bounding box exclusion equal to $O(1)$. Therefore, within the double FOR loop, we have a total complexity equal to $O(1)$, which must then be multiplied by the number of bounding boxes squared (for completeness, it should be multiplied by $B * (B-1)$, but using an asymptotic notation and considering the worst case we can directly use $0(B^2)$, so, in the end, the total complexity is $O(B^2)$.

Table 5 briefly reports the time complexity for each Post-Processor component evaluated.

5.4. AR-projector

This component has an important role since it projects the results of network processing into reality. The analysis of its complexity results thus fundamental and it is based on the work described in Macario Barros, Michel, Moline, Corre, and Carrel (2022). In the tools used for the development of DeepReality, i.e. Unity with AR foundation, Unity itself uses the native algorithms of the platform (iOS or Android), hence it is not possible to know the real implemented algorithms and how they are optimised within the available API.

In particular, the monocular SLAMs, i.e., ORB-SLAM algorithm (Mur-Artal, Montiel, & Tardos, 2015) is suitable within mobile platforms since it uses ORB extractors, that allow real-time extraction, and SLAMs for mapping and tracking. These extractors deserve an appropriate and detailed analysis.

5.4.1. ORB

ORB is a classical computer vision algorithm that uses FAST to extract keypoints and apply the BRIEF descriptors, thus its complexity must be defined based on both component.

- *Complexity of FAST*: it is applied to the entire image, with full resolution, i.e. at the resolution captured by the smartphone sensor. FAST involves FOR cycles that perform operations on the entire image. The operations that are carried out are the calculation of two thresholds starting from the pixels selected by the two cycles, which are therefore considered as operations carried out in $O(1)$ and subsequently, these thresholds are used for a series of IFs that are carried out once again on the whole image in the worst case, so the total computational complexity for the execution of FAST is $O(N)$.
- *Complexity of BRIEF*: in this case, it is considered the whole image as input, with the K key points extracted through FAST, and it must be considered that the descriptors are calculated considering patches of the image. Moreover, BRIEF descriptors are in binary bases and are represented using the symbol E. BRIEF is based on the steps described in Algorithms 3. In particular, the first operation of the greyscale conversion takes time complexity of $O(N)$. Gaussian smoothing is a convolution operation applied on the image. One filter is applied and $F = 1$ of square dimension and is used D^2 and its time complexity is $O(ND^2)$. Finally, the FOR cycle is executed on the number of keypoints used, within which we find the TAU function which, from a computational point of view, is an IF and therefore has an execution time equal to $O(1)$ which, added to the number of bits, will have complexity $O(E)$ and multiplied by the number of keypoints, will be $O(KE)$.

Table 5
Summary of Time complexity analysis for Post-Processor.

Components	Time complexity
Memory creation	$O(1)$
Class string insertion in memory	$O(C)$
Extract Box dimension	$O(1)$
Extract Box confidence	$O(1)$
SOFTMAX	$O(BC)$
Top result selection	$O(BClogC)$
Mapping to cell	$O(1)$
Filtering	$O(B^2)$
Total Post-Processor for MobileNetV2	$O(C)$
Total Post-Processor for YOLO	$O(fPRBClogC) + O(B^2)$

```
Conversion to grayscale
Application of Gaussian Smoothing on the image
for i in Range(1, ..., K) do
    $\sum_{J=1}^{E}(2^J * TAU(P, X_J, Y_J))$
end
```

Algorithm 3: BRIEF pseudocode

The computational complexity of ORB is:

$$O(N) + O(N) + O(ND^2) + O(KE) = \\ = O(ND^2) + O(KE) \tag{4}$$

5.4.2. SLAM

This is used for keypoint tracking and mapping. Based on the paper Burgard, Brock, and Stachniss (2008), this algorithm is executed in $O(K)$ with K that represents the number of keypoints. This is proved by the realtime execution of the devices used for the tests, these considerations are useful to give an upper asymptote for the temporal execution but the real implementations of the producers use hardware and software devices that we cannot know and therefore they cannot be taken into account in this analysis.

It is not possible to know the algorithms embedded within the proprietary APIs of Google and Apple. The analysis carried out in this section considers computations in the worst case and without optimisation, but given the performance, it can be deduced that the producers have optimised both the algorithms and also the hardware to guarantee stability and efficiency as well as effective operation. Table 6 recap the time complexity for this phase.

5.5. Time complexity analysis

The time complexity for each block is summarised in Table 7. In particular, apart from the dimension of the input image (N), the other parameters can be considered constant (e.g., filters or neurons of the DL models). For this reason, the time complexity of the execution can be assumed equal to O(N).

As shown in Fig. 7, it is possible to define an upper limit for the execution time beyond which the device does not have the appropriate capabilities to perform these types of algorithms. This limit, when considered from the perspective of computational capacity, becomes the lower limit of computational capacity required for devices to use deep learning algorithms within their applications. In particular, Fig. 7

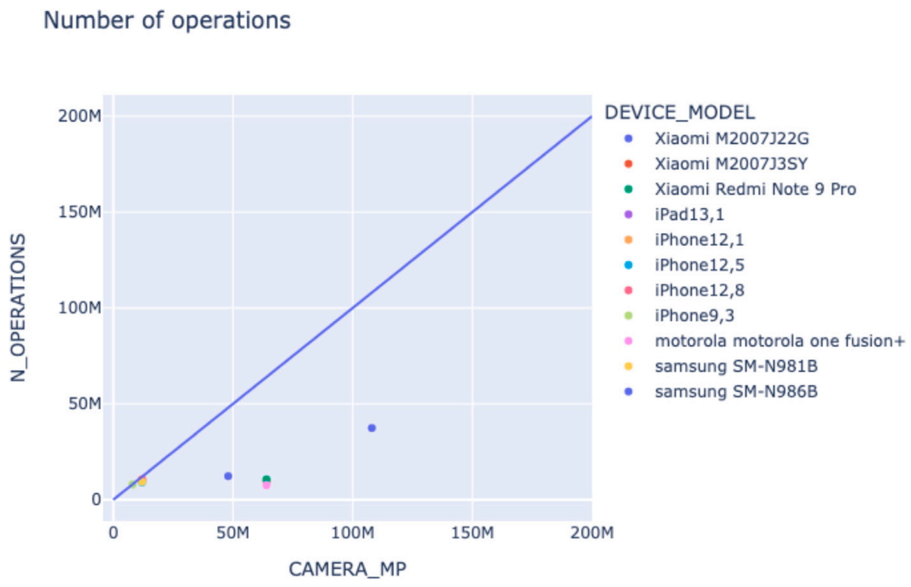

Fig. 7. Time execution on different smartphones compared to the time limit obtained by the Time complexity analysis.

Table 6
Summary of Time complexity analysis for AR-Projection.

Components	Time complexity
FAST	$O(N)$
Grayscale conversion	$O(N)$
Gaussian smooth	$O(ND^2)$
Application to all Key-point	$O(KE)$
Total AR-Projection	$O(ND^2) + O(KE)$

Table 7
Time complexity.

Components	Time complexity	Simplified
Pre-Processing	$O(ND^2)$	$O(N)$
Post-Processing (MobileNet)	$O(cC)$	$O(1)$
Post-Processing (YOLO)	$O(fPRBClogC) + O(B^2)$	$O(1)$
Model Executor (MobileNet)	$O(LNFD^3) + O(NI)$	$O(N)$
Model Executor (YOLO)	$O(LNFD^2) + O(LND^2)$	$O(N)$
AR-Projector	$O(ND^2) + O(KE) + O(K)$	$O(N)$

shows the time required for execution (in seconds) when varying the input size. Considering the simplified complexity O(N), it is possible to assume a linear dependence of the execution time with respect to the size of the input, i.e., the size of the image. For this reason, we define a linear function representing the bisector of the first quadrant of a graph and considering that the minimum resolution of the cameras available on devices is 12 Mpx, we can represent the time limit on the graph and verify that the times measured on the terminals are less than it. From Fig. 7, it is possible to figure out that the different coloured dots represent the time spent by the different smartphones for running the model, considering the camera pixels and the number of iterations.

While the a priori analysis provides valuable insights into expected performance levels, it represents an idealised scenario that does not account for real-world variations and device-specific attributes. It is precisely for this reason that we conducted the a posteriori analysis, which involved the practical implementation of the DeepReality on a range of smartphones. Fig. 7 represents the empirical results of our a posteriori analysis, depicting the actual execution times achieved on different devices. By comparing these results to the time limits obtained from the a priori analysis, we gain a comprehensive understanding of the extent to which theoretical estimates align with real-world execution. This empirical validation not only validates our theoretical framework but also provides essential insights into the deviations between predicted and observed execution times. Furthermore, the connection between the a priori and a posteriori analyses highlights the iterative nature of our approach. The insights gained from the empirical results guide us in refining the accuracy of our theoretical predictions. We view the a posteriori analysis not as a deviation from our theoretical framework but as a mechanism for continuous improvement and optimisation.

6. Usability evaluation

In this section, we provide an in-depth account of the usability evaluation conducted to gauge the practical viability of DeepReality. The evaluation was designed to encompass various dimensions of usability, offering a comprehensive understanding of DeepReality's effectiveness within real-world scenarios. The usability assessment was framed around a selection of usability dimensions, chosen strategically to provide a well-rounded perspective on the DeepReality's practicality. These dimensions were selected based on the DeepReality's intended objectives and tested with expert developers. The analysis of the usability evaluation results provided comprehensive insights into the DeepReality's performance across distinct usability dimensions:

- Ease of Use: Developers found the integration process intuitive and straightforward, contributing to elevated user satisfaction.
- Learnability: Developers familiar with Unity and deep learning concepts adapted quickly to the toolkit's functionalities, while novices required a slight learning curve.
- Efficiency: Task completion times consistently fell within acceptable limits, affirming the toolkit's efficiency in executing its features.
- Effectiveness: The toolkit effectively demonstrated object recognition capabilities and facilitated seamless integration of AR content, aligning closely with its intended purposes.
- User Satisfaction: Overall user satisfaction scores demonstrated positivity, signifying the toolkit's potential value in practical applications.

7. Conclusions and future works

AR is a mobile technology that allows to visualise, with the same point of view of the user, virtual contents superimposed on a monitor of a device (hand-held or head-mounted) that is framing a scene with the camera. AR applications have proven to be particularly effective in several domains. To facilitate the development of AR experiences by overcoming its limitations, AR and AI can be combined to obtain unique and immersive experiences. In this work, it is presented a framework for the integration of DNNs in AR applications integrated in a development environment that allows the realisation of multi-platform applications such as Unity. DeepReality integrate within Unity AR module any DL model, through ARFoundation and Barracuda inference engine. It performs object semantic processing within the scene, and extended semantic effects for incongruent objects, overcoming the environmental tracking, which is feature-based. To test the usability of the framework, two test have been performed and the execution time and memory usage data have been analysed, demonstrating the feasibility and possibility of integrating and using DNNs models in mobile applications for AR. The framework is also open-source and it is freely available in the Unity asset store, which allows to easily integrate and use in the development platform itself. The reasons that led us to choose Unity as a platform for releasing the plug-in can be found in the current trend of mobile platforms, where there are mainly two platforms, iOS and Android, but in this evolving market, it is important to allow developers the fastest possible time-to-market. Unity as a development platform for the release of the plug-in enables the rapid development of AR/VR applications, which thanks to the plug-in also speed up the use of DL. Using Unity, the native AR APIs are used, as the ARFoundation used in the plug-in activates the specific APIs for the selected platform during compilation. Furthermore, Barracuda allows the use of a standard format for exchanging models, ONNX, thanks to which models created not only with Tensorflow but also with other frameworks used for modelling neural networks can be extracted. In addition, the plug-in integrates templates into the application, so the terminal runs the template and no connection is required.

In the future, DeepReality can be improved not only by using Unity components for the integration itself but also by taking advantage of the native frameworks available today for the creation of models, such as Tensorflow and PyTorch, which today allow integration using native code and models natively developed using them and therefore have no genericity. Limits of Barracuda package will be also overwhelming, since it does not support all types of layers and operators, limiting the range of architectures that can be imported.

It will be also evaluated the possibility to develop an integration platform that allows even app developers who have no knowledge of DL to integrate models into their applications using ready-to-use models optimised for the software platforms for which they are developing. In addition to this, Unity will also be able to integrate and use these models for applications developed for new software platforms such as HarmonyOS, which is currently not supported by generic languages such as *C#*, as soon as they are compatible with the development platform.

CRediT authorship contribution statement

Roberto Pierdicca: Conceptualization, Methodology, Software, Writing – review & editing. **Flavio Tonetto:** Validation. **Marina Paolanti:** Investigation, Writing – original draft. **Marco Mameli:** Visualization. **Riccardo Rosati:** Data curation. **Primo Zingaretti:** Supervision.

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Roberto Pierdicca reports was provided by Università Politecnica delle Marche. Roberto Pierdicca reports a relationship with Università Politecnica delle Marche that includes: employment. Roberto Pierdicca has patent licensed to No patent. Nothing to declare

Data availability

No data was used for the research described in the article.

Acknowledgements

This project is funded from the European Union's Horizon 2020 research and innovation programme through the XR4ALL project with grant agreement No 825545.

References

Abadi, Martín, Agarwal, Ashish, Barham, Paul, Brevdo, Eugene, Chen, Zhifeng, Citro, Craig, et al. (2016). Tensorflow: Large-scale machine learning on heterogeneous distributed systems. arXiv preprint arXiv:1603.04467.

Abdi, Lotfi, & Meddeb, Aref (2018). Driver information system: A combination of augmented reality, deep learning and vehicular ad-hoc networks. *Multimedia Tools and Applications*, *77*(12), 14673–14703.

Alhaija, Hassan Abu, Mustikovela, Siva Karthik, Mescheder, Lars, Geiger, Andreas, & Rother, Carsten (2017). Augmented reality meets deep learning for car instance segmentation in urban scenes. In *British machine vision conference*: *vol. 1*, (p. 2).

Amin, Dhiraj, & Govilkar, Sharvari (2015). Comparative study of augmented reality SDKs. *International Journal on Computational Science & Applications*, *5*(1), 11–26.

Banfi, Fabrizio, Brumana, Raffaella, & Stanga, Chiara (2019). Extended reality and informative models for the architectural heritage: from scan-to-BIM process to virtual and augmented reality.

Bhattarai, Manish, Jensen-Curtis, Aura Rose, & Martínez-Ramón, Manel (2020). An embedded deep learning system for augmented reality in firefighting applications. In *2020 19th IEEE international conference on machine learning and applications* (pp. 1224–1230). IEEE.

Burgard, Wolfram, Brock, Oliver, & Stachniss, Cyrill (2008). Data association in o(n) for divide and conquer SLAM. In *Robotics: science and systems III* (pp. 281–288).

Cheng, Qiuyun, Zhang, Sen, Bo, Shukui, Chen, Dengxi, & Zhang, Haijun (2020). Augmented reality dynamic image recognition technology based on deep learning algorithm. *IEEE Access*, *8*, 137370–137384.

Deng, Jia, Dong, Wei, Socher, Richard, Li, Li-Jia, Li, Kai, & Fei-Fei, Li (2009). Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition* (pp. 248–255). Ieee.

Devagiri, Jeevan S, Paheding, Sidike, Niyaz, Quamar, Yang, Xiaoli, & Smith, Samantha (2022). Augmented reality and artificial intelligence in industry: Trends, tools, and future challenges. *Expert Systems with Applications*, Article 118002.

Ding, Zhipeng, Han, Xu, & Niethammer, Marc (2019). VoteNet: A deep learning label fusion method for multi-atlas segmentation. In *International conference on medical image computing and computer-assisted intervention* (pp. 202–210). Springer.

Gammeter, Stephan, Gassmann, Alexander, Bossard, Lukas, Quack, Till, & Van Gool, Luc (2010). Server-side object recognition and client-side object tracking for mobile augmented reality. In *2010 IEEE computer society conference on computer vision and pattern recognition-workshops* (pp. 1–8). IEEE.

Ghasemi, Yalda, Jeong, Heejin, Choi, Sung Ho, Park, Kyeong-Beom, & Lee, Jae Yeol (2022). Deep learning-based object detection in augmented reality: A systematic review. *Computers in Industry*, *139*, Article 103661.

Han, Junwei, Zhang, Dingwen, Cheng, Gong, Liu, Nian, & Xu, Dong (2018). Advanced deep-learning techniques for salient and category-specific object detection: A survey. *IEEE Signal Processing Magazine*, *35*(1), 84–100.

Howard, Andrew G, Zhu, Menglong, Chen, Bo, Kalenichenko, Dmitry, Wang, Weijun, Weyand, Tobias, et al. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861.

Kästner, Linh, Frasineanu, Vlad Catalin, & Lambrecht, Jens (2020). A 3d-deep-learning-based augmented reality calibration method for robotic environments using depth sensor data. In *2020 IEEE international conference on robotics and automation* (pp. 1135–1141). IEEE.

Kim, Minseok, Choi, Sung Ho, Park, Kyeong-Beom, & Lee, Jae Yeol (2021). A hybrid approach to industrial augmented reality using deep learning-based facility segmentation and depth prediction. *Sensors*, *21*(1), 307.

Kim, Hak Gu, Lim, Heoun-taek, & Ro, Yong Man (2019). Deep virtual reality image quality assessment with human perception guider for omnidirectional image. *IEEE Transactions on Circuits and Systems for Video Technology*.

Lalonde, Jean-François (2018). Deep learning for augmented reality. In *2018 17th workshop on information optics* (pp. 1–3). IEEE.

Lampropoulos, Georgios, Keramopoulos, Euclid, & Diamantaras, Konstantinos (2020). Enhancing the functionality of augmented reality using deep learning, semantic web and knowledge graphs: A review. *Visual Informatics*, *4*(1), 32–42.

Le, Huy, Nguyen, Minh, Yan, Wei Qi, & Nguyen, Hoa (2021). Augmented reality and machine learning incorporation using YOLOv3 and arkit. *Applied Sciences*, *11*(13), 6006.

Lim, Heaun-Taek, Kim, Hak Gu, & Ra, Yang Man (2018). VR IQA net: Deep virtual reality image quality assessment using adversarial learning. In *2018 IEEE international conference on acoustics, speech and signal processing* (pp. 6737–6741). IEEE.

Lin, Cheng-Hung, Chung, Yang, Chou, Bo-Yung, Chen, Hsin-Yi, & Tsai, Chen-Yang (2018). A novel campus navigation APP with augmented reality and deep learning. In *2018 IEEE international conference on applied system invention* (pp. 1075–1077). IEEE.

Lin, Tsung-Yi, Maire, Michael, Belongie, Serge, Hays, James, Perona, Pietro, Ramanan, Deva, et al. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision* (pp. 740–755). Springer.

Liu, Li, Ouyang, Wanli, Wang, Xiaogang, Fieguth, Paul, Chen, Jie, Liu, Xinwang, et al. (2020). Deep learning for generic object detection: A survey. *International Journal of Computer Vision*, *128*(2), 261–318.

Lowe, David G. (1999). Object recognition from local scale-invariant features. In *Proceedings of the seventh IEEE international conference on computer vision*: *vol. 2*, (pp. 1150–1157). Ieee.

Macario Barros, Andréa, Michel, Maugan, Moline, Yoann, Corre, Gwenolé, & Carrel, Frédérick (2022). A comprehensive survey of visual SLAM algorithms. *Robotics*, [ISSN: 2218-6581] *11*(1), http://dx.doi.org/10.3390/robotics11010024, URL https://www.mdpi.com/2218-6581/11/1/24.

Matsuda, Yuji, Hoashi, Hajime, & Yanai, Keiji (2012). Recognition of multiple-food images by detecting candidate regions. In *2012 IEEE international conference on multimedia and expo* (pp. 25–30). IEEE.

Moreno-Armendáriz, Marco A, Calvo, Hiram, Duchanoy, Carlos A, Lara-Cázares, Arturo, Ramos-Diaz, Enrique, & Morales-Flores, Víctor L (2022). Deep-learning-based adaptive advertising with augmented reality. *Sensors*, *22*(1), 63.

Mur-Artal, Raul, Montiel, Jose Maria Martinez, & Tardos, Juan D. (2015). ORB-SLAM: A versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, *31*(5), 1147–1163.

Naspetti, Simona, Pierdicca, Roberto, Mandolesi, Serena, Paolanti, Marina, Frontoni, Emanuele, & Zanoli, Raffaele (2016). Automatic analysis of eye-tracking data for augmented reality applications: A prospective outlook. In *International conference on augmented reality, virtual reality and computer graphics* (pp. 217–230). Springer.

Nguyen, Minh, Tran, Huy, Le, Huy, & Yan, Wei Qi (2017). A tile based colour picture with hidden qr code for augmented reality and beyond. In *Proceedings of the 23rd ACM symposium on virtual reality software and technology* (pp. 1–4).

Nowacki, Paweł, & Woda, Marek (2020). Capabilities of arcore and arkit platforms for ar/vr applications. In *International conference on dependability and complex systems* (pp. 358–370). Springer.

Park, Kyeong-Beom, Kim, Minseok, Choi, Sung Ho, & Lee, Jae Yeol (2020). Deep learning-based smart task assistance in wearable augmented reality. *Robotics and Computer-Integrated Manufacturing*, *63*, Article 101887.

Paszke, Adam, Gross, Sam, Massa, Francisco, Lerer, Adam, Bradbury, James, Chanan, Gregory, et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*: In *Advances in neural information processing systems.vol. 32*,

Pierdicca, Roberto, Paolanti, Marina, Naspetti, Simona, Mandolesi, Serena, Zanoli, Raffaele, & Frontoni, Emanuele (2018). User-centered predictive model for improving cultural heritage augmented reality applications: An HMM-based approach for eye-tracking data. *Journal of Imaging*, *4*(8), 101.

Pierdicca, Roberto, Tonetto, Flavio, Mameli, Marco, Rosati, Riccardo, & Zingaretti, Primo (2022). Can AI replace conventional markerless tracking? A comparative performance study for mobile augmented reality based on artificial intelligence. In *Extended reality: first international conference, XR salento 2022, lecce, Italy, July 6–8, 2022, proceedings, Part II* (pp. 161–177). Springer.

Polap, Dawid, Kesik, Karolina, Ksiazek, Kamil, & Wozniak, Marcin (2017). Obstacle detection as a safety alert in augmented reality models by the use of deep learning techniques. *Sensors*, *17*(12), 2803.

Rao, Jinmeng, Qiao, Yanjun, Ren, Fu, Wang, Junxing, & Du, Qingyun (2017). A mobile outdoor augmented reality method combining deep learning object detection and spatial relationships for geovisualization. *Sensors*, *17*(9), 1951.

Redmon, Joseph, & Farhadi, Ali (2017). YOLO9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 7263–7271).

Sandler, Mark, Howard, Andrew, Zhu, Menglong, Zhmoginov, Andrey, & Chen, Liang-Chieh (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (pp. 4510–4520).

Sereno, Mickael, Wang, Xiyao, Besançon, Lonni, McGuffin, Michael J, & Isenberg, Tobias (2020). Collaborative work in augmented reality: A survey. *IEEE Transactions on Visualization and Computer Graphics*.

Shen, Lujun, Mo, Jinqing, Yang, Changsheng, Jiang, Yiquan, Ke, Liangru, Hou, Dan, et al. (2023). SurvivalPath: AR package for conducting personalized survival path mapping based on time-series survival data. *PLoS Computational Biology*, *19*(1), Article e1010830.

Stanney, Kay M, Archer, JoAnn, Skinner, Anna, Horner, Charis, Hughes, Claire, Brawand, Nicholas P, et al. (2022). Performance gains from adaptive extended reality training fueled by artificial intelligence. *The Journal of Defense Modeling and Simulation*, *19*(2), 195–218.

Subakti, Hanas, & Jiang, Jehn-Ruey (2018). Indoor augmented reality using deep learning for industry 4.0 smart factories. *vol. 2*, In *2018 IEEE 42nd annual computer software and applications conference* (pp. 63–68). IEEE.

Svensson, Jan, & Atles, Jonatan (2018). Object detection in augmented reality. *Master's Theses in Mathematical Sciences*.

Tan, Mingxing, Pang, Ruoming, & Le, Quoc V. (2020). Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition* (pp. 10781–10790).

Tanskanen, Aapo, Martinez, Asier Alcaide, Blasco, Deborah Kuperstein, & Sipiä, Laura (2019). Artificial intelligence, augmented reality and mixed reality in cultural venues. *Consolidated Assignments from Spring 2019*, 80.

Unity 0000a. About AR Foundation, Available Online, URL https://docs.unity3d.com/Packages/com.unity.xr.arfoundation@5.0/manual/index.html.

Unity 0000b. Introduction to barracuda, Available Online, URL https://docs.unity3d.com/Packages/com.unity.barracuda@1.0/manual/index.html.

Vaswani, Ashish, Shazeer, Noam, Parmar, Niki, Uszkoreit, Jakob, Jones, Llion, Gomez, Aidan N, et al. (2017). Attention is all you need. In *Advances in neural information processing systems*.

Wang, Shaohan, Zargar, Sakib Ashraf, & Yuan, Fuh-Gwo (2021). Augmented reality for enhanced visual inspection through knowledge-based deep learning. *Structural Health Monitoring*, *20*(1), 426–442.

Zhao, Zhong-Qiu, Zheng, Peng, Xu, Shou-tao, & Wu, Xindong (2019). Object detection with deep learning: A review. *IEEE Transactions on Neural Networks and Learning Systems*, *30*(11), 3212–3232.