BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
SPECIALIZATION Computer Science, English

DIPLOMA THESIS

Augmented Reality Food Detection via YOLO

Supervisor
Lect. Dr. Alina Delia Calin

Author
Timofte Razvan-Mihai

2024

UNIVERSITATEA BABEŞ-BOLYAI CLUJ-NAPOCA
FACULTATEA DE MATEMATICĂ ŞI INFORMATICĂ
SPECIALIZAREA Informatica, Engleza

LUCRARE DE LICENŢĂ

Detectarea Mancarii in Realitate Augumentata cu YOLO

Conducător ştiinţific
Lect. Dr. Alina Delia Calin

Absolvent
Timofte Razvan-Mihai

2024

ABSTRACT

Food is one of the core human needs. It is an undeniable fact. Yet, despite its importance, a concerning reality persists: many individuals lack comprehensive knowledge about the nutritional content and ingredients of the food they consume. This gap in understanding can have serious repercussions on public health, leading to inadequate dietary choices, nutritional deficiencies, diet-related diseases and allergic reactions that could be easily avoidable with enough information.

In response to this challenge, this paper seeks to present a promising solution to empower consumers with instant access to comprehensive food information in a seamless, comfortable and easy-to-use way. By utilizing real-time object detection combined with augmented reality to both identify food items and overlay nutritional data in real-time, individuals can make more informed decisions about their dietary intake, regardless if they know or not what the food before them is in the first place.

We attempt this challenge with three different versions of YOLO models starting with the newest, YOLOv9 and YOLOv8n, and ending with the most computationally affordable, YOLOv2-tiny. The results could be better, as could the dataset, but these both will be revisited in the future for various improvements, as the current dataset could be considered more of a 'lite' version of the final set.

Contents

Chapter 1

Introduction

As of 29th of September 2023, there are at least 2 billion people worldwide who are affected by obesity, and it is projected that 38% of the adult population will be overweight and 20% obese before 2030 [17].

Diet is one of the most impactful factors on calories, and fast food, which is large in calories and little in nutritional value, has become overly popular. A popular reason this happens is because a significant portion of the population lacks comprehensive knowledge regarding this dichotomy present in nutrition.

Obesity and weight aren't the only issues when it comes to food, however. Food allergies affect a significant portion of the population, with prevalence rates varying by region and demographic factors. Common food allergens include soy, peanuts, and fish. Food allergies also appear to be increasing in some regions [24].

Food labeling regulations have proven useful in providing details about packaged foods, of course [2]. This helps both with informing people of the nutritional values of the food they are about to consume and of any allergens it may consume. Labeling, however, is usually limited to prepackaged food, and overlooks fresh items.

The Internet plays a pivotal role in helping people access information about the food they want to consume. With a vast array of online resources, individuals can easily search for recipes, details about calories or other nutritional values, and dietary advice to make informed choices.

Unfortunately, despite the abundance of information available online, individuals may not always know the name or exact ingredients of a dish. It can be time-consuming or cumbersome to input keywords and phrases related to the food they are seeking, especially if they are unsure of terminology or spelling, as could happen in a touristic context.

Furthermore, the process of searching for food information on the Internet may not always yield accurate or reliable results. Misleading or outdated information, conflicting advice, biased sources can all complicate the search process.

As technology grows, however, so do our options for problem-solving. Modern AI models have reached the point they can rival human image recognition capabilities [23], which, when paired with the quality of cameras present in day-to-day devices such as smartphones, leads to the potential for a system that combines both for a more intuitive and efficient approach to accessing food information while also satisfying the human desire for quick and convenient access to information.

The object detectors leveraged for the task are from the You Only Look Once family, the version nine, eight, and three. We create and augment a handmade dataset of select, common foods and experiment with the models' multiple configurations, whether pre-trained on the COCO dataset or untrained.

We also develop a mobile application with the help of the Unity engine to allow the integration of augmented reality with the detection model. This ensures availability and affordability, as billions of smartphones are capable of augmented reality and the YOLO architecture is tailored towards both speed and accuracy.

1.1 Objectives

This paper aims to study the design, implementation and performance of a system that seeks to accomplish precisely what was mentioned prior: a bridge between individuals' lack of knowledge about specific foods and their desire for quick and convenient access to information, done by leveraging technologies such as real-time object detection and augmented reality.

Thus, the model in question has to be highly effective, achieving a good balance between speed and accuracy, lest it be too slow and cumbersome for users or too inaccurate and ultimately untrustworthy. For this reason, the YOLOv9 was initially chosen, thanks to its state-of-the-art performance and innovative design that allows it to run in reasonable time within an acceptable threshold of accuracy. We also test the YOLOv8n and the YOLOv2-tiny as alternatives for lower-end smartphones, with the latter being the most optimal amongst the three. As there are an approximate seven billion smartphone mobile network subscriptions [28], over 1 billion of which engage in the use of augmented reality in an active manner [5], the proposed application would be readily available to most consumers.

Chapter 2

Existing Solutions

This section will address the various applications that have already been done in regards to the detection of food items, along with their performance and features. It bears mentioning, however, that our approach differs greatly in the manner in which the relevant information is shown to the user, namely via Augmented Reality, as opposed to all of the sources cited in this section.

2.1 Real-time Food Detection by YOLOv2

Article [25] is one such application in the object detection sphere which focuses on food items, published in 2019. The model which it is based upon is a DCNN (Deep Convolutional Neural Network) based on MobileNet, adapted with the one-stage detection framework YOLOv2, and trained on the UECFood100 and UECFood256 datasets. The application sports two different input modes: real-time, in which the camera frames serve as input, while the other gives users the option to select a photo that is then run through the food detection model and outputs an analysis of the result.

The datasets mentioned are primarily based around asian foods, specifically Japanese cuisine. This leads to less usefulness for the average user unfamiliar with such food, but also presents more options for those who know of them.

In terms of strategy, the model makes use of depthwise separable convolution, which allows for more lightweight DCNNs. Setting the filters of the convolutions to a fixed size of 3x3, the number of computations is divided by at least 8, therefore speeding up calculations heavily in exchange for accuracy, allowing for actual real-time inference.

Seeing as the input is supposed to accept real-time camera frames, which will rarely be taken in optimal conditions, overfitting is a valid concern. Measures were taken to combat overfitting, such as processing half of the data with a data augmen-

tation technique (blur, horizontal flip, etc.), which also lead to more training time.

The architecture of the model contains 30 layers of 3.5 million parameters overall, each followed by a batch norm and a ReLU nonlinearity, save for the last layer (fully-connected) that feeds into a YOLO output layer. Each input image is divided into an (S, S) grid, after which N bounding boxes are predicted alongside scores.

Shown in figure 2.1 is the performance of the app, which does not rely on any external servers for inference, tested on a OnePlus 5 and a Google Pixel 2 in both local and real time. Owing to the lightweight design provided, the average time taken by the computer to complete the task (per image) is 75ms. In terms of accuracy, the mAP (Mean Average Precision) observed is 75.05% for UECFood256 and 76.36% for UECFood100.

Inference time	CPU time	15ms
	Wall clock time	75ms
DCNN model size		8.1MB
Runtime memory		242.2MB

Figure 2.1: Application performance

2.2 Real-time Food Object Detection using MobileNetV2 & InceptionV2

Another application described in [18] and published in 2020, this one tailored more towards Indian cuisine as opposed to universal. Thus, the dataset has also changed, each image taken from various online sources and some manually labelled. Noteworthy is that the UECFood256 dataset has been used here, also. As before, multiple data augmentation techniques were utilized to prevent the overfitting of the data.

As mentioned in the analysis of the previous paper, UECFood256 is primarly composed of Japanese foods, and therefore less suitable for global use, but more practical for those within the region.

In relation to the object detection setup, it stands to reason that a SSD model has been chosen, as it is more tailored towards real-time object detection as opposed to RCNNs (Region-based Convolutional Neural Network), which are much more computationally-expensive, preferring accuracy over speed of inference.

The two SSD-based CNN architectures mentioned and built in the article are MobileNetV2 and InceptionV2, along with an attempt at Resnet50 which had yielded unsatisfactory results. The main difference between the two chosen architectures

is that MobileNet uses depthwise separable convolution, leading to less parameters, while Inception uses standard convolution, leading to increased performance. Overall, the highest performance was provided by InceptionV2, though at a speed of 14 fps, as seen in figure 2.2.

	MobilenetV2	InceptionV2
mAP	0.59	0.738
mAP(Large)	0.632	0.769
mAP(Medium)	0.308	0.575
mAP(Small)	-1	0.612
50IOU	0.909	0.9645
75IOU	0.804	0.8627
Recall	0.524	0.792
Classification Loss	1.98	1.284
Localization Loss	0.293	0.245
Regularization Loss	0.63	0.626
Total Loss	2.903	2.15

Figure 2.2: Result metrics of MobileNetv2 and Inceptionv2 architectures

Training consisted of using a pre-trained checkpoint, which had been trained for the COCO dataset, for 80000 steps for Inception and 60000 steps in the case of MobileNet. Both models were trained until convergence, meaning that the total loss had a variance of less than $1e^{-}3$ for at least 10 consecutive epochs.

2.3 Real-time Food Object Detection via Shallow CNN

In the paper [29], published in 2017, a different dataset is used: FruitVeg-81, with only vegetable and fruit classes. It also bears mentioning that the user must confirm the result before any other desired information is shown.

FruitVeg-81, as the name suggests, is a dataset composed entirely of 81 different kinds of fruits and vegetables, with 15373 images total. It contains three levels of labels, with the first being the fruits / vegetables themselves, the second being the same fruits / vegetables but sporting a different appearance (e.g. golden apples) and the third is packaging types.

In this case, the CNN used is shallow, containing only a few layers, being geared more towards minimum complexity and good accuracy. It works with images pre-processed to a size of 56x56 pixels and consists of three convolutional layers, a pooling layer between each pair. Similar to the first such application discussed, the layer that follows is a fully-connected one, which is then in turn followed by a Softmax classification layer.

The images within the dataset are augmented via common methods, such as cropping, rotating, and mirroring. An 82nd class, made with images taken from ImageNet, is added, a 'garbage' class for non-food products for the sake of reducing false positives.

Statistics seen in figure 2.3 suggest that image quality differs based on the phone models. Furthermore, the addition of the new 'garbage' class seems to also lower accuracy. As can be seen, best top-1 accuracy (which represents the percentage that the guess was correct) was 76.14% for the 81-model variant and 71.74% for the 82-model variant. Another detail to note is that the registered speed of the recognition system is in the 10 fps range.

Model	Baseline					Non-food				
	Top-1	Top-2	Top-3	Top-4	Top-5	Top-1	Top-2	Top-3	Top-4	Top-5
Samsung Galaxy S3	72.99	84.34	88.93	92.27	94.58	64.45	80.72	87.55	91.15	93.39
Samsung Galaxy S5	76.14	86.66	90.82	92.46	93.82	71.74	84.30	89.11	92.69	94.52
HTC	71.99	84.06	87.86	89.77	91.03	60.17	77.89	84.49	89.00	91.77
HTC One	65.28	76.95	82.84	85.65	88.40	52.30	71.32	80.41	85.50	88.14
Motorola G	62.43	73.72	78.22	81.00	82.94	53.70	68.54	75.17	79.98	84.21
Avg	69.77	81.15	85.72	88.23	90.16	60.47	76.53	83.35	87.66	90.41

Figure 2.3: Result metrics of shallow CNN architecture

Chapter 3

Theoretical Background

3.1 Convolutional Neural Networks

First proposed in 1995 in Yann LeCun's paper [13], wherein LeCun and his colleagues introduced the LeNet-5 architecture, which showed promising results in the handwritten digit recognition tasks. CNNs are a class of deep learning models designed primarily for processing data structured in grids, most commonly images, and are characterized by their ability to adaptively learn spatial hierarchies of features directly from raw input data.

3.1.1 CNN Layered Architecture

CNNs consist of layers like convolutional layers, fully connected layers and also pooling layers (seen in Fig. 3.1) which work in tandem to extract increasingly abstract features. Likewise seen in the figure is the feed-forward nature of CNNs, which refers to the mono-directional flow of data through the network, from input to output, without feedback loops. This type of dataflow also facilitates parallel computation, which enables CNNs to take advantage of modern computing hardware like GPUs for quicker training and inference.

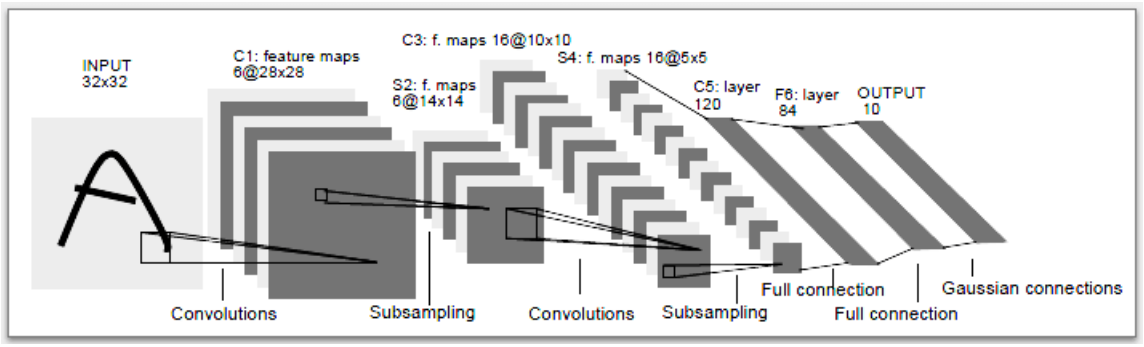


Figure 3.1: Architecture of a common CNN model by LeCun [13]

Convolution Layer

As the name suggests, the fundamental operation of CNNs is convolution (also known as feature detection), which consists of applying a filter (kernel) over an input. The mentioned filters are often spatially small. Element-wise multiplication is computed between the filter and the superimposed part of the input, and then an activation function is applied over the added obtained values, a popular option being ReLU (Recitifed Linear Unit) because it allows non-linearly scaling these values into more meaningful representations.

Based on the kernel's values, different features can be extracted from the input, starting from edges, corners and other simple shapes and patterns, as seen in Fig. 3.2. Each convolutional layer can have multiple kernels that produce multiple feature maps of their own. The number of channels also impacts the number of feature maps, usually tripling the amount in the case of an RGB image.

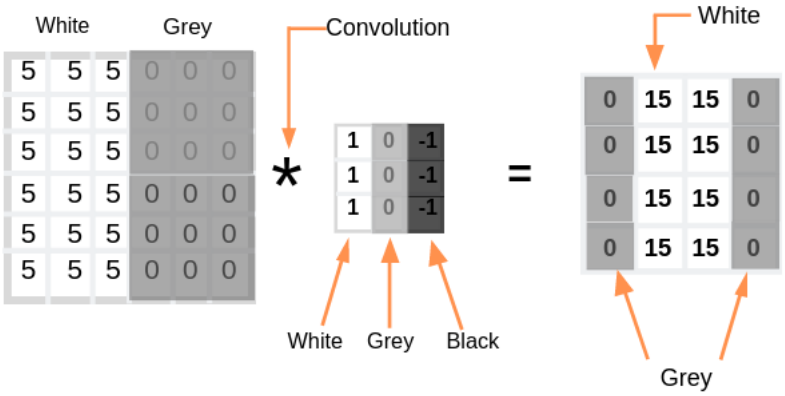


Figure 3.2: Horizontal edge feature extraction via convolution operation ([8])

Pooling Layer

Pooling layers are used to downsample the achieved feature maps and keep only the most important parts, with the rest being discarded. These also help against overfitting and in speeding up calculations for later layers due to the smaller image size.

Most commonly encountered are average pooling and max pooling. In these layers, a kernel is slid over the feature maps, with only the maximum pixel value in the superimposed region being saved to the feature map, respectively the average pixel value.

Fully-Connected Layer

The fully-connected layers are part of the classification component at the end of the CNN. It is called as such because every neuron inside of a fully-connected layer has a connection to every neuron of the previous layer.

For input, these layers take the highest level of the abstracted features, rather than the raw input pixels. As for output, which is given by the last layer, it is of size 1 x 1 x n, where n is the number of classes, and each value represents the class score computed by the layer.

3.1.2 Information Bottleneck Principle

The Information Bottleneck Principle is a fundamental challenge when it comes to deep learning, and refers to the fact that data is lost as it passes through successive layers of a network, and the potential for more loss only increases the deeper it goes, as shown in Eq. 3.1 below:

$$I(X, X) >= I(X, f_\theta(X)) >= I(X, g_\phi(f_\theta(X))), \tag{3.1}$$

where I represents mutual information and f and g are transformation functions with parameters θ and ϕ, respectively.

Essentially, this makes the possibility of using incomplete information during the training process a real concern, which would result in unreliable gradients and sub-standard convergence. Thus, the network's ability to accurately predict the target is compromised.

3.1.3 Deep Supervision

It is a fact that, by increasing a network's size in terms of depth and width (levels and units at each level, respectively), a CNN's accuracy can be improved. The disadvantage of this, though, is that bigger CNNs have more parameters, which leads to slower back-propagation convergence and also increases the chances of overfitting. To solve this problem while also retaining the accuracy of deeper CNNs, the concept of deep supervision has been explored in the paper [14];

The concept is the following. In traditional neural networks focused on object detection, there is usually a single output layer that predicts the presence and location of an object in the input image. Deep supervision involves additional supervision points at various intermediate layers in the network, seen in Fig. 3.3. Each intermediate layer captures a different level of features from the input image, and at each of these intermediate layers, the network may make predictions about the presence

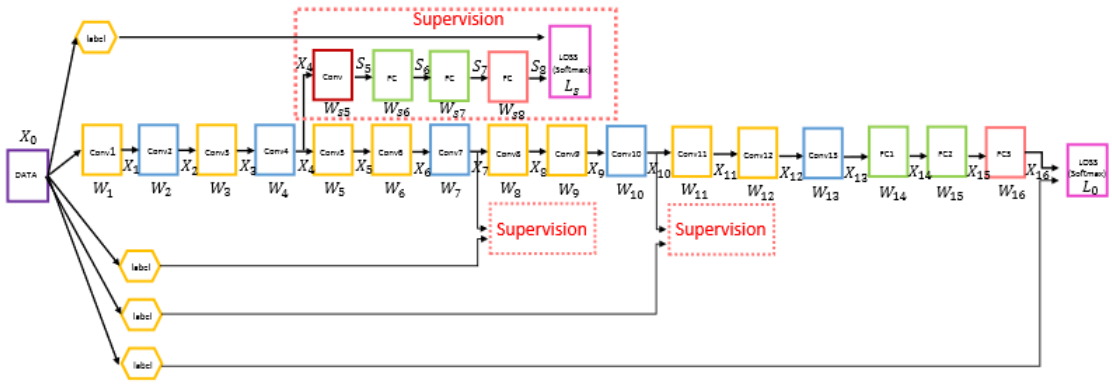


Figure 3.3: Illustration from paper [31], where a CNN with 13 convolution layers is presented. Extra supervision loss branches can be identified by the dashed red boxes.

and location of the object.

Therefore, instead of relying solely on the final output of the last layer, deep supervision introduces additional supervision signals. These ensure that gradients flow through the entire depth of the network, addressing the problem of vanishing gradients. Deep supervision also provides constraints on the learning process, preventing overfitting and guiding the network towards more generalized features.

3.2 Real-Time Object Detection

Real-time object detection is a task in computer vision that aims to identify and locate objects of interest within frames in real-time. This has many applications across various domains, including autonomous driving, robotics, surveillance, augmented reality, and so on.

3.2.1 Faster R-CNN

Described in paper [22], Faster Region-Convolutional Neural Network is a two-stage object detection framework works based on the Region Proposal Networks (RPN) concept, which takes an image of any size as input and outputs a set of rectangular oject proposals, each with an objectness score.

The RPN (Fig. 3.4), as mentioned, generates candidate bounding boxes which potentially contain objects of interest. It works with feature maps extracted from a CNN backbone, typically a pre-trained network like MobileNet. The RPN achieves its goal by sliding a small network, typically a set of convolutional layers, over the feature map, and then predicting the bounding box proposals at each spatial location.

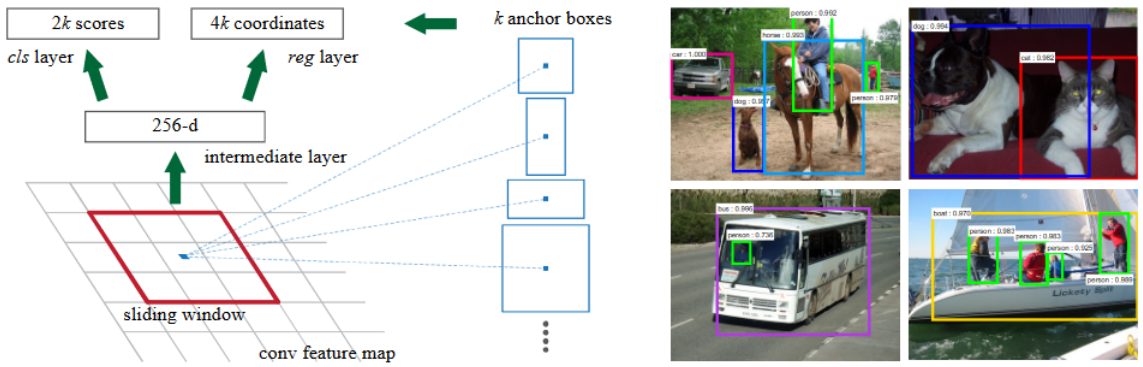


Figure 3.4: Seen on the left is the RPN. On the right are four detections using Region Proposal Network proposals. Image sourced from: [22].

It makes use of Region of Interest (RoI) pooling or RoI align to extract feature vectors of fixed size from each region proposal, enabling efficient processing and inputs of varying sizes.

What allows the model such accuracy, however, is also what gives it less speed compared to one-stage detectors. Having two separate stages means an increased computational overhead and latency. Its complexity also makes it more challenging to train and deploy, especially on resource-constrained devices or in real-time applications where resources are limited.

3.2.2 EfficientDet

EfficientDet [27] represets a breakthrough in object detection architecture, inspired by the EfficientNet model [26].

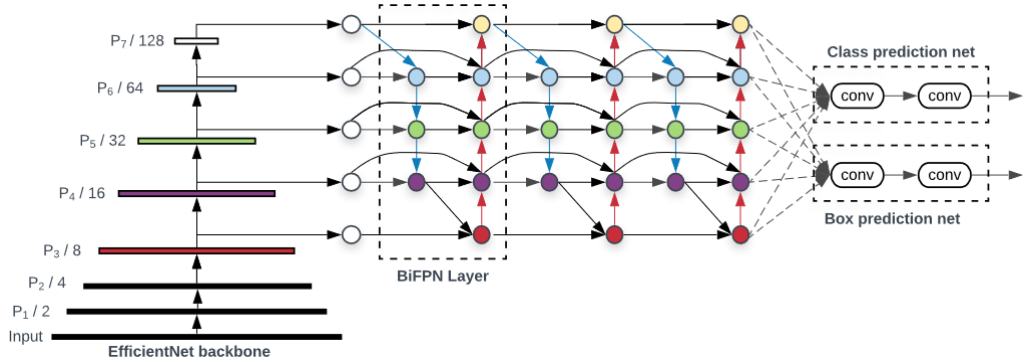


Figure 3.5: The components of the EfficientDet architecture. On the left is the EfficientNet backbone. In the middle is the BiFPN layer as the feature network. On the right is the shared class/box prediction network. Both of the latter are repeated multiple times based on resource constraints. Image sourced from: [27].

The backbone network is a lightweight one, such as EfficientNet, used to extract hierarchical features from input images (Fig. 3.5(a)). The backbone network is crucial in capturing informative representations while minimizing overhead in terms

of computation.

Bidirectional Feature Pyramid Network (BiFPN) is a novel feature fusion incorporated by EfficientDet, combining multi-scale features from different network layers (Fig 3.5(b)). BiFPN enhances feature reuse and promotes the flow of information across the network, allowing for accurate detection at various scales. EfficientDet also employs separate prediction heads for object classification and bounding box regression (Fig. 3.5(c)).

In terms of limitations, EfficientDet suffers from an increased complexity in terms of model architecture, primarily due to the BiFPN. This could, in turn, lead to a more difficult training and deployment similar to Faster R-CNN. Regardless, EfficientDet sports higher efficiency compared to Faster R-CNN, which is by design, as the first is focused on efficiency and the second on accuracy.

3.2.3 You Only Look Once

You Only Look Once (YOLO) [19] was revolutionary in the object detection scene for its single-stage approach that directly predicts bounding boxes and class probabilities from the entire image in a single pass. Unlike traditional two-stage detectors like Faster R-CNN, YOLO is more suited for real-time applications precisely because of its efficiency.

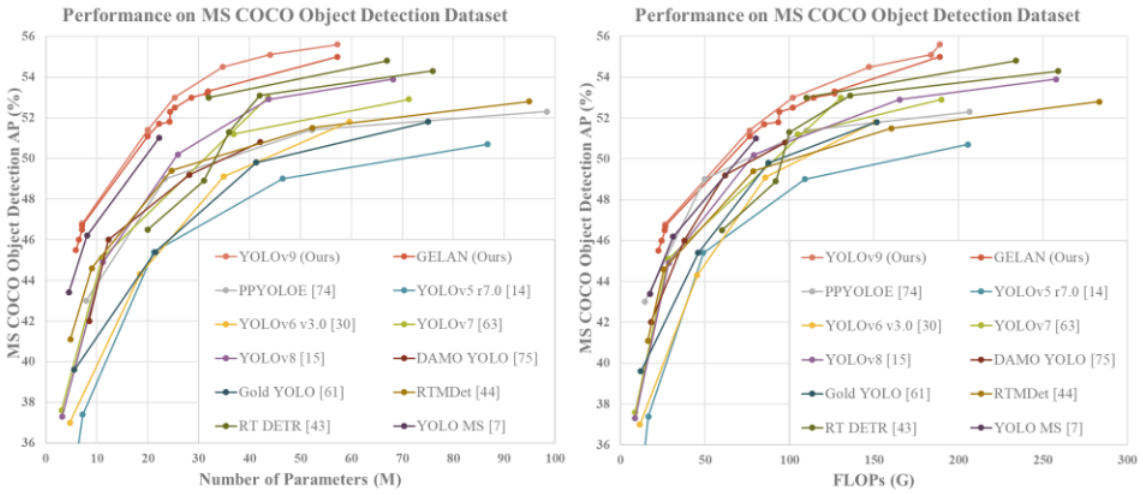


Figure 3.6: Comparison of current state-of-the-art real-time object detectors. Image sourced from: [30].

As mentioned above, YOLO uses a single-stage architecture, where a CNN predicts both bounding boxes and class probabilities without the need to separate the two tasks. It does this by dividing the input image into a grid of cells and running prediction for objects contained within each cell. This grid-based approach allows it to handle objects of multiple sizes and aspect ratios, helped along by the presence of a feature pyramid network (FPN).

In terms of architecture, the model has had many advancements since its conception, and thus its features have changed accordingly. YOLOv2, for example, made use of Darknet-19 as a backbone that extracts features from the input [20]. YOLOv3, its successor, raised the bar by utilizing Darknet-53 [21].

At the time of writing, the most recent and performant YOLO model is currently YOLOv9 [30]. It not only uses the advances of its predecessor models (with YOLOv7 as the basis for development), but also focuses on solving the issue of slow convergence or poor convergence results through in-depth analysis of the information bottleneck principle. As can be seen in Fig. 3.7, these changes are highly advantageous for YOLOv9 in comparison to other, prior versions of the model, in terms of accuracy.

3.3 Augmented Reality

Augmented Reality (AR), coined as such by Thomas Preston Caudell in 1992, who'd developed one such AR application for industrial use, has many definitions. There tends to be a confusion regarding the exact details of what AR entails versus something like Virtual Reality (VR).

It is theorized that there are four different environments that, starting from the real world, lead to a virtual world [3]. The first is the Real Environment (RE) which is the one we live in. The second, is Augmented Reality, in which our physical reality is augmented to contain virtual elements. Next is Augmented Virtuality (AV), which is the inverse of AR in that it starts as an entirely virtual reality and instead also contains real elements. Finally, there is Virtual Reality, which is a completely synthetic world.

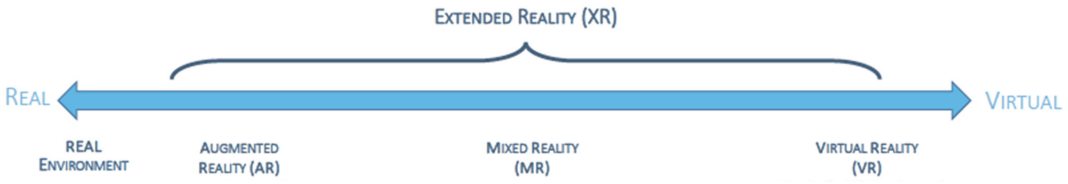


Figure 3.7: The so-called Reality-Virtuality Continuum schematic. Image sourced from: [4].

AR is, at its core, a transformative technology. It overlays digital information and virtual objects onto the real-world environment, such as images, videos, and even 3D models. AR also accurately tracks the user's position and orientation in the environment in real-time, achieved through techniques like Simultaneous Localization and Mapping (SLAM). Digital representations of the physical environment are generated through spatial mapping algorithms, while tracking algorithms continously update the user's position and orientation relative to this map.

In terms of AR technologies, there are four main ones: marker-based AR, markerless AR, projection-based AR and recognition-based AR.

Recognition-based AR

Also known as AR based on overlaps, recognition-based AR uses computer vision algorithms to recognize and track real-world objects or features. By analyzing the characteristics of objects such as texture, color, and shape, the system can identify and augment them with digital content.

Chapter 4

Augmented Food Detection Application

4.1 Requirements

The purpose of this subsection is to describe the functionalities and features of the software application.

This application aims to provide the user with a seamless, accurate and comfortable method of detecting food via their device, whether a headset or a mobile phone, in two stages. The first stage is continuous, in which the application constantly scans for food items taken from the live camera feed of the device. The second stage is triggered by the user, after which the nutritional information of the detected items will be shown in an AR manner. The AR function is meant to allow the viewing of multiple food items' information without the need to switch screens or flood the display with text.

Perform real-time object detection

The application receives live input from the camera continuously. The live images will be fed to the object detector for the sake of displaying the resulting class via bounding box whenever the detection process is done.

Retrieve nutritional information

The user is able to request information regarding the scanned food items. This request will be sent to an external API, which will retrieve figures such as approximate calories, allergens, protein, fat, sugars and others.

Display in augmented reality

Upon the retrieval of the aforementioned information at the behest of the user, the application will estimate the spatial coordinates of the detected object in the real world and display its details directly above its location.

4.2 Implementation

This subsection contains the technologies with which the application is developed, trained and tested, both in terms of software and hardware.

4.2.1 Hardware

During the first iteration of training for the object detection model, the publicly available T4 GPU offered by Google Colab is used. However, due to the limitations imposed by Google Colab, a switch is made to using our local resources for the sake of training. Thus, the rest of the iterations were done on a desktop system containing an NVIDIA GeForce RTX 4060 Ti GPU with 16GB of VRAM GDDR6, 16GB DDR4 RAM 2133 MHz, and an AMD R5 2600 CPU.

Testing the inference process of the object detectors was done on the same desktop system mentioned previously. The final mobile application was tested on an A50s with 6GB of RAM and an A54s with 8GB RAM.

4.2.2 Software

The YOLOv9 object detection model is trained via the notebook provided by the developers of the YOLOv9 model. Similarly, the YOLOv8n model is trained via ultralytics' provided library and notebooks. Roboflow is used for the annotation and augmentation of the dataset images. The model is then exported into ONNX format for deployment on Unity 2022 and above. CUDA v12.1 is used in order to allow training via personal GPU, along with cuDNN v8.9.7. For the final version, we use Unity 2020.

In terms of Unity libraries, the core ones utilized are ARFoundation v4.0.9 along with ARCore v4.0.9 and ARKit v4.0.9. Sentis was attempted at first with version 1.2.0-exp2 and the 1.4.0-pre3, but neither were properly optimized for NMS and phone deployment. Therefore, we instead use Barracuda 1.0.4, as later versions also have issues. Android Logcat of any version is used in order to debug the application while utilizing the mobile application.

For training the newer models we made use of Python 3.8, Tensorflow 2.X, ONNX 1.9, tf2onnx 1.8.4, opencv-python, scipy, torch, tensorboard, comet, pandas, torchvi-

sion, numpy and other libraries for the sake of plotting, logging, exporting and deploying the models. For training and converting the older model, we made use of Tensorflow 1.15, Python 3.7, ONNX 1.5, and tf2onnx 1.5.4.

4.3 The Object Detector

4.3.1 Model

YOLOv9

The initial object detection model proposed is the YOLOv9 (You Only Look Once) [30]. It is a Deep Convolutional Neural Network (DCNN) that seeks to improve upon all previous YOLO versions by not only building upon previous iterations (YOLOv7) but also solve the issue of poor convergence or slow convergence that take place due to information bottleneck.

At its basis, YOLOv9 is composed of a backbone network, a set of anchor boxes and a detection head. The network is responsible for the feature extraction part, the anchor boxes serve as reference points for the model to predict the location and size of objects, while the detection head predicts the class probabilities and bounding boxes.

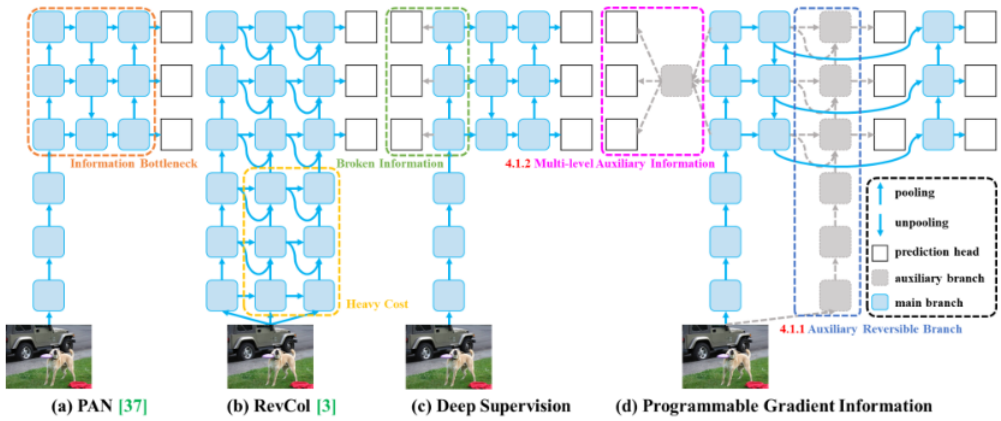


Figure 4.1: From left to right: Path Aggregation Network (PAN) [15], Reversible Columns (RevCol) [7], conventional deep supervision, and the Programmable Gradient Information (PGI) architecture proposed in the YOLOv9 paper, from which this image is taken: [30].

YOLOv9 is the first model to implement a Programmable Gradient Information (PGI) framework (4.1), which is made to prevent data loss and ensure accurate gradient updates, and the Generalized Efficient Layer Aggregation Network (GELAN) (Fig. 4.2), whose purpose is the optimization of lightweight models with gradient path planning. It replaced ELAN in YOLOv9 with GELAN using CSPNet blocks

with planned RepConv taking the place of computational blocks. The downsampling module is simplified and the anchor-free prediction head is optimized. For the auxiliary loss part of PGI, YOLOv7's auxiliary head setting is completely followed.

The PGI framework consists of three main components: a main branch, an auxiliary reverse branch, and multi-level auxiliary information. Seen in Fig. 4.1(d) is the aforementioned architecture, in comparison to other related networks and methods.

The main branch is the primary component used during the inference process as seen in Fig. 4.1 and does not require any other inference cost.

The auxiliary reverse branch is an addition meant to resolve the problems that arise with the deepening of DCNNs discussed in the theoretical section 3.1.2, meaning information bottleneck, where the loss function generates unreliable gradients. It does this by helping ensure that the loss function provides accurate guidance, mapping data to targets, thus avoiding false correlations based on incomplete features.

Seen in Fig. 4.1(d)(4.1.1) is the role of the auxiliary reverse branch, which is treated as an expansion of the deep supervision branch, done to avoid directly using reversible architecture for the main branch, which would've increased inference time. As such, by leveraging the auxiliary reversible branch, the main branch can now extract more effective features in shallow networks, unlike traditional reversible architectures which would struggle with such networks. The main branch is not forced to retain complete original information, and instead it is updated by generating useful gradients through the auxiliary supervision mechanism.

Multi-level auxiliary information (Fig. 4.1(d)(4.1.2)) targets error accumulation found in deep supervision. Error accumulation is, as the name suggests, the amassing of small errors and noise on an epoch by epoch basis during training, which leads to inaccuracy.

The concept is to insert an integration network between the main branch and the hierarchical layers of the feature pyramid. This integration network combines gradients from different prediction heads, ensuring that the main branch receives information about all target objects, and prevents specific object information from dominating the features learned by the main branch's pyramid hierarchy.

In summary, multi-level auxiliary information ensures that all target objects' information is considered when updating parameters in the main branch, and therefore alleviates issues with broken or incomplete information in deep supervision, while also allowing for more effective learning and prediction across various object sizes.

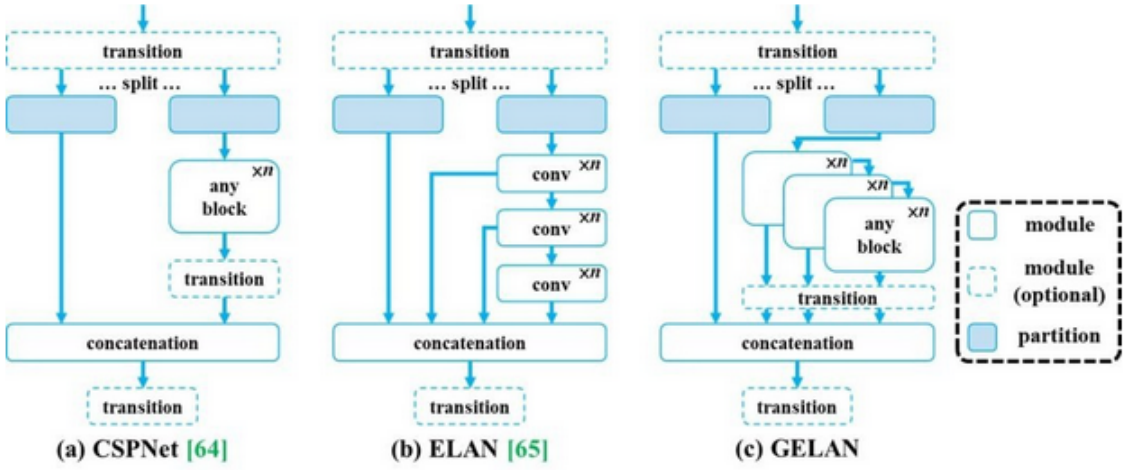


Figure 4.2: GELAN architecture in comparison to CSPNet and ELAN. GELAN allows users to choose appropriate computational blocks arbitrarily for different detection devices by considering the number of parameters, computational complexity, inference speed and accuracy, all at once. Image sourced from: [30].

YOLOv8

In spite of the performance of YOLOv9, however, it is not yet viable for the average smartphone to run, as it simply requires too much memory. Not only that, but the creators of the model have not, as of the time of writing this paper, implemented a nano version of the model, which is essentially a more light-weight version of the YOLOv9 model that contains less parameters.

As such, the Ultralytics YOLOv8 [11] was picked as the next attempt at building a functional mobile app for a mid-range smartphone. It is the prior version of YOLOv9, based on YOLOv5, with various modifications in terms of model scaling and architecture tweaks.

For the sake of our experiment, we used the nano version of the YOLOv8 model, the YOLOv8n, with 3.2 million parameters as opposed to YOLOv9's 25.3 million, and trained it on the same dataset as the YOLOv9 model. YOLOv8 supports multiple vision AI tasks, such as detection, pose estimation, segmentation, classification and tracking.

The YOLOv8 consists of three main parts, similar to the YOLOv9 one: the backbone, the neck and the head. It is highly similar to the structure of YOLOv5, and is depicted in 4.3.

For the backbone, the YOLOv8 model uses a custom CSP-Darknet53 structure, which is a CNN that is 53 layers deep as its name suggests. It partitions the feature map of the base layer and then merges the two partitions through a cross-stage hierarchy for better gradient flow.

The neck of the model connects the head and the backbone, where Spatial Pyra-

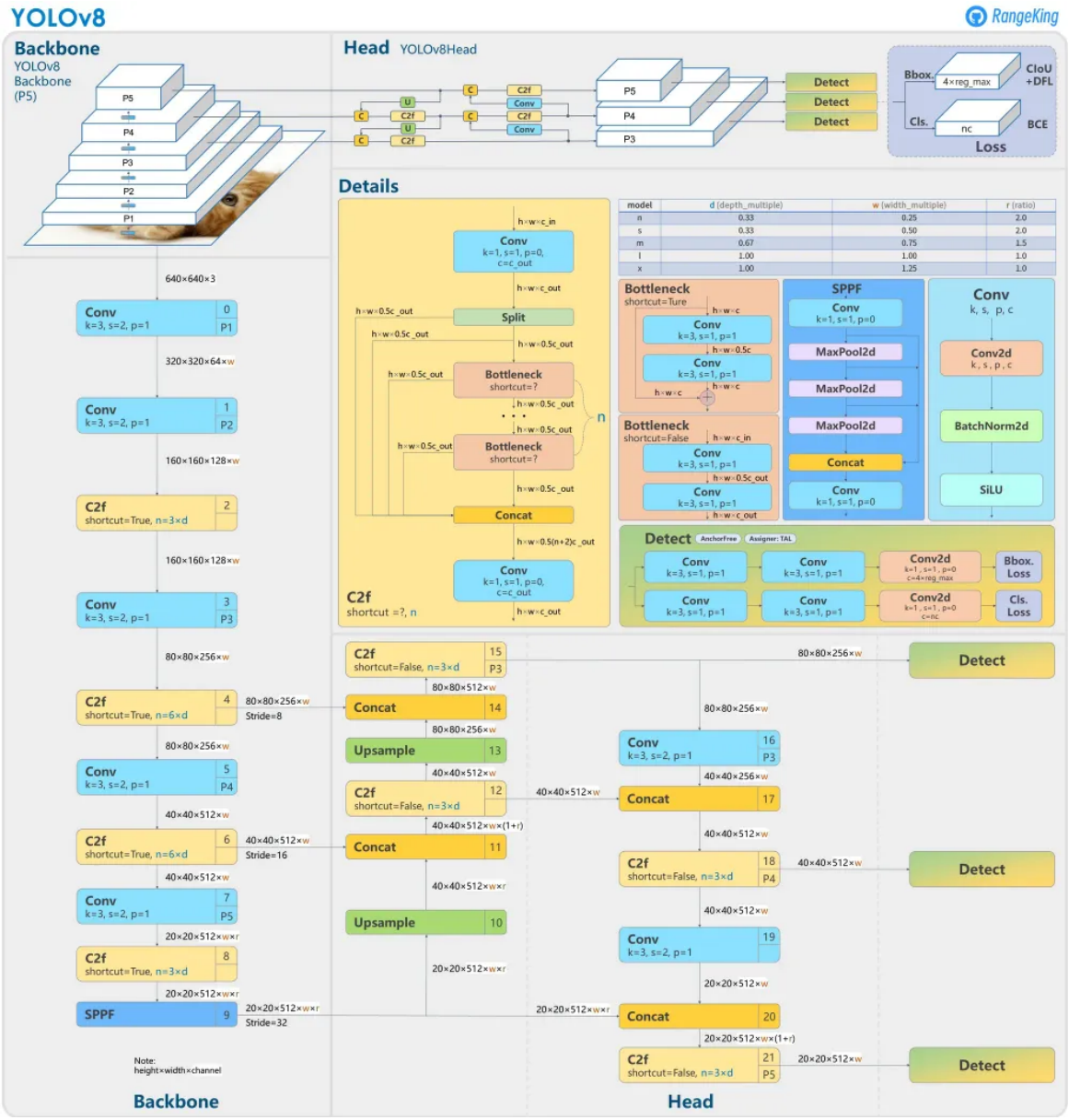


Figure 4.3: The structure of the model is depicted in the image above. Source: [10]

mid Pooling - Fast (SPPF) and a Path Aggregation Network. The latter ensures information flow across different spatial resolutions, thus leading to the model's capability to capture multi-scale features efficiently.

The head is the one responsible for generating the final output. It utilizes a modified version of the YOLO head, incorporating dynamic anchor assignment and a novel IoU (Intersection over Union) loss function.

In terms of features, it is especially notable that YOLOv8 is anchor-free, meaning that it will predict the center of an object instead of the offset from a known anchor box. This leads to a speed-up of Non-Maxima Suppression (NMS) which is what ensures that there are no overlapping detections and ensure that detections are relevant. The C2f module was also replaced from YOLOv5 with the C3 one, which, as

(a) Shear (b) Horizontal Flip (c) Hue Shift (d) Simple

Figure 4.4: Various data augmentation techniques used.

opposed to concatenating the outputs of the bottleneck, uses only the last output.

4.3.2 Data

The dataset is made with images taken from multiple sources and annotated by hand, such as the Food-101 dataset [6], the UECFood-100 dataset [16] and others such as Open Images Dataset v7 from Google [12]. Only five of many classes were kept for the initial stages of training, each class containing 204 images, and represent some of the most common foods encountered: Pizza, Hamburger, Hotdog, Donuts, and Chicken Wings. The number is kept rather low thanks to the benefits of transfer learning.

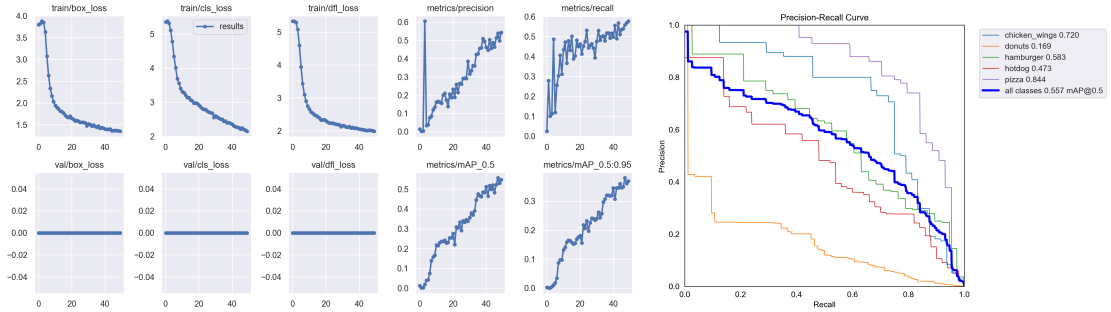
Multiple data augmentation techniques are applied on the dataset, such as horizontal flip, crop, rotation, shear and exposure, some of them seen in Fig. 4.4. Hue changes are also applied in very minimal fashion, as color is a crucial aspect towards correct identification of food. This is done via Roboflow [9]. YOLOv9 also contains hyperparameters related to data augmentation that are used in multiple experiments, though kept to a minimum so as not to conflict with the pre-existing augmentation.

All the mentioned techniques are meant to better allow the user to detect food items regardless of environment, while also not sacrificing the precision of the detection in a significant manner. Unlike normal object detection, food detection is more sensible to lighting, as some foods differ simply based on color or even a single ingredient. Therefore, data augmentation is done carefully and with only select techniques.

4.3.3 Training

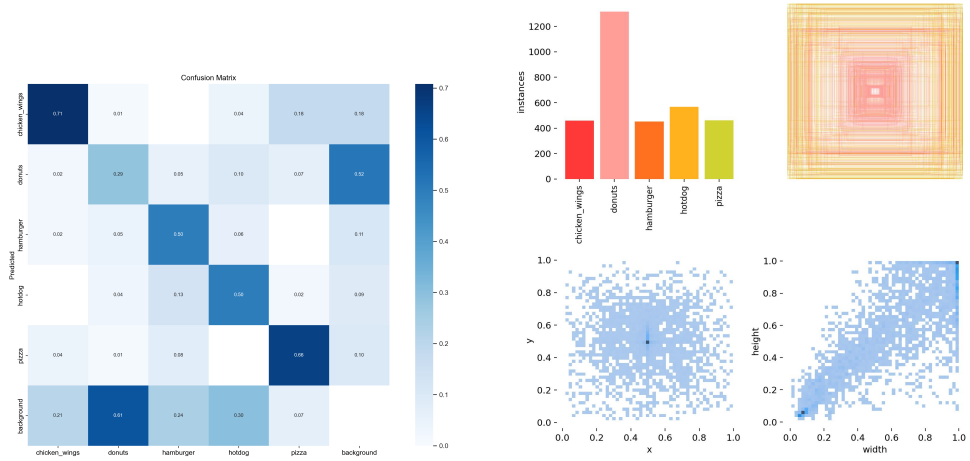
Training-wise, both the untrained YOLOv9 model the pre-trained version whose weights were adjusted for the COCO dataset are put to the test, along with similar experiments for YOLOv8. The logic for using the pre-trained version despite the seeming discrepancy between domains is that COCO already contains some food

labels, such as apples, sandwiches, and pizza. Therefore, inference training should be possible and require a smaller food dataset. On the other hand, precisely because of the abundance of labels other than food in the COCO dataset, precision might suffer, and so the untrained variant is also tested.



(a) Results. (b) PR Curve.

Figure 4.5: YOLOV9: All 5 labels included, pre-trained weights.

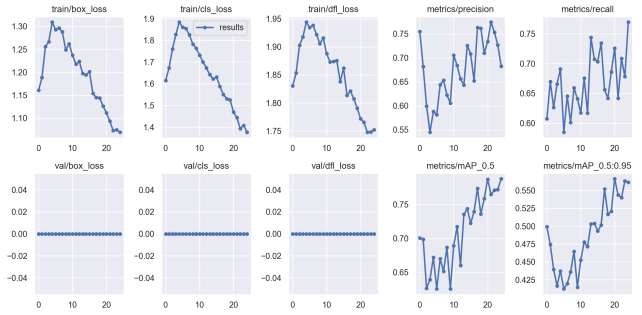


(a) Confusion matrix. (b) Labels instances.

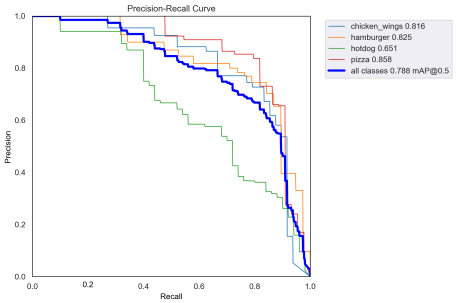
Figure 4.6: YOLOV9: On the left (a) is the confusion matrix, which shows an evident problem when it comes to predicting background instead of the donuts class. On the right (b) is the graph of label instances, where it can be seen that the results are slightly skewed because of the many instances of donuts per image.

In terms of hyperparameters, not much is changed from the recommended defaults, except for the data augmentation hypermarameters, which are completely re-purposed. The initial learning rate value is 0.01 for Stochastic Gradient Descent, while the scaling factor is 0.01, which is the final learning rate in a OneCycleLR schedule. The momentum for the SGD optimizer is 0.937, which is a common value (usually within 0.9 - 0.99). Regularization strength is low, with the weight decay hyperparameter set to 0.0005.

The warm-up phase consists of three epochs, which helps with instability prevention. Likewise, the momentum has its own warm-up, beginning at 0.8 before



(a) Results. Donuts label removed, pre-trained weights.



(b) PR Curve.

Figure 4.7: YOLOV9: Donuts label removed, pre-trained weights.

gradually rising to 0.937.

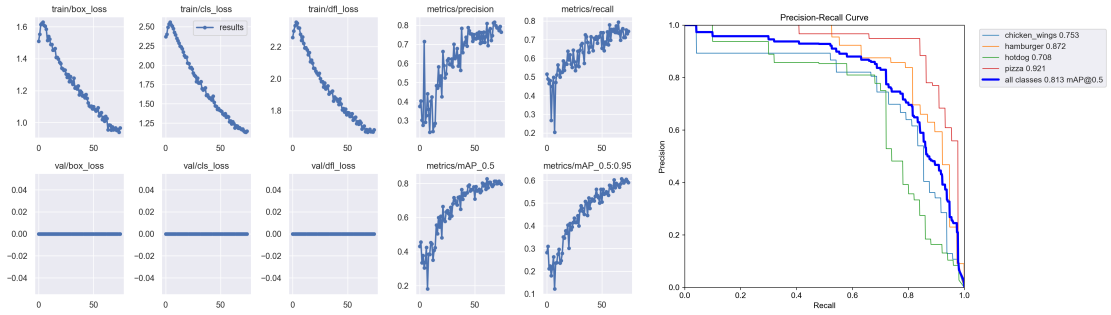
The first attempt starts with 50 epochs using the pre-trained COCO weights, using 8 workers and 8 batches, on local GPU. This results in a mean Average Precision (mAP) at Intersection over Union (IoU) threshold 0.5 of, at best, 0.55. The results, also seen in Fig. 4.5, are not desirable, and a look at the confusion matrix reveals that the predicted/actual value of the 'Donuts' class is 0.29, while the predicted 'Background' and actual 'Donuts' value is 0.61, meaning that the donuts are often predicted as being part of the background as in Fig. 4.6. On another look through the dataset, the issue becomes apparent. Since donuts can either be considered as a single object or as multiple objects in a group, they were also annotated as such. However, the choice of doing so is done rather arbitrarily: only in images with boxes of donuts were they annotated as a group, and otherwise they were annotated one by one. This leads to detection problems as the model struggles to find common ground between single donuts and groups of them.

The second round of training, this time done within 25 epochs, is done with the intention of helping the model settle on a more certain representation of the donut class. Unfortunately, although the mAP rises to 0.64, the confusion matrix reveals that the 'donuts' class is still problematic. Therefore, for the sake of testing the model's capabilities, the donuts class is removed and only the other four are kept.

The results are much better, seen in Fig. 4.7. Trained for a total of 100 epochs, within rounds of 25, the mAP rises to 0.78 and the confusion matrix ratios between predicted/actual classes are all above 0.6 within 100 epochs.

With acceptable results on the side of the pre-trained model, another experiment is done with the untrained weights values. Here, the highest mAP reached is 0.81, and the lowest predicted/actual value of the confusion matrix is 0.7. Overall, judging by the graphs in Fig. 4.8, it seems like an increase in precision.

However, with the poor real-time performance obtained on the application itself,

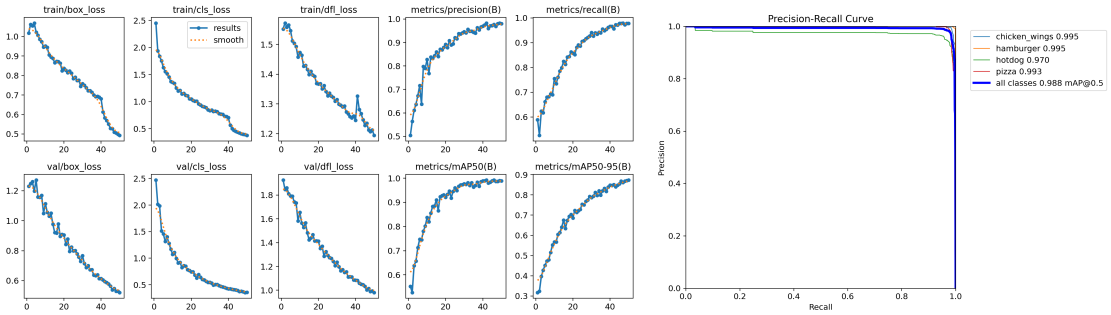


(a) Results. (b) PR Curve.

Figure 4.8: YOLOV9: Donuts label removed, untrained weights.

YOLOv9 had to be replaced and similarly trained on both pre-weighted and unweighted variations of its nano model.

The first proper training of the YOLOv8n model starts with 50 epochs, hyperparameters left to their default value, and with the pre-weighted variant serving as the basis. The results can be seen in figure 4.9.



(a) Results. (b) PR Curve.

Figure 4.9: YOLOV8: Pre-trained weights.

The results are seemingly much better. The validation losses compared to the training losses are close enough that there appears to be no significant overfitting. Likewise, the metrics of precision and recall are high, suggesting a low false positive rate and a similarly low false negative rate. Finally, the mean average precision at 50% IoU is very high at 0.98, while the mean average precision at thresholds ranging from 50% to 95% is also high (0.87).

With such results, we decide to give the model a test run on random images to verify its behaviour. It performs rather well, though noticeably worse when it came to being confidently wrong in comparison to the YOLOv9 models. Thus, we move onto the unweighted variant and begin training, foregoing any further training on the pre-weighted one to avoid overfitting.

Considering the current experiment utilizing no pre-trained weights, we raise the number of epochs to 100, but stop the process at around 75, not noticing a great

enough increase in metric values to continue the training. As such, there is no figure to display the results as with the other experiments.

It is, however, mentionable that a weaker performance was achieved, with a maximum mAP50 hovering at around 0.90% and a maximum mAP50-95 of 0.70%. Likewise, during live testing, the detector would wrongly spot food objects where there were none more frequently than the pre-trained variant.

4.3.4 Results

In this subsection, the results are all compared in tabular fashion.

Metric	box_loss	cls_loss	dfl_loss	precision	recall	mAP_0.5
V9: Pre-trained	1.3534	2.1491	1.9841	0.54337	0.57714	0.54863
V9: Pre-trained, donuts removed	1.2969	1.9858	1.956	0.56268	0.59215	0.60308
V9: Untrained, donuts removed	1.3042	1.8994	1.9998	0.61687	0.58149	0.64019
V8n: Pre-trained	0.49279	0.37657	1.1941	0.98043	0.97845	0.98876
V8n: Untrained	0.95284	1.4077	1.5293	0.82217	0.7367	0.83077

Table 4.1: Results obtained at 50th epoch.

As seen in table 4.1, for the YOLOv9 model there is an overall 10% increase in mean average precision when comparing the pre-trained initial model and dataset to the better-tuned and untrained one, which both removed the problematic donuts class and changed some of the hyperparameters regarding data augmentation. The losses are still relatively high, but those are fixed as the number of epochs is increased.

On the other hand, the YOLOv8n model has much better performance and a lower loss score overall, but is more overfit, which is more easily seen during live testing. A problem that is deemed less important in the scope of the application, as the user can easily shift the camera around for a better scanning angle, while the increase in speed is irreplaceable.

The final results of each configuration seen in table 4.2 make it clear that the first path is not one worth pursuing before all annotated images in the donuts class are improved upon, especially visible for the mAP of the donuts class sitting at 0.199, which is a subpar value.

On the other hand, the other two YOLOv9 configurations seem to be very close together in terms of precision, each class differing by a float value of around 0.05-0.07 when comparing its pre-trained and untrained version. Surprisingly and rather noteworthy is that the 'Pizza' class has seen an improvement when using untrained

Class	Chicken Wings	Hamburger	Hotdog	Pizza	Donuts	All
V9: Pre-trained	0.759	0.808	0.561	0.896	0.199	0.645
V9: Pre-trained, donuts removed	0.816	0.825	0.651	0.858	NIL	0.788
V9: Untrained, donuts removed	0.753	0.872	0.708	0.921	NIL	0.813
V8n: Pre-trained	0.995	0.995	0.970	0.993	NIL	0.988
V8n: Untrained	0.853	0.872	0.708	0.849	NIL	0.830

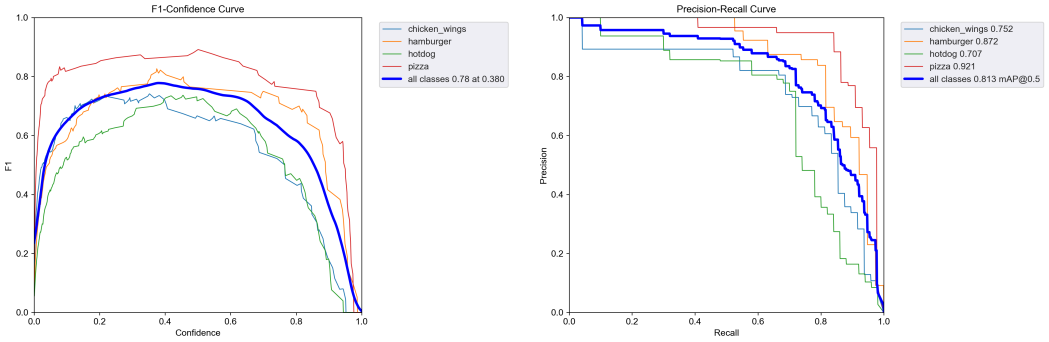
Table 4.2: Final precision value of each class.

weights over the pre-trained ones, despite the COCO dataset which the pre-training was completed upon having a 'Pizza' class of its own.

Likewise notable is that the only class that has seen an improvement on the YOLOv9 pre-trained model was the 'Chicken Wings' class, which had a slightly better mAP (by approximately 0.003) even during the very first experiment.

For the final part of the training and testing, the previously untrained version of the YOLOv9 model (the last V9 experiment) was trained on the validation set in 16 batches at IoU 0.7 and confidence 0.001. The results are strikingly similar to the final mAP, seen in Fig. 4.10. The PR curve (b) shows a total mAP of 0.813 and a slight difference of at most 0.001 between individual class precision values in validation versus training.

The results of the YOLOv8n model were much better numerically than those of the YOLOv9 model. Not only does YOLOv8n have less parameters and therefore requires less computational power, but it also has a higher accuracy, if potentially more overfit. Still, speed is preferred in the case of the mobile application, as YOLOv9 is too big of a model to properly run on a mid-range smartphone.



(a) F1 Curve. (b) PR Curve.

Figure 4.10: Final V9 validation results (no donuts label, previously untrained model weights.

4.3.5 Discussion

In this section we will present the insights gained from our use of object detection, the limitations encountered, possible improvements and also compare our work to existing solutions.

The first and main point of training a real-time object detection model to recognize food items has been achieved successfully, and more-so lead to gaining a new understanding of how to best go about this endeavour in the future.

The second point, that of running the real-time object detector mobile device, has seen numerous issues and unsolvable challenges, mostly due to the hardware available, but also due to a lack of documentation for certain new technologies and libraries, mostly on the Unity side of the application.

Finally, the third point, combining AR with AI, has been achieved, which leads to an advantage over other applications of similar nature due to the seamlessness that only AR can offer.

Insights

The results displayed in the previous section have made it clear that the dataset's quality is very important, perhaps more so than usual, in how the model will behave. Every food object must be approached critically, and all images in relation to it must be annotated similarly. Refraining from doing so would lead to the poor results of the 'Donuts' class, where some images had only one or two donuts which were then annotated, while others had whole boxes.

Therefore, in order to successfully create a proper food dataset, we must first reason whether to take food in singular entities or as groups. An object detector trained to recognize groups of donuts will not recognize a singular donut, and vice-versa. Similarly with hot wings, or other such foods.

Another insight gained in terms of dataset creation is a significant amount of foods share traits that make it difficult for object detectors to confidently classify each item. An example frequently encountered is that of the 'Hotdog' and the 'Hamburger' classes, where certain images depicting hotdogs would bear striking resemblance to hamburgers due to a lack of a fully visible sausage, therefore falling more into the latter class than the first.

The final observation on this topic is that the augmentation of images must be carefully done. Unlike general object detection, food is very much based on colour, perhaps even more so than shape. Hue and saturation changes, if necessary, are to be very lightly applied.

Limitations

It is apparent that the current mobile phone technology, or at the very least the technology readily available to the average consumer, is not yet ready for the state-of-the-art object detectors presented.

YOLOv9, although showing satisfying results, has yet to receive a 'nano' version of its own, which leads to an overabundance of parameters and computations that cannot be run on the average smartphone reliably. Its innovative architecture is also much too complex, raising the time it would take to train the model and run inference.

YOLOv8n, its immediate predecessor, falls short in the same way as YOLOv9, though with more practical results. While it still raises the issue of a lack of memory when attempted on a mid-range smartphone, it runs for an average of 5-7 FPS on a high-end phone. Unfortunately, this is unusable when combined with the AR side, which requires more fluidity in terms of performance for the users to gain the full experience.

Thus, we turn to a much more tried and tested model that we know will work well, the YOLOv2-tiny. For this, we take a pre-trained YOLOv2-tiny model, and quickly find that it is much smoother in its operation with Unity and Barracuda, even if its inference results leave much to be desired. During detection, discounting the occasional freeze frame, it runs with an approximate 20-30 frames per second.

Unfortunately, we could not determine if the poor performance of the YOLOv9 and YOLOv8n models was primarily because of the Sentis library's on-going problems with Non-Maxima Suppression or not. YOLOv2-tiny is both a more lightweight and simpler model than the above, so it stands to reason that it would be faster in terms of performance. Yet it also uses the Barracuda library, which is not known for having issues with NMS, and is specifically made compatible with YOLOv2-tiny.

Possible Improvements

The first improvement possible is also related to the limitations presented above, namely changing the object detection model. Though the most recent of detectors may not work, there are many before them which could strike the perfect balance between speed and detection performance, such as YOLOv4-tiny, or YOLOv5n.

Secondly, to improve the application itself, a newer version of Unity could be used, though that would require a newer version of Barracuda or even Sentis, which would in turn require one of the newer models mentioned above.

The third improvement could be brought by adding more quality images and classes to the dataset. The more classes that are added, the more images that are

needed for the detector to successfully differentiate between sub-types of food, such as a hypothetical classification of soup, of which there are many types that distinguish themselves via minor details.

Related Work

As summarized in Chapter 2, each of the existing solutions utilize a light-weight real-time object detection model and also pre-existing datasets.

Our solution is most similar to the first one presented which similarly utilized YOLOv2 in their implementation, though ours uses a custom dataset that currently contains fast food, which is more relatable to the average consumer. Our solution also has better mAP if we are to take into account the YOLOv9 and YOLOv8n results, and a higher frames/second count when comparing the applications themselves.

Likewise, the second solution also falls short in terms of would-be accuracy and FPS, as does the third, whose dataset is less geared towards food and more towards cutilvars. It bears mentioning that we also make use of AR technology to more comfortably display information for users, which is another bonus in our favour compared to all three existing solutions.

4.4 The Unity Application

4.4.1 ONNX, Barracuda and Sentis

In order for it to be possible to work with any ML / AI models in Unity in the first place, a neural network inference library such as Barracuda, or as it known at the time of writing this paper, Sentis, is needed. It is what allows the implementation of our custom model into Unity, along with a suite of other benefits such as optimization for mobile devices, consoles and desktops.

Open Neural Network Exchange, or ONNX, is an open-source format that represents machine learning models. It enables transferring between different frameworks like PyTorch, Caffe2 and Tensorflow. It supports a wide range of operators, such as tensor operations, neural network layers, recurrent layers, transformations, loss functions and so on.

For our model to be usable in Unity, it must first be converted into ONNX format, and then accepted by the Barracuda/Sentis library, both of which come with their own problems.

To begin with, exporting a machine learning model to ONNX can be done with a simple use of the torch.onnx.export, called on the aforementioned model. This then

executes the model with a given input tensor and records a trace of what operators are used to compute the given outputs.

At the time of this experiment, however, YOLOv9 did not officially support the export functions, as a re-parametrization was needed before exportation. The functions to do said re-parametrization were only present and implemented in the YOLOv7 repository, and needed to be moved over. Fortunately, the appropriate support was soon added and development could continue.

The next step is importing the ONNX model into Unity. The largest and most frequently encountered issue when utilizing Barracuda is the lack of support for more modern models. Dynamic shape operations, control flow operations or even operators used in handling sparse tensors are unsupported by the latest version of Barracuda at the time of this paper. This problem is encountered when trying to import YOLOv9 into Unity, as a large amount of operations within its layers were deemed to be unsupported by Barracuda. Therefore, we switched to the newest, if experimental, Barracuda package, which was renamed to Sentis.

While the Sentis package comes with its own limitations, it does allow for the implementation of any new object detection model such as YOLOv9 and YOLOv8.

With the custom-trained YOLOv9 model now exported to ONNX and imported into Unity for development came the final step of writing the necessary script for testing the model itself. By adapting an existing YOLOv7 script implemented in Unity with Sentis, we successfully confirmed the desktop performance and accuracy of our real-time detection model when used within Unity.

The script itself compiles the model with additional functional layers for post-processing, such as Non-Maximum Suppression (NMS), via Sentis functions. The input source is then set up as the webcam texture, which takes real-time video frames and then performs inference on each one, processes the output tensors to extract the bounding box coordinates and labels, and then overlays the extracted items over the display. A DrawBox method is responsible for creating the graphical representations of the bounding boxes and labels. In terms of optimization, a pool of bounding boxes is maintained for the sake of reusability.

Unfortunately, when testing the model live on a mid-range smartphone, the A50s 6GB, it became clear that the YOLOv9 model is too computationally intensive for the device, as the application could not even start before throwing up errors mentioning the lack of memory available. An A54 was then used, which could run the detection with very few frames.

The YOLOv8n is then similarly exported to ONNX format and plugged into Unity, but the same issue persists. While the A50s can now run the application, it is much too laggy to be usable. Further optimizations also seem not to make an impact on the application. Adding the AR component on top of this leads, once

again, to a lack of memory.

After much investigation, it appears that Sentis is rather problematic when it comes to phone optimization and has an issue with models that have Non Maxima Suppression (NMS), leading to poor performance. Therefore we shift to Barracuda 3, the predecessor of Sentis, and also revert the version of the Unity engine to a 2022 variant.

The application is now usable yet again on the A50s, but with poor performance and heavy frame drops. Implementing a button that begins the detection and stops after 5 seconds improves usability, but detection still stops after the memory becomes a problem soon again. It appears that combining a new AI model with AR and then running the application in real-time is not yet possible for a mid-range smartphone.

In order to prove the utility and potential of integration between AR and AI, however, we shift to an older, more light-weight real-time object detection model: the YOLOv2-tiny.

4.4.2 Food API

The food API used is a free one provided by CalorieNinjas. It allows users to input a string and returns nutritional information based on the detected food. For the sake of speed, we first call the API at the start-up phase of the application with every class name, and then store the information so that it is both up-to-date and quickly available to be displayed.

After detection via the model, the class name is returned as part of the output. Based on the class name acquired, we then change the text of the AR prefab that will be spawned to contain the information relevant to the detected food item / class name.

The relevant script is NutritionFetcher, which is designed for the purpose of fetching and storing information for our list of labels via the external API. It initializes the process of fetching as soon as it starts via the Unity Start() method, which calls a coroutine as soon as possible and iterates through each label, sends the requests to the API, and parses the JSON responses to extract the relevant information.

As mentioned, the script also contains two functions that allow it to save and load the information by using a text file, should the API not be available.

Two custom data structures are used to optimize the code and make it more readable, NutritionItem and NutritionResponse. The first represents a single item of nutritional information, with properties for its nutritional values. The second represents the JSON response structure from the external API and combines multiple

NutritionItems in a list.

Finally, the script contains a public method GetNutritionalInfo() which allows other scripts access to the information of a given label.

4.4.3 Augmented Reality

Unity AR Foundation is the framework used to implement the Augmented Reality portion of our application, developed by Unity Technologies for the sake of simplifying the process of developing AR applications across multiple platforms. The unified API lets developers build AR experiences that can run on iOS, Android Devices, and other such platforms supported, by acting as a bridge between the ARKit (iOS) and ARCore (Android) development platforms.

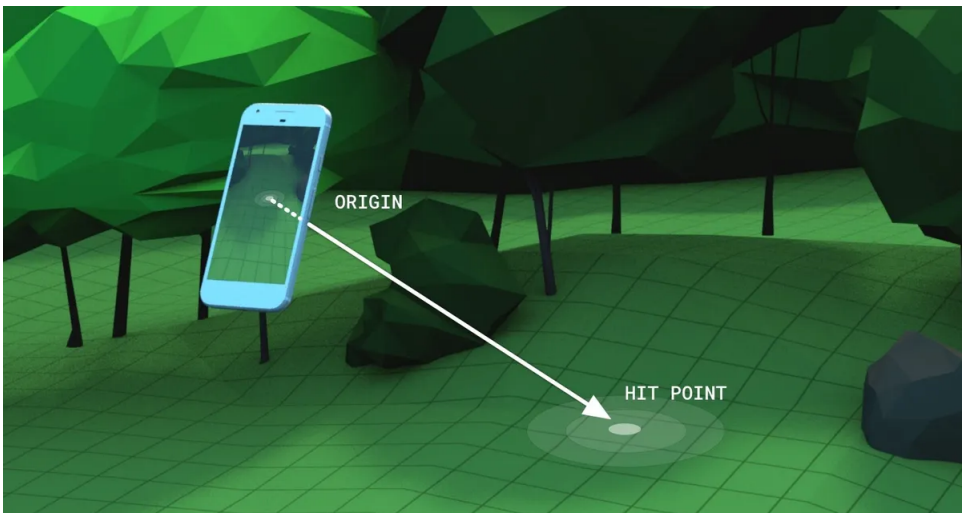


Figure 4.11: A raycasting example for AR. Source: [1]

The framework supports AR features such as detection and tracking for real-world objects, plane detection (horizontal and vertical), point cloud generation, image recognition and object tracking. It also has built-in support for user interaction in AR applications, which allows users to interact with the displayed virtual objects via gestures, touch, or other input methods.

Possibly the biggest advantage of the framework, however, is its seamless integration with Unity's existing development environment, which provide the user with access to many tools, assets and plugins. It is the reason why we chose to use Unity in the first place in order to develop our application, as no other engine or environment afforded a convenient way to utilize and manipulate an object detection model while also providing support for the development of Augmented Reality, much less cross-platform capabilities.

The AR portion begins with the initialization of an AR Origin game object, which represents the center of worldspace in an AR scene. It is the reference point from

which all the tracking data, such as position and rotation, comes from, which is set up at the start of the AR experience. It also includes components to facilitate camera movement within the virtual world based on real world movements, such as turning one's head or moving one's device around. This is what allows virtual objects to remain in place even after the user's camera has moved away from them entirely.

The next major AR component we make use of is ARRaycastManager, which serves as a bridge between Unity's physics system and the ARFoundation framework by essentially detecting real-world surfaces. It allows developers to cast rays, which are then used to detect surfaces such as tabletops, walls or floors. We utilize this concept to determine where the virtual text objects should be placed so that they are as close as possible to the detected food item. Raycasting can be seen in 4.11.

ARRaycastManager sorts hits by range in ascending order, therefore placing the closest hit at index 0. As a consequence, we start raycasting only when bounding boxes are considered to be stable (if they are static for a certain amount of frames), and place the new AR anchor based on the first hit in the array. We also keep a dictionary which has the anchors as keys and the bounding boxes as values for the sake of later removal.

Finally, we have a prefab, short for "prefabricated object", which is a reusable asset that allows us to store and configure a GameObject complete with components and properties. We utilize this so we can create multiple instances of a specialized object containing a TextMesh whose value changes based on the label of the detected object and a script that ensures the text always rotates based on the user's position, so that it is always readable.

4.4.4 User Interface

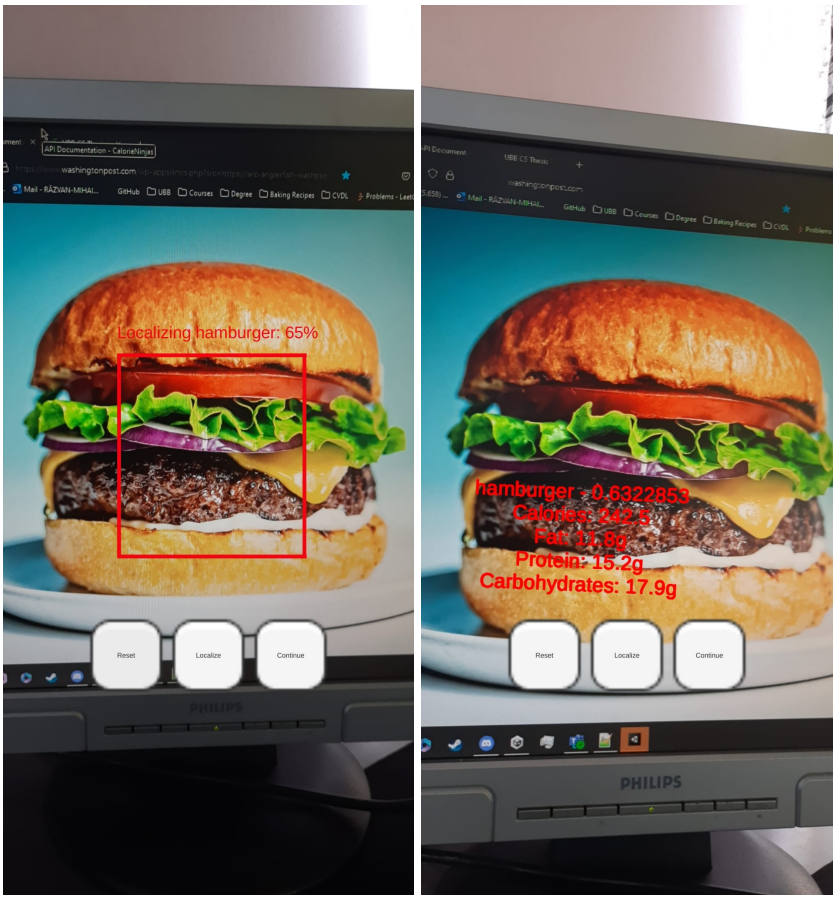
The user interface is kept minimal intentionally, as the purpose of our application is to be a proof of concept that augmented reality and artificial intelligence can be bridged together for inspiring results.

As the Unity application loads in, the user is presented with the back camera view of their phone and three main buttons, seen in 4.12.

The left button, entitled "Reset", allows the user to completely reset the scene and stop any on-going detections. This includes deleting existing anchors, but does not unload the information retrieved from the API.

The middle button, "Localize", shortens the time required by the detector to confidently place the AR prefab. It is meant to be used in case the user already knows the detected food and agrees that the detector has achieved the right result.

The right-most button, "Continue", is meant to be a soft reset. It stops any on-



(a) The detection process & bounding box.

(b) The text prefab placed within augmented reality.

Figure 4.12: The application in use.

going detections, but does not delete the existing anchors, ergo it allows the user to continue detecting and placing more AR prefabs so that they may then go through each one at their leisure.

Should the user not press any of the buttons, the detector will keep attempting to find a recognizable food item. Once one is found, it will continue to scan it for approximately 30 frames, and if the confidence does not change and neither does the bounding box during those allocated frames, then it will place the text prefab by itself.

After localization is done and the augmented reality part is successfully completed, the object detector no longer looks for food, and instead remains dormant until the user resets, continues the scan, or restarts the application.

4.4.5 Design

As our application is made in Unity, simply showing the class diagram is not enough to accurately represent its intricacies, nor to truly show how each module interacts

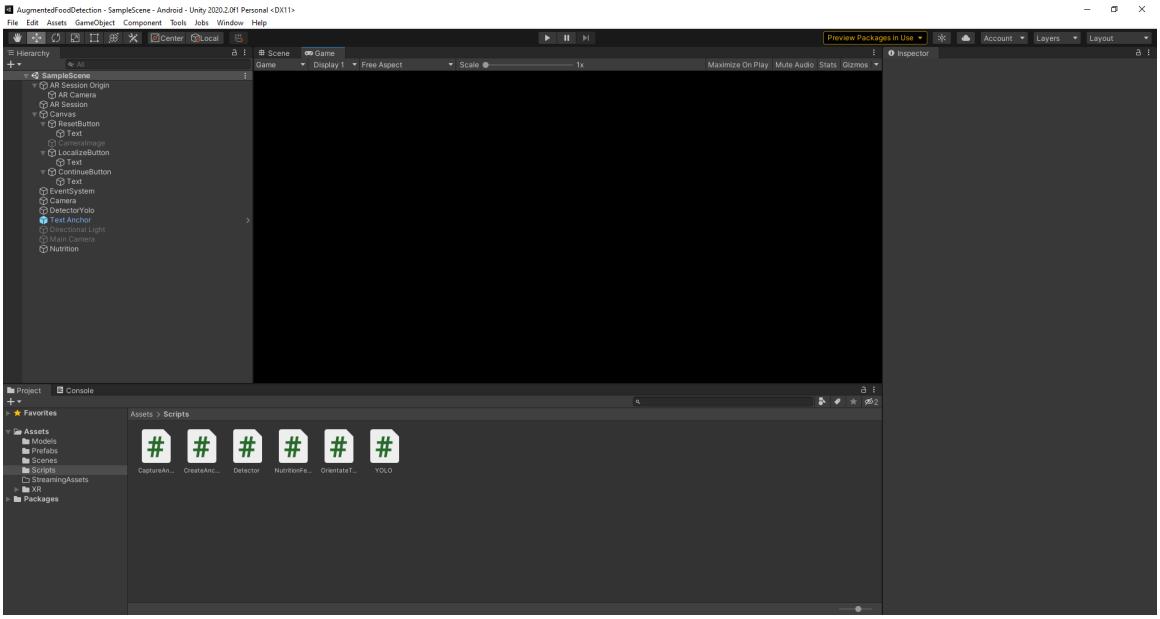


Figure 4.13: The game scene within Unity, including hierarchical view.

with one another. Seen in figure 4.13 is a screenshot of the Unity editor displaying the ”Game” view and the ”Hierarchy”. The Hierarchy panel shows the structure of the scene with GameObjects and their components.

AR Session Origin, as explained in the section 4.4.3, serves as the central reference point in an AR scene. This object is crucial because it anchors the tracking data, including position and rotation, which are established at the start of the AR experience. This GameObject has the CreateAnchor script, which is responsible for creating and managing anchors at specific points in the AR environment. It also keeps track of created anchors and works in conjunction with CaptureAndDetect to place the anchors at the detected objects' locations.

Placing the CreateAnchor script on the AR Session Origin object ensures that the anchors are correctly positioned relative to the user's environment. It also allows easy interactions with other AR components.

AR Session manages the lifecycle and configurations of the AR experience, along with tracking its state and some error handling.

DetectorYolo is an EmptyObject-type GameObject that holds the script YOLO. It initializes the model and its labels file, preprocesses images, normalizes model outputs, extracts bounding box dimensions, calculates IoU and applies NMS to reduce the number of overlapping bounding boxes. Placing it on an empty object allows for easier initialization at the start of the application.

Nutrition is also an EmptyObject-type GameObject that is used to house the NutritionFeedback script. It fetches nutritional information for a list of food labels from an external API (CalorieNinjas). As with the YOLO script, it is placed on an empty object so that it is initialized at the start of the application.

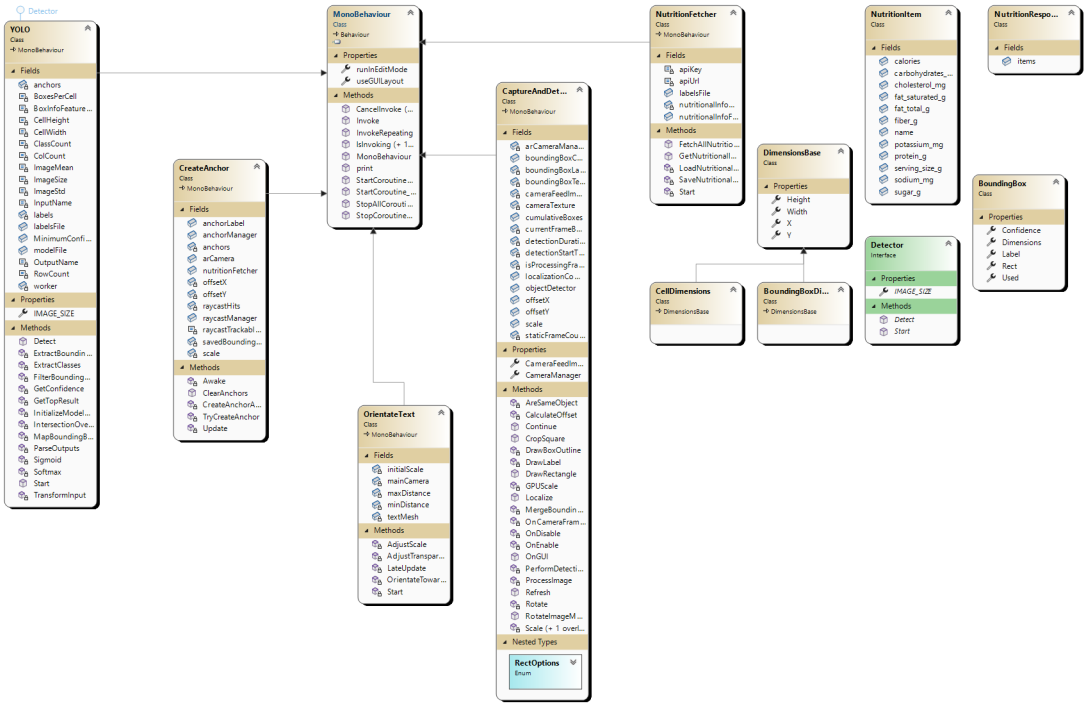


Figure 4.14: The application's class diagram.

The Canvas object contains the buttons and the background that is used as the RawImage by the CaptureAndDetect function.

Camera contains the CaptureAndDetect script, which manages the AR camera, captures frames, processes the feed for detection, determines when localization is complete, and draws bounding boxes. The script is placed on an empty object for the sake of modularity, single responsibility, controlled initiation and easier configuration.

The text prefab is also present within the hierarchy due to its nature as a prefab. We attach a script to it, OrientateText, which makes it so that the text automatically orientates itself to face the user's camera at all times, and also grow / shrink based on the distance, up to a maximum, respectively minimum, value.

Seen in figure 4.14 is the class diagram of the application. There can be seen what connections exist between the scripts themselves, most inheriting from MonoBehaviour. MonoBehaviour is a base class in Unity which contains the framework that allows scripts to be placed on GameObjects in the first place.

We make use of multiple helper classes, amongst which is the BoundingBox class that represents the bounding box outputs, and DimensionsBase, which serves as a base class for defining dimensions of objects within the context of a detection system. It encapsulates common properties that are shared among different types of dimensions, such as position (X, Y) and size (Width, Height). By defining these properties in a base class, we try to promote some code reuse and provide a useful interface for working with dimensions.

Likewise, CellDimensions represents dimensions specific to a cell in the context of an object detection system. For YOLO, the image is divided into these cells, with each being responsible for bounding boxes within its purview.

4.4.6 Testing

Testing is within a total of 49 main iterations, with more unaccounted ones in-between that had unremarkable changes. For the sake of debugging, we make use of both Android Logcat, which presents real-time information about the android application running on a connected device, and Debug.Log(), which is part of Unity's scripting API.

The subject of testing varies across iterations. The first 5 main tests consist of attempting to make the YOLOv9 model work with Sentis and also perform well on the mobile application. We test for both CPU and GPU-based detection. We also test on both a Samsung A50s smartphone and an A54, with both having unsatisfying results.

The following 10 tests are all focused on YOLOv8n, where we attempt to boost its performance via multiple means, including but not limited to: changing from GPU-based detection to CPU, using IL2CPP scripting backend over Mono, using ARM64 architecture over ARM7, turning on multi-threading, and detecting over fewer frames, rather than every frame. Tests are run on both a Samsung A50s smartphone and an A53, though both fail to provide convincing results.

For the next 8 main tests, we attempt to use YOLOv3-tiny with our own dataset. Unfortunately, the issue this time stems from the older libraries being used, which lead to issues with not only having Unity and Barracuda recognize the model itself, to failing to properly set the input and output of the model within Unity.

The greater half of tests that follow are all focused on utilizing a pre-existing YOLOv2-tiny model, pre-trained on the Food100 dataset, and integrating it with Unity's AR framework. Yet again we attempt multiple different settings and optimizations such as multi-threading and even merging bounding boxes, though this time the effects are visible and acceptable. We put an end to the testing phase once all main functionalities are deemed to be working.

Chapter 5

Conclusions and Future Work

In conclusion, our work provides a solution to the lack of knowledge regarding nutrition and food-related allergies in a convenient and quick way. Through utilizing the power of a state-of-the-art real-time object detection model such as the YOLO architecture that ensures minimal information loss and an acceptable balance between speed and accuracy, and connecting the power of AI with that of AR to change the way information is displayed and therefore processed by consumers.

Experiments were run on the YOLOv9 and YOLOv8n model, starting with a small handmade dataset, augmented and split into train-test-validation images, and then tried on both the unweighted and weighted version of YOLOv9 and YOLOv8n with various results. A difficulty was encountered in deciding how the 'Donuts' class should be annotated in images, due to the fact that annotating them one-by-one was not only overly time-consuming, but also would result in a flood of information once displayed in AR form. Alternatively, group annotations would end up in less reliable results. Ultimately, the class was scrapped, and training proceeded with the remaining four. After careful comparison of the two, it was decided that the unweighted version was better in terms of precision compared to the pre-weighted one, with the former sporting a mAP of 0.78, and the latter a mAP of 0.81.

Further performance-related difficulties prevented us from reaching a favourable outcome to our experiment, and so we turned away from the current state-of-the-art models, and instead looked to the tried-and-tested older models. From there, we utilized a pre-trained YOLOv2-tiny model and succeeded in our task at the cost of noticeable precision loss.

Potential applications of the solution were reviewed, highlighting ideas such as assistive dining for use in restaurants or food courts, food tourism and cultural exploration, and smart kitchen applications which could go beyond the detection of types of foods, and instead focus on quality and spoilage of the foods detected. All of these would benefit from the same concept of leveraging a real-time object detector such as YOLO and combining it with the visual power of augmented reality.

Future work was also considered, mainly focused on the performance of the application and the utilization of state-of-the-art models, which currently cannot be run reliably by mid-range smartphones and other such devices. Likewise, depth approximation for the placement of the AR text could be improved upon so that it is more accurately placed to its actual location. This would require a depth estimation model or similar, which is also too computationally intensive to attempt alongside detection and AR simulation, at least as of the time of writing this paper.

In terms of strengths and opportunities, the health benefits were enunciated, while the emergent nature of AR and the recent technological breakthroughs of AI were also mentioned. For weakness and threats, it was made clear that wrongly identifying objects, especially those in the culinary sphere, could have unfortunate repercussions and lead to more loss of trust than with normal object detection tasks. Likewise, standing in contrast to the mentioned benefit of combining new technology, a lack of proper support and documentation for said technology can also be seen as a weakness.

In the end, the results could be improved upon greatly, starting with a larger dataset of more classes and also a change of model. This, however, will be left as future work, as this would require resources we do not currently sport, such as multiple GPUs capable of intense training along with more RAM memory and disk space.

Bibliography

[1] A. Ameye. A cursor for augmented reality. https://ameye.dev/notes/ar-cursor/, September 2023. accessed Online 5th of June.

[2] S. Araya, A. Elberg, C. Noton, and D. Schwartz. Identifying food labeling effects on consumer behavior. *Marketing Science*, 41(5):982–1003, 2022.

[3] F. Arena, M. Collotta, G. Pau, and F. Termine. An overview of augmented reality. *Computers*, 11(2):28, 2022.

[4] A. Benassi, A. Carboni, S. Colantonio, S. Coscetti, D. Germanese, B. Jalil, R. Leone, J. Magnavacca, M. Magrini, M. Martinelli, et al. Augmented reality and intelligent systems in industry 4.0. 2020.

[5] M. Boland. Arinsider mobile ar users break the billion barrier. Available from https://arinsider.co/2023/07/17/mobile-ar-users-break-the-billion-barrier/, 17 July 2023. accessed Online 24th of April 2024.

[6] L. Bossard, M. Guillaumin, and L. Van Gool. Food-101–mining discriminative components with random forests. In *Computer Vision–ECCV 2014: 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part VI 13*, pages 446–461. Springer, 2014.

[7] Y. Cai, Y. Zhou, Q. Han, J. Sun, X. Kong, J. Li, and X. Zhang. Reversible column networks. *arXiv preprint arXiv:2212.11696*, 2022.

[8] DeepLearning.ai. C4W1L02 Edge Detection Examples. `https://www.deeplearning.ai/`. Online; accessed 14 April 2024.

[9] N. J. H. T. e. a. Dwyer, B. Roboflow (version 1.0). Available from https://roboflow.com. computer vision., 2014. accessed Online 19th of April 2024.

[10] F. Jacob Solawetz. What is yolov8? the ultimate guide. [2024]. Roboflow Blog: https://blog.roboflow.com/whats-new-in-yolov8/, Jan 11, 2023. accessed Online 25th of May.

[11] G. Jocher. Ultralytics yolov8. Available from https://github.com/ultralytics/ultralytics, January 10th 2023. accessed Online 25th of May.

[12] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *IJCV*, 2020.

[13] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[14] C.-Y. Lee, S. Xie, P. Gallagher, Z. Zhang, and Z. Tu. Deeply-supervised nets. In *Artificial intelligence and statistics*, pages 562–570. Pmlr, 2015.

[15] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia. Path aggregation network for instance segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 8759–8768, 2018.

[16] Y. Matsuda, H. Hoashi, and K. Yanai. Recognition of multiple-food images by detecting candidate regions. In *2012 IEEE international conference on multimedia and expo*, pages 25–30. IEEE, 2012.

[17] B. Morag, P. Kozubek, and K. Gomułka. Obesity and selected allergic and immunological diseases—etiopathogenesis, course and management. *Nutrients*, 15(17):3813, 2023.

[18] A. Ramesh, A. Sivakumar, and S. S. Angel. Real-time food-object detection and localization for indian cuisines using deep neural networks. In *2020 IEEE International Conference on Machine Learning and Applied Network Technologies (ICMLANT)*, pages 1–6. IEEE, 2020.

[19] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[20] J. Redmon and A. Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[21] J. Redmon and A. Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[22] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. *Advances in neural information processing systems*, 28, 2015.

[23] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. *International journal of computer vision*, 115:211–252, 2015.

[24] V. Sampath, E. M. Abrams, B. Adlou, C. Akdis, M. Akdis, H. A. Brough, S. Chan, P. Chatchatee, R. S. Chinthrajah, R. R. Cocco, et al. Food allergy across the globe. *Journal of Allergy and Clinical Immunology*, 148(6):1347–1364, 2021.

[25] J. Sun, K. Radecka, and Z. Zilic. Foodtracker: A real-time food detection mobile application by deep convolutional neural networks. *arXiv preprint arXiv:1909.05994*, 2019.

[26] M. Tan and Q. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.

[27] M. Tan, R. Pang, and Q. V. Le. Efficientdet: Scalable and efficient object detection. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10781–10790, 2020.

[28] P. Taylor. Number of smartphone mobile network subscriptions worldwide from 2016 to 2022, with forecasts from 2023 to 2028. Available online at: https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, accessed Online 19th of April 2024.

[29] G. Waltner, M. Schwarz, S. Ladstätter, A. Weber, P. Luley, M. Lindschinger, I. Schmid, W. Scheitz, H. Bischof, and L. Paletta. Personalized dietary self-management using mobile vision-based assistance. In *New Trends in Image Analysis and Processing–ICIAP 2017: ICIAP International Workshops, WBICV, SSPandBE, 3AS, RGBD, NIVAR, IWBAAS, and MADiMa 2017, Catania, Italy, September 11-15, 2017, Revised Selected Papers 19*, pages 385–393. Springer, 2017.

[30] C.-Y. Wang, I.-H. Yeh, and H.-Y. M. Liao. Yolov9: Learning what you want to learn using programmable gradient information. *arXiv preprint arXiv:2402.13616*, 2024.

[31] L. Wang, C.-Y. Lee, Z. Tu, and S. Lazebnik. Training deeper convolutional networks with deep supervision. *arXiv preprint arXiv:1505.02496*, 2015.