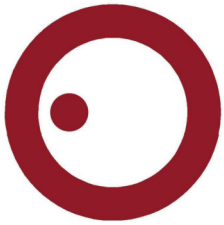
Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharia Informática
Mestrado em Eng.ª Informática – Computação Móvel

IMPLEMENTATION OF AI-POWERED COMPUTER VISION SYSTEMS FOR EDGE DEVICES USING YOLO AND FACE API

JOÃO REBELO DOS SANTOS



escola superior de tecnologia e gestão
instituto politécnico de leiria

Instituto Politécnico de Leiria
Escola Superior de Tecnologia e Gestão
Departamento de Engenharia Informática
Mestrado em Eng.ª Informática – Computação Móvel

IMPLEMENTATION OF AI-POWERED COMPUTER VISION SYSTEMS FOR EDGE DEVICES USING YOLO AND FACE API

JOÃO REBELO DOS SANTOS
Number: 2232646

Internship report under the supervision of Professor Doctor Marisa da Silva Maximiano.

A C K N O W L E D G M E N T S

I would like to thank Professor Doctor Marisa da Silva Maximiano, my internship report supervisor, for her availability and guidance throughout the entire process

RESUMO

Este relatório de estágio apresenta o desenho, a implementação e a avaliação de dois sistemas leves de inteligência artificial orientados para dispositivos de borda (edge), desenvolvidos durante um estágio académico na Bitcliq Technologies. Motivada pelo aumento dos custos computacionais, energéticos e do consumo de água associados a modelos de grande escala, e pela necessidade prática de soluções executáveis em hardware com recursos limitados, a investigação explora abordagens que privilegiam desempenho em tempo real, robustez e capacidade de operar em modo offline. O primeiro projeto implementa um pipeline de deteção de objectos e classificação de espécies de amêijoa utilizando o modelo leve YOLO nano, treinado num conjunto de dados altamente desequilibrado; os resultados indicam que o modelo constitui uma solução viável para implantação em dispositivos de borda, atingindo baixas latências (0,1 s de inferência) e boa capacidade de generalização quando são aplicadas estratégias básicas de aumento de dados. O segundo projeto consiste numa aplicação de reconhecimento facial a correr no navegador, com modelos executados no cliente via face-api.js, concebida para automatizar o registo de entrada/saída de trabalhadores mesmo em condições de rede intermitente; a solução funciona bem em dispositivos modernos, mas o desempenho degrada-se em hardware mais antigo. Ambos os sistemas chegaram a estágios de prova de conceito: o pipeline para amêijoas produziu um modelo funcional (suspenso antes da prototipagem final) e a implementação com face-api.js permanece um protótipo funcional em fase inicial. Em resumo, os projetos desenvolvidos e apresentados no documento demonstram que a selecção cuidada de modelos leves e de estratégias de implantação pode reduzir significativamente os requisitos de recursos, mantendo níveis de desempenho compatíveis com muitas necessidades práticas de IA em edge.

ABSTRACT

This internship report delivers on the design, implementation and evaluation of two lightweight, edge-oriented AI systems developed during an academic internship at Bitcliq Technologies. Motivated by the growing compute, energy and water costs of large-scale AI and by the practical need for deployable solutions on constrained hardware, the work explores approaches that prioritize real-time performance, robustness and offline operation. The first project implements a real-time clam detection and species-classification pipeline using a lightweight YOLO nano model trained on a highly imbalanced dataset; results show the model is a viable option for edge deployment, achieving low latencies (0.1 s inference) and strong generalization given basic data-augmentation strategies. The second project produces a browser-based face recognition application that runs models client-side with face-api.js to support employee check-in/out under intermittent network conditions; the implementation performs well on modern devices but degrades on older hardware. Both systems reached proof-of-concept stages: the clam pipeline yielded a functional model (stopped prior to final prototyping) and the face API implementation remains an early but working prototype. Overall, the internship report demonstrates that carefully selected lightweight models and deployment strategies can meaningfully reduce resource requirements while meeting many practical needs for edge AI.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

Adam	Adaptive Moment Estimation.
AI	Artificial Intelligence.
CCL	Connected-Component Labeling.
CNN	Convolutional Neural Network.
COCO	Common Objects in Context.
CPU	Central Processing Unit.
CSS	Cascading Style Sheets.
EHGS	Enhanced Hunger Games Search.
EXIF	Exchangeable Image File Format.
FN	False Negative.
FP	False Positive.
GPT-3	Generative Pre-trained Transformer 3.
GPU	Graphics Processing Unit.
GT	Ground Truth.
HTML	HyperText Markup Language.
HTTPS	Hypertext Transfer Protocol Secure.
ID	Identifier.
IoT	Internet of Things.
JS	JavaScript.

L1	L1 Regularization (Lasso).
L2	L2 Regularization (Ridge).
LFW	Labeled Faces in the Wild.
mAP	mean Average Precision.
mAP@0.5	mean Average Precision at Intersection-over-Union (IoU) threshold 0.5.
mAP@0.5:0.95	mean Average Precision averaged across IoU thresholds from 0.5 to 0.95 in steps of 0.05.
NMS	Non-Maximum Suppression.
NumPy	Numerical Python.
OpenCV	Open Source Computer Vision Library.
P	Precision.
PC	Personal Computer.
PIL	Python Imaging Library.
PIN	Personal Identification Number.
PPS14	Portuguese Traceability Standard.
PyTorch	Python Machine Learning Library.
QR	Quick Response (code).
R	Recall.
R&D	Research and Development.
ReLU	Rectified Linear Unit.
ResNet	Residual Neural Network.
ROI	Region of Interest.
RT-DETR	Real-Time Detection Transformer.

SSD	Single Shot MultiBox Detector.
TFJS	TensorFlow JavaScript.
TFLite	TensorFlow Lite.
TP	True Positive.
UI	User Interface.
VGG	Visual Geometry Group.
XSS	Cross-Site Scripting.
YOLO	You Only Look Once.

1 INTRODUCTION

This report documents work carried out during an academic internship at Bitcliq Technologies [5]. The document therefore reflects applied Research and Development (R&D) activities performed within the company and its operational context, and focuses on the design, implementation and evaluation of systems that were developed or prototyped as part of that internship.

1.1 HOSTING COMPANY DESCRIPTION

Bitcliq Technologies is a Portuguese Blue-Tech startup that develops R&D-driven digital solutions for the blue economy and related primary sectors, with a focus on cloud platforms, traceability and intelligent information systems. Its product portfolio includes BIG EYE Smart Fishing — a cloud platform for real-time fleet operations and seafood traceability [3] — and Lota Digital, a blockchain e-marketplace for the first sale of fish that is currently not under active development. Bitcliq follows an iterative, project-driven workflow in which long-running flagship platforms are continuously enhanced while new pilots and R&D demonstrators (often combining blockchain, Artificial Intelligence (AI) and Internet of Things (IoT)) are developed via partnerships and public programmes [4].

1.2 BACKGROUND INFORMATION

The original scope of the internship was the development of a blockchain solution (based on Hyperledger Fabric) to support the Portuguese Traceability Standard (PPS14) / Farm2Fork traceability initiative [11, 34], an effort aimed at ensuring trustworthy, end-to-end provenance for food products by recording product data from early production stages through to retail. Farm2Fork's objective — to make complete lifecycle information available to downstream actors and consumers (for example via Quick Response (code) (QR) codes and an AI assistant) — is broadly comparable in intent to widely publicized blockchain traceability pilots in industry

(e.g., Walmart's 2017 pilot) [22, 47], but differs in its explicit aim to capture data from the earliest production stages and to expose that information directly to end users.

Due to delays in the Farm2Fork timeline the internship scope was adapted and the work was instead done on other applied R&D tasks. Principal among these was an AI-powered clam classification project: an image-classification pipeline developed to automate species identification of shelled catch, with the primary objective of reducing the manual labour required for sorting clams in an efficient manner. Among other contributions, I also developed a browser-based, AI-enabled face-recognition web application for the SmartFarm platform, specifically aimed at streamlining worker clock-in/clock-out procedures.

1.3 SCOPE

The primary deliverable became an AI-based clam classification and detection pipeline: an image-classification and detection system intended to automate species identification and reduce the manual labour required for clam sorting. In addition, the internship included the development of a browser-based, AI-enabled face recognition application for the SmartFarm platform to digitise worker clock-in/clock-out procedures and support local (offline-capable) attendance logging.

Although the blockchain solution and the AI systems were conceived within the same organisational programme of work, the two kinds of systems are technically distinct: the AI projects reported here do not implement or depend on the blockchain deliverable. What unites the two internship activities is a shared, practical objective — producing applied, deployable solutions that operate on constrained hardware or in intermittent-network environments (edge devices, tablets and local compute) and that prioritise efficiency, robustness and ease of integration into existing operational workflows. The following chapters therefore concentrate on the clam classification/detection pipeline and the browser-based face recognition application, describing requirements, development methodology, datasets, architectures, and evaluation results.

1.4 MOTIVATION AND OBJECTIVES

Artificial Intelligence has advanced rapidly in recent years, with increasing model complexity often accompanied by improved performance. While much of the spotlight is placed on cutting-edge models running in cloud environments, there is also demand for AI solutions that are suitable for deployment on edge devices. These environments, such as embedded systems, IoT hardware and browser-based platforms, pose unique challenges due to their limited computational power, constrained memory and often requiring real-time performance. The trend toward ever-larger cloud-hosted models also raises sustainability concerns, further accentuating the need for viable low resource solutions. For instance, training Generative Pre-trained Transformer 3 (GPT-3) has been reported to directly evaporate on the order of 700,000 liters of freshwater, and global AI water withdrawals are projected to reach billions of cubic meters, as stated in [25].

As such, the motivation behind this document stems from the need to bridge the gap between high-performing AI models and the resource constraints typical of real-world industrial applications. Industries often require solutions that are not only accurate and reliable, but also lightweight enough to run without dedicated GPUs or continuous cloud access.

This document explores two AI-based systems designed specifically with edge deployment in mind. The first applies the YOLO ([52]) object detection algorithm to develop a real-time clam segregation tool. This system aims to deliver high classification accuracy while operating with the limited resources of industrial embedded hardware in mind. The second project involves the development of a facial recognition application using Face API, intended to run directly within a web browser, for clients with an unstable connection to the server.

Both projects share the common objective of demonstrating that AI solutions can meet industrial standards of reliability and performance while remaining lightweight enough to function effectively on edge devices. This dual focus on accuracy and efficiency is critical for enabling broader adoption of AI in constrained environments where cloud-based processing is either impractical or undesirable.

1.5 DOCUMENT STRUCTURE OVERVIEW

The remainder of the document is organized as follows:

- **Chapter 2 — Fundamentals and Key Concepts:** introduces core definitions, preprocessing concepts (including connected-component labeling), dataset preparation techniques (normalization and augmentation), model architecture concepts (transfer learning, quantization), and an overview of the principal model families explored (VGG, ResNet, YOLO, Face API), together with a brief state-of-the-art discussion that frames the experimental choices made in later chapters.
- **Chapter 3 — Methodology and System Architecture:** describes the development methodology, dataset preparation, component-level architectures, requirements evolution and evaluation strategies for both the clam pipeline and the browser-based face recognition application. Hardware and tooling used during development are also specified.
- **Chapter 4 — Implementation and Development:** documents the implementation details and development process. For the clam project this includes the exploration of CCL-based preprocessing, the transfer-learning experiments with VCC16 and ResNet, dataset re-annotation for YOLO and the eventual shift to a YOLO-based detector/classifier. For the face project this chapter covers the Face API integration, service worker caching, local storage queuing and the client User Interface (UI)/flows.
- **Chapter 5 — Testing, Results, and Evaluation:** presents the experimental setup, testing methodology, measured results and their interpretation. This includes CCL segmentation outcomes, classification and detection metrics for VCC16, ResNet and YOLO variants, runtime measurements on target devices, and operational observations for the browser-based application.
- **Chapter 6 — Conclusions and Future Work:** summarises the main contributions, reflects on limitations and constraints, and proposes directions for future improvement for both the clam classification/detection pipeline and the facial recognition application.
- **Bibliography and Appendices:** contain cited references and supporting material (for example notes, dataset summaries and the internship declaration).

2

FUNDAMENTALS AND KEY CONCEPTS

This chapter introduces the fundamental concepts necessary to contextualize the methods and approaches employed in this project. It provides a high-level overview of key technical components. Rather than offering exhaustive detail, each concept is discussed in general terms, focusing on their purpose, typical use cases, and relevance to the problem domain. Select examples are included to illustrate their practical applications and potential implications for system design and implementation.

Additionally, this chapter presents an overview of related projects, as well as a review of the current state of the art. It outlines major developments in the field that have informed or influenced the direction of this research, including foundational works and notable extensions thereof. It highlights prevailing trends, ongoing challenges, and recent innovations, providing a broader context for the contributions and choices made in this project.

2.1 CORE DEFINITIONS

This section outlines the fundamental definitions that provide the foundation for the methods and analyses presented in this internship report.

2.1.1 *Service Workers*

Service workers are scripts that run in the background of web browsers, enabling features like offline access and background synchronization by intercepting and caching network requests. Their programmability allows developers to enhance user experience by serving cached content when the network is unavailable. However, the ability to cache and control network requests introduces potential security risks, particularly when third-party service workers are involved. These risks include Cross-Site Scripting (XSS) ([6], and [7]) attacks, where malicious scripts can be injected into service workers, and cache poisoning, where attackers manipulate cached data to serve malicious content. Such vulnerabilities can lead to privacy

breaches and unauthorized access to sensitive information. To mitigate these risks, it is essential to implement secure coding practices, validate inputs, and ensure that service workers are registered and served over HTTPS, as recommended by the studies [23], [6], and [7].

2.1.2 *Connected-Component Labeling*

CCL ([15]) is a technique in image analysis which allows for object detection and segmentation by identifying contiguous regions in binary images, where each region, distinct from the background, is a different object. The process typically begins by converting the image to black and white, where foreground pixels represent potential object areas, then depending on the task at hand, neighboring objects may be classified as being parts of the same greater object, or some objects may be considered irrelevant, often referred to as "icebergs", being discarded from the output, typically based on their size. The image 1 illustrates what the input and output of this process would look like.

As discussed in the study [15], CCL algorithms have seen major advances in efficiency, making them highly suitable for real-time segmentation tasks. It notes that while traditional two-pass algorithms are still widely used for their simplicity, more advanced methods like run-based and union-find approaches can provide significant speed and memory improvements depending on image structure and noise levels.

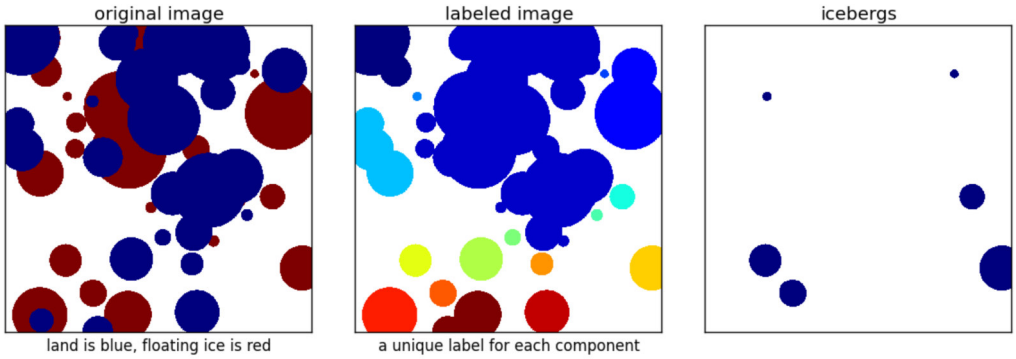


Figure 1: Connected-Component Labeling applied to a binary image ([12])

2.1.3 *Dataset*

The Dataset comprises the collection of usable datapoints available for training, such as annotated pictures of different species when making a model to predict the species of an animal. The Dataset is typically split into three groups: the training set, the testing set and the validation set. The training set is fed directly to the model during training for it to learn from, while the validation set is used intermittently during training to validate how well the process is going. The testing set is used after training is finished to bentchmark how well the model can be expected to perform on new data. As detailed in the study [30], there is no real consensus on the ideal proportions of a dataset's distributions. Common distributions are 70% training and 30% testing (having no validation set) (illustrated in image 2), and 80% training, 10% testing and 10% validation (illustrated in image 3). As the study found, there is no golden ratio to be found, as the larger the dataset is, the smaller the testing and validation sets can be, proportionally speaking. This is due to the fact that the goal of these two sets is to provide a good generalization of what unseen data the model will be encountering after it is finished training. Thus, having multiple examples with similar features in these sets does not provide a significant advantage, for the most part, the model is either capable of generalizing the features for all of them, or none of them. This also means that the smaller a dataset is, the larger the validation and testing sets should be, proportionally, which further escalates the issue of having a small dataset. When splitting a dataset it might be worth the effort to seek to minimize data leakage, meaning to avoid having similar images be present both in the training set and one of the others, which can lead to obtaining an inaccurate estimation of model performance on unseen data.

A dataset that over/under represents one or more classes is said to be imbalanced. The more pronounced this imbalance is, the more it will affect the model's performance. As observed by the study [8], models trained on imbalanced datasets tend to struggle with making predictions on elements from the under represented class(es), presenting lower accuracy (compared to other classes) and overfitting, both of which worsen the more pronounced the imbalance becomes.

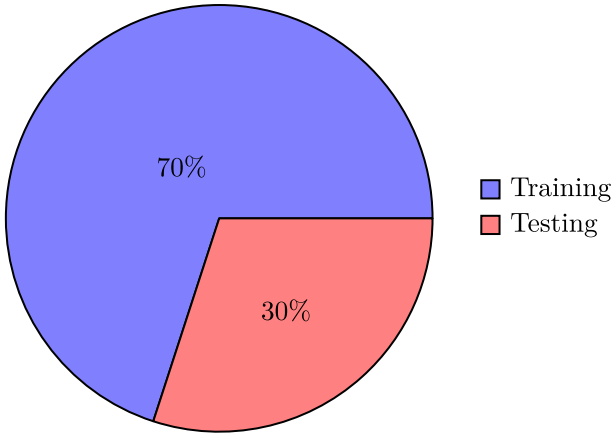


Figure 2: Typical dataset split for simpler or older workflows with no validation set (70% Training / 30% Testing).

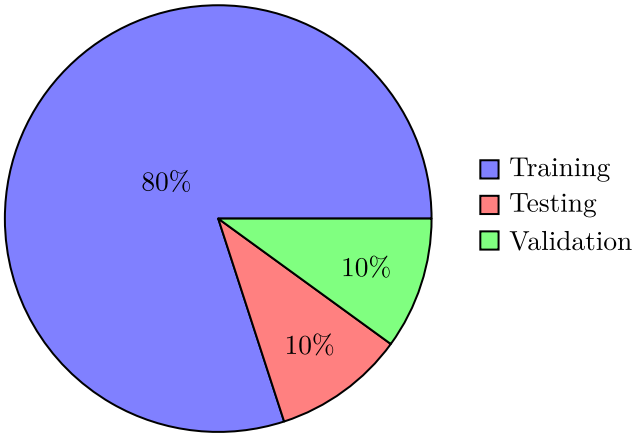


Figure 3: Balanced dataset split for larger datasets or modern workflows (80% Training / 10% Testing / 10% Validation).

2.1.4 *Data Normalization*

Data normalization describes the process of ensuring the various characteristics of data points fall within common ranges, such as [0,1] or [-1,1]. For instance, when describing the characteristics of real estate properties, one might work with features such as the year of construction, number of rooms, price history, number of nearby facilities, and total area, all of which exist on vastly different scales. These discrepancies typically lead to instability during training, resulting in a slower learning process, difficulty in optimization, and possible convergence issues. Normalizing the data can improve training stability and learning speed. The image 4

provides a simplified representation of the previous example. A common application of normalization is rescaling pixel values in images from a [0,255] scale to [0,1] when working with an image-based dataset. Another approach, Z-score normalization, standardizes data by transforming it to have a mean of 0 and a standard deviation of 1.

While data normalization involves directly changing the data that is fed into the model, other normalization approaches exist where the activations of a deep layer are themselves normalized, such as batch normalization and layer normalization.

Since the use of normalization techniques help stabilize the learning process, they allow for safer use of higher leaning rates, particularly when dealing with techniques like batch normalization and layer normalization. This was observed in the study [19] which when compared to using the baseline learning rate, saw an increase in achieved model accuracy of 2.1% at 30 times the original learning rate, also requiring considerably fewer steps to reach this stage in training (a little over a third). The image 5 illustrates the advantage provided by the use of batch normalization, which can be summarized as ensuring no singular feature disproportionately affects the weight updates during training (and prediction).

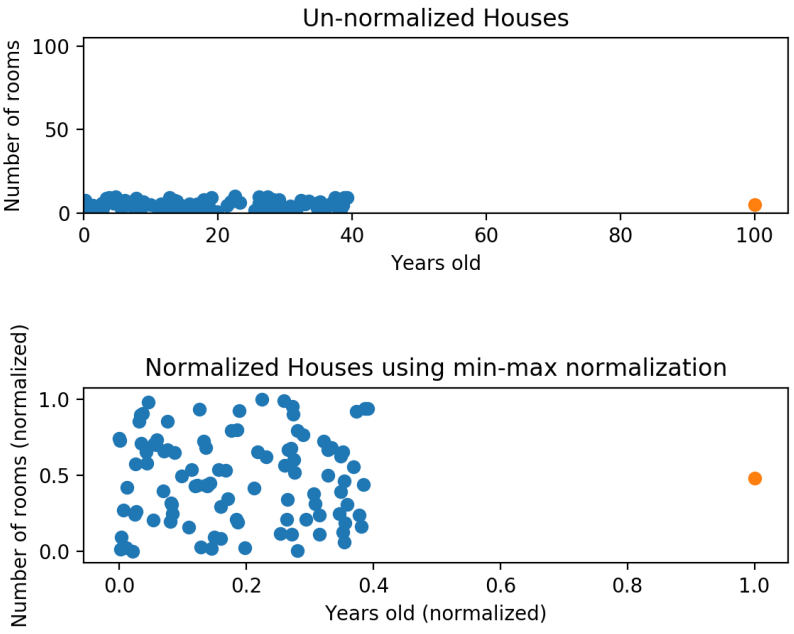


Figure 4: Normalization of number of house rooms ([37])

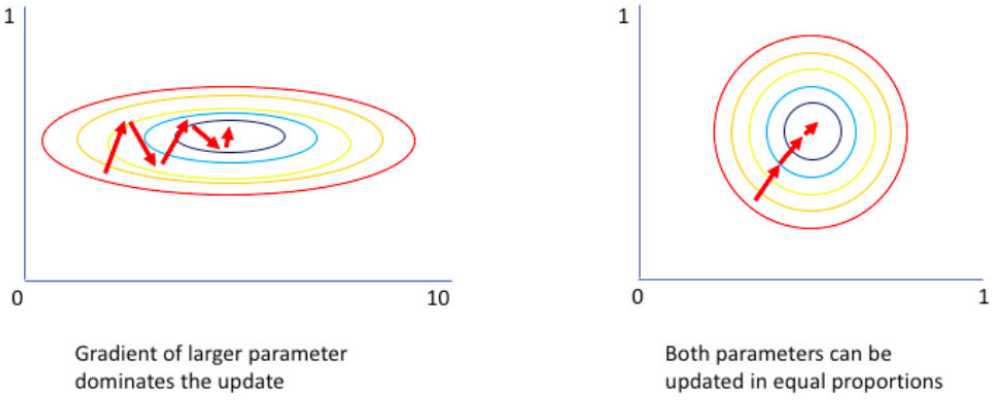


Figure 5: Comparison of training without and with batch normalization ([48])

2.1.5 *Data Augmentation*

When working with small or imbalanced dataset, it is common practice artificially increase the number of datapoints by taking existing ones and applying changes to them to produce new ones. This process can involve a myriad of techniques. Since the goal of data augmentation is to produce new datapoints that closely represent real data, it is advisable to avoid the use of multiple of these techniques at once, as this can result in a datapoint that has been too distorted to represent real data. In the case of images commonly used techniques are zooming in/out, adding noise, turning the image black and white, reducing/increasing brightness, removing parts of the image, rotating it and flipping it, some of which are illustrated in the image 6. While these are typically effective at producing sufficiently different datapoints to increase the dataset, they still raise data leakage concerns, among other issues.

In regards to removing select sections from an image, it is paramount that the features which are necessary to correctly categorize the image remain in the produced datapoint. The same logic can also be applied to zooming in, and to turning a the image black and white, in cases where color is the only distinguishing feature. The introduction of new patterns to the data is also a concern. In the case of adding noise to images, the model might learn to pick up on noise, rather then real features. This issue can be particularly felt when augmenting data to offset an imbalanced dataset. Given a minority class, if techniques like adding noise are used while the original dataset does not have a significant presence of images with noise, the model

might learn that noise is a feature of the minority class. This is not an issue if all classes receive a significant number of new datapoints with this augmentation.

In summary, small or imbalanced datasets can be improved through data augmentation, though the appropriate techniques to use vary between datasets.

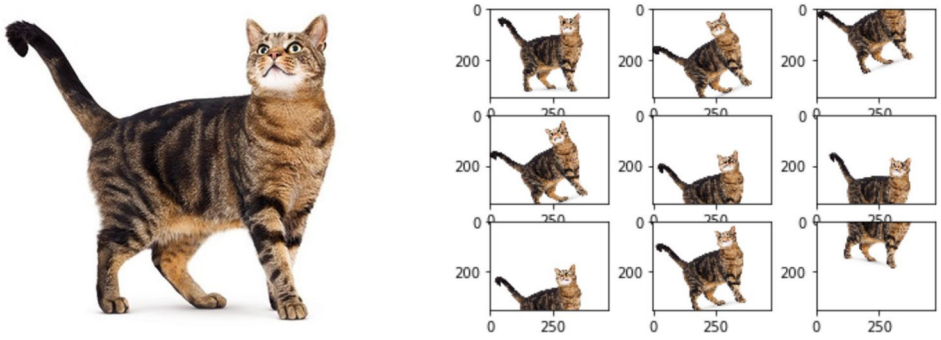


Figure 6: Basic image datapoint augmentation ([16])

2.1.6 *Training*

Training a model involves repeatedly going over the training dataset, where each complete go around the dataset is referred to as an epoch. During training the model's weights will be repeatedly changed with the aim of reducing the loss function, being changed at every step, which is determined by the batch size. For instance, given a batch size of 32, each step will involve having the model make predictions on the next 32 datapoints, after which the weights are adjusted. During the training process hyper parameters, such as the learning rate, will get gradually adjusted, based on how the model's performance evolves throughout the epochs. A model that is trained for too few epochs will not have enough time to learn the features in the dataset required to make accurate predictions, therefore making a large number of incorrect predictions both on the known data as well as unknown data. This is referred to as having an underfitted model. On the opposite end of the spectrum, a model that spends too many epochs going over the training data will begin to "memorize" it, become specialized in correctly predicting those particular datapoint, but having a significantly worse performance in unseen data. This is referred to as having an overfitted model. The goal with training is to achieve a well fitted model, where it has appropriately generalized the features of the dataset to be able to correctly predict both seen and unseen data. Training the model can be thought

of as trying to determin a function that describes the datapoints' destribution, as illustrated in the figure 7.

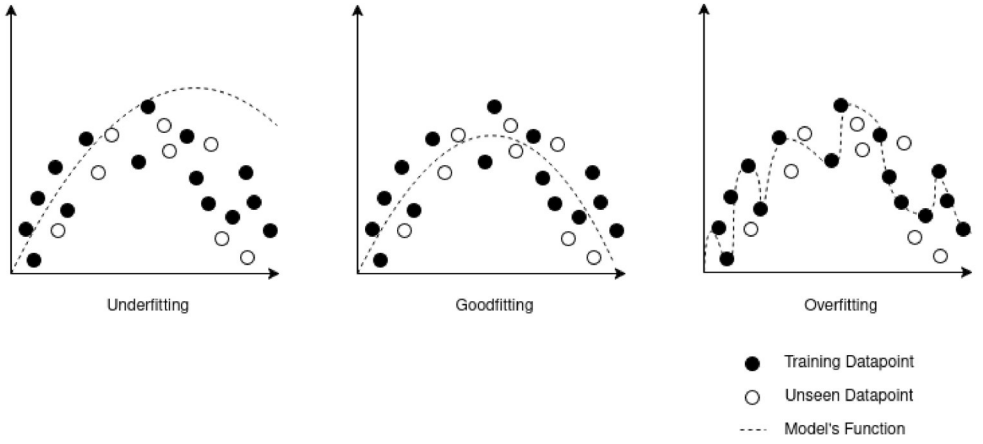


Figure 7: Model fitting comparison

The learning rate will determine how large the changes to the weights will be at each pass. If it is too low, the model will take longer to converge, requiring many epochs to train. On the other hand, if the learning rate is too high, the model is prone to getting stuck in extreme values or overfitting early. For these reasons the use of normalization techniques to stabilize learning, as well as loss functions that dynamically adjust the learning rate throughout the training process are common practice, allowing for a safer use of higher learning rates. Ultimately, the best learning rate will vary from model to model, and will require trial and error to determine, with the starting point typically being more on the conservative side.

The use of a validation dataset will entail having the model make predictions on the entire validation set at the end of each epoch, as a means of gauging how well the model has abstracted the features it has to learn. This allows for further adjusting of hyper parameters, as it provides a better overview of how well a model is performing, allowing for early detections of overfitting. To take advantage of this, it is common practice to use early stopping, a technique which involves saving the best performing weights for the validation set throughout the training process, and stopping the process early if there is no improvement or the performance deteriorates after a given amount of epochs.

The smaller the training dataset is, the more epochs will be necessary to learn the same features, as each epoch will itself be shorter. This creates an issue for small datasets, where by needing more epochs, they are more prone to overfitting. State of the art image classification models, such as YOLO [49], were trained on

very large datasets, Common Objects in Context (COCO) in YOLO's case, allowing them to have a good abstraction of basic features in the earlier layers. A common way to take advantage of this to minimize the impact of a small dataset is to use transfer learning, where an already trained model is used as the base for a new model, having its weights frozen, while the new layers are trained. Typically the is followed by some epochs of fine tuning, which involves unfreezing some of the later layers of the base model and using a lower learning rate (usually a tenth of the one used in the training process), to better specialize the model to the given task.

2.1.7 *Regularization*

Regularization techniques are essential in preventing overfitting during the training process, especially when working with limited datasets. These techniques work by constraining or modifying the learning process in a way that discourages overly complex models, improving the model's ability to generalize to unseen data.

One of the most widely used regularization methods is dropout ([45]). During each training step, a defined percentage of neurons in a given layer are randomly deactivated (set to zero). This prevents the model from becoming overly reliant on specific neurons and forces it to distribute learning across the entire network. To maintain the scale of activations, the remaining active neurons temporarily receive a proportional increase in their weights. Dropout has proven effective in reducing overfitting, though it may interact negatively with other regularization techniques, depending on the specific configuration.

Another category of regularization involves penalizing the model's weights to control their magnitudes. L1 regularization ([32]), also known as Lasso, promotes sparsity by pushing smaller weights toward zero and allowing larger weights to increase further. This effectively performs feature selection, enabling the model to focus on the most impactful input features. However, if applied too aggressively, L1 regularization can lead to overfitting, especially in cases where discarded features still contain relevant information.

In contrast, L2 regularization ([32]), or Ridge regularization, penalizes large weights more heavily, encouraging the model to distribute importance more evenly across features. This discourages the model from relying too strongly on a few specific features, thus improving its generalization capability. While effective at reducing overfitting, excessive use of L2 regularization may cause underfitting by suppressing important patterns in the data.

In practice, L1 and L2 regularization are often used together (a technique referred to as elastic net regularization) to combine the sparsity benefits of L1 with the smoothing effects of L2. However, both methods directly influence the weight values, which can lead to undesirable interactions when used alongside dropout. For instance, L1 may drive too many weights to zero when combined with dropout, while L2 may counteract dropout's intended temporary increase in weights for active neurons.

An alternative method that avoids such conflicts is max-norm regularization ([31]). This technique imposes an upper bound on the norm of each neuron's weight vector and scales weights down proportionally to remain within that limit. Max-norm regularization allows weights to grow only within a controlled range, maintaining their directional learning while preventing excessively large values. Notably, max-norm does not suffer from the same negative interactions with dropout, making it a preferred option when both techniques are to be used together.

It is worth noting that the study [45] showed that dropout can still have very positive results with L2, so wether or not their use together is appropriate will vary from case to case and will come down to experimentation. It did also find that pairing Max-Norm with dropout yielded better results than L2 paired with dropout however.

2.1.8 *Model Architecture*

A model's ideal architecture will vary between tasks, having room for large complexity and hyper specification. That being said, as illustrated in image 8, a model will typically have an input module, which receives the input and performs necessary changes to it (preprocessing), a backbone module, which extracts features from the input, a neck module, which further processes and refines the features extracted by the backbone, and a head module, which makes predictions based on the features processed by the neck. Adapting a model's architecture to a specific task might involve adding a "zero layer" offset to the input image, ensuring features in the edges of the input receive the same attention as the rest, or utilizing multiple heads to make different types of predictions on the same, or different segments of the features extracted by the backbone. On a more general scale, architectures can also be adapted by changing the amounts of resources used by the model. Most commonly, more layers are added to a model when attempting to improve its performance, accuracy wise, however, as displayed by the study [46], focusing solely on this as a means to scale up a model will have quickly diminishing returns. Instead the study

proposes scaling a model's architecture in depth (number of layers), width (number of neurons per layer) and resolution (input image size), verifying that any given one of these three options, when used by themselves, will have a significant "return on investment" initially, but see subsequent returns be far smaller. It proposes the following formula in how to scale them:

$$d \;\; \alpha^{\phi}, \quad w \;\; \beta^{\phi}, \quad r \;\; \gamma^{\phi} \tag{1}$$

$$\alpha \cdot \beta^{2} \cdot \gamma^{2} \approx 2 \tag{2}$$

where ϕ is a user-defined coefficient that determines the available computational resources, and α, β, γ control how those resources are allocated among depth, width, and resolution, respectively. As the formula suggests, the depth is the parameter whose scale influences the model the most, though the other two are still crucial for achieving the optimal performance.

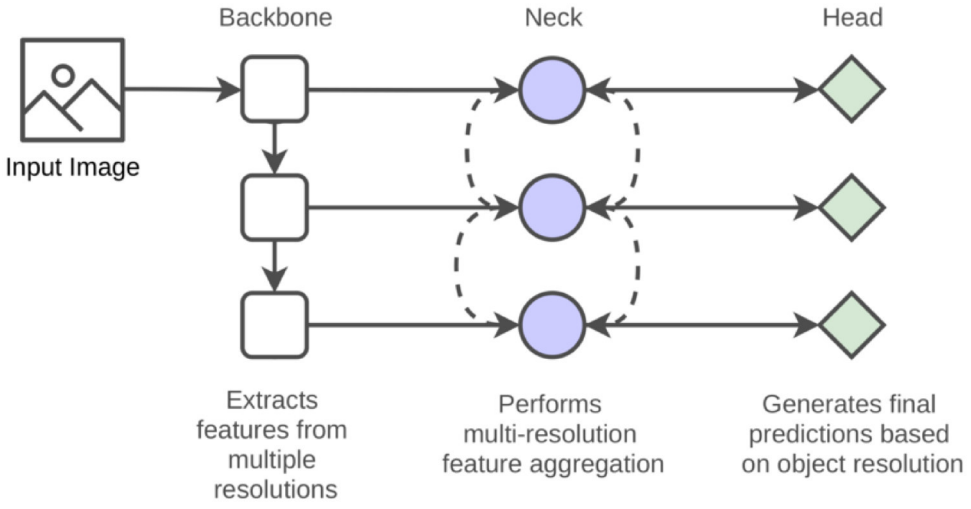


Figure 8: Example high level model architecture ([33])

2.1.9 *Quantization*

Quantization is the process of mapping a model's high-precision parameters and activations (i.e. 32-bit floating-point) down to lower-precision representations (i.e. 16-

bit floats, 8-bit integers, or even binary values). Common variants include dynamic range quantization, float16 quantization, and full integer (int8) quantization. It is typically applied post-training to produce a lighter-weight model optimized for edge devices, yielding substantially smaller model files and faster inference at the cost of causing some diminishment to the model's accuracy. Quantized models are often deployed using TFLite, a lightweight version of TensorFlow designed for mobile and embedded systems. TFLite converts standard TensorFlow models by applying quantization, pruning (removal of unnecessary/low-impact weights/connections), and graph optimizations (i.e., removal of redundant operations), enabling them to run efficiently on resource-constrained hardware. TensorFlow Lite distinguishes itself as a multi-framework model deployment and optimization platform by supporting the conversion of models from TensorFlow, PyTorch, and JAX into the FlatBuffers format (.tflite), enabling a wide range of state-of-the-art models to be used across myriad edge devices ([26]).

In the study [40], it was found that post-training int8 quantization alone shrank vision models by roughly 70–80% and cut inference latency by up to 60%, with accuracy losses under 2%. When combined with pruning, models saw up to 90% size reductions, though accuracy degradation rose to 4–6%. For large language models, int8 quantization resulted in models that were 2 to 3 times smaller, but with a 5–10% increase in prediction error. The authors also benchmarked TensorFlow Lite's quantization pipeline against comparable frameworks (i.e. PyTorch and ONNX Runtime), finding TFLite to be on par in compression ratios and inference speed for edge deployment, with only marginal accuracy differences.

2.2 OVERVIEW OF CORE TECHNOLOGIES

This section presents the main technologies relevant to this internship report and discusses their role in the problem domain.

2.2.1 *VGG*

VGG networks ([42]) marked a milestone in deep convolutional neural network design by demonstrating that a deep stack of small (3×3) convolutional filters could lead to improved performance on large-scale image classification tasks. At the time of its release, many of today's widespread normalization techniques, such as batch normalization, were not yet incorporated into network designs, so the original VGG

models relied solely on repeated convolutional and max-pooling layers followed by fully connected layers. Although VGG's straightforward, easy-to-understand architecture was revolutionary and remains a benchmark for performance on datasets like ImageNet, its high computational cost and memory demands, along with slower training and inference times, have led the research community to develop more efficient and robust architectures. Modern state-of-the-art models now integrate innovations such as residual connections and normalization layers, which improve both training stability and overall efficiency compared to the original VGG design.

2.2.2 *YOLO*

YOLO (You Only Look Once) ([35]) revolutionized the object detection scene when it was introduced by offering an approach its competitors were not using at the time. While traditional methods typically relied on computationally intensive sliding window techniques—scanning the image in patches and performing multiple inferences—YOLO treated object detection as a single regression problem ([54]). This allowed it to predict bounding boxes and class probabilities for multiple objects in a single forward pass of the neural network, resulting in dramatically improved inference speeds.

Although its initial accuracy lagged slightly behind other methods([35]), YOLO's real-time capabilities quickly established it as a leading option in the field. Since its original release, numerous improvements have been made, and it has been solidified as a state-of-the-art model for real-time object detection. Beyond detection, modern YOLO versions are capable of tasks such as object segmentation, pose recognition, and classification, among others.

A typical YOLO architecture consists of three main components: the backbone, neck, and head, each contributing to the model's ability to detect objects efficiently. The backbone is a convolutional neural network that processes the input image to extract feature maps capturing essential visual information. These features are then passed to the neck, which merges outputs from different stages of the backbone to create multi-scale representations—helping the model detect objects of various sizes by combining detailed spatial and abstract semantic information. Finally, the head uses these aggregated features to predict bounding boxes, objectness scores, and class probabilities across different regions of the image. It is through this streamlined pipeline that YOLO is able to perform object detection in a single pass, maintaining both speed and accuracy across diverse scenarios ([54]).

The versions of YOLO considered state of the art include versions 8 through 12, each introducing unique architectural refinements to the original concept. Some, like YOLOv8 ([9]) and YOLOv11 ([52]), are optimized for efficiency on lower-end devices, while others, like YOLOv12 ([49]), are designed to leverage the capabilities of more powerful hardware. Despite their differences, it is often recommended to experiment with multiple versions for a given task, as performance can vary in ways that are not always predictable based on the architecture alone ([51]).

Each version typically includes a range of model sizes—from nano to extra-large—giving users a wide selection that minimizes the need for manual architectural adjustments to meet specific resource or performance constraints. Additionally, these models are easy to convert to formats like TFLite for edge deployment, offering even greater flexibility.

Another notable strength of YOLO is its adaptability to different input image sizes. The models are designed to make predictions at three different scales, which helps maintain high accuracy across varying object sizes and allows for a degree of tolerance when images differ from the training resolution—as long as the objects themselves are not unusually large or small ([21]).

All things considered, YOLO remains a powerful and user-friendly choice for object detection and related tasks, balancing speed, accuracy, and hardware compatibility in a way few other models can match.

2.2.3 *Haar-based Object Detection*

Haar-like feature-based object detection methods rely on cascades of simple classifiers and boosted feature selection to detect objects rapidly with low computational cost [2, 24, 53]. These methods use integral images to compute features efficiently and are particularly effective for tasks like frontal face detection under controlled conditions. Unlike modern approaches such as YOLO, which perform end-to-end learning with convolutional neural networks and can handle a wide variety of object types and orientations, Haar-based methods are rigid and sensitive to changes in viewpoint, lighting, or occlusion [13]. Furthermore, because Haar cascades process regions independently using binary decisions, they are less effective at managing overlapping or clustered objects, resulting in increased false positives or missed detections when multiple objects appear simultaneously [2]. Nevertheless, in scenarios where computational efficiency outweighs accuracy—such as on low-power devices

or when detecting frontal faces in real time—Haar classifiers remain a viable, if limited, solution [24].

2.2.4 *Face API*

Face API is a javascript library implemented exclusively on top of tensorflow.js, which allows for models to be executed directly within a browser, instead of relying on the traditional approach of client-server communication. Face API comes with several models available to it designed face recognition. The "Single Shot MultiBox Detector (SSD) Mobilenet V1" and the "Tiny Face Detector" models are the main ones, as they both serve the same function of detecting which and where the faces in a given picture are. The first is more precise but also more computationally expensive option, based on MobileNet V1, while the latter, at the cost of struggling with smaller faces, achieves low enough resource requirements to be mobile and web friendly, being based on Tiny YOLOv2 ([10]). The remaining models server the purposes of face landmark detection, face and expression recognition and age and gender prediction. This library is state of the art when used with the "Tiny Face Detector" to achieve real time browser based face recognition, which ,while niche, displays the viability of tensorflow.js. It is important to note that any model in tensorflow (or adjacent) format can be converted to the TensorFlow JavaScript (TFJS) format, which quite simply separates the weights of the model into several 4MB files so that they can be cached by browsers, meaning this library can easily be supplemented with new models ([28]).

2.3 STATE OF THE ART

This section reviews recent studies and advancements in the field, highlighting state-of-the-art methods and their applications relevant to this work.

2.3.1 *Sound distinction*

The study [44] tackles a version of the cocktail party problem, the challenge of isolating vocal tracks from a musical mixture. In this work, the authors develop a convolutional deep neural network that learns to distinguish between the vocal component and instrumental accompaniment in polyphonic music. By training

the network on a large dataset of paired mixtures and isolated vocal tracks, the model learns to map complex time-frequency representations of mixed audio directly to its vocal-only counterpart. This approach effectively leverages convolutional layers to capture both local spectral features (the different frequencies or notes in the sound) and broader temporal patterns (how these frequencies change over time). These two elements allow the network to differentiate between sound sources, enabling it to separate the vocals from the background music. As a result, the model demonstrates a notable ability to extract clean vocal signals, even when the background instrumentation exhibits overlapping frequency content.

2.3.2 *Face API*

The study [27] demonstrates how the face-api.js library can be integrated into web applications to enable real-time, browser-native face recognition and expression analysis. It utilizes pre-trained models for detection, landmarking, recognition, and expression estimation, achieving over 99% accuracy on standard benchmarks like Labeled Faces in the Wild (LFW) (Labeled Faces in the Wild) dataset. The authors introduce a "face accuracy metrics" formula, which quantifies recognition reliability across static images, video frames, and live webcam input by measuring the match rate over time and conditions. To address security concerns, they propose lightweight counter-spoofing techniques—such as detecting facial movement, micro-expressions, and using challenge-response prompts—to prevent photo/video-based attacks. Additionally, they outline a role-based access control system where each face is linked to predefined user roles, enabling secure, client-side login without transmitting biometric data to a server.

2.3.3 *ResNet v2*

The study [18] explores the application of the Inception ResNet v2 architecture for early detection of breast cancer using ultrasound images. The researchers employ transfer learning to leverage pre-trained weights and enhance performance, while also applying data augmentation techniques—such as rotation, zoom, and rescaling, to increase the diversity of the training dataset. The dataset is categorized into three classes: Benign, Malignant, and Normal.

The model architecture combines the representational power of Inception modules, which apply multiple convolution filters of different sizes in parallel to capture

features at various scales, with the training stability of residual connections, which help prevent vanishing gradients by allowing gradients to flow directly through shortcut paths. This hybrid design enables the network to be both deep and computationally efficient. The model achieved 89.72% accuracy on training data and 90% accuracy on test data, significantly outperforming baseline Convolutional Neural Network (CNN) approaches. These results demonstrate the model's effectiveness in classifying breast ultrasound images and underscore the potential of advanced deep learning architectures in enhancing early diagnostic capabilities in medical imaging.

2.3.4 *VGG*

The study [55] proposed an enhanced VGG16 architecture to improve image classification performance. They modified the original model by reducing convolutional blocks, incorporating batch normalization layers, and adding global average pooling layers. These changes reduced the model complexity by 79% in terms of parameters, leading to faster training times and improved accuracy across six benchmark datasets. This approach addresses the high computational cost traditionally associated with the original VGG16 model.

On the other hand, in the study [17] VGG16 was optimized for logo classification by integrating it with the Enhanced Hunger Games Search (Enhanced Hunger Games Search (EHGS)) algorithm, a metaheuristic inspired by natural competition and survival strategies. EHGS is an evolutionary optimization technique that iteratively searches for the best hyperparameters, such as learning rate, batch size, and architecture parameters, by simulating competitive behaviors to explore and exploit the search space efficiently. Key enhancements of EHGS include a "local best" mechanism to intensify search around promising solutions and a "local escaping" mechanism to avoid premature convergence to local optima.

While evolutionary optimization techniques like EHGS can significantly boost model performance, they typically require training multiple candidate models during the search process, which increases computational resource demands and training time. This trade-off means that although the final optimized model often performs better, practitioners need to balance resource availability and optimization complexity when applying such techniques. The EHGS-optimized VGG16 model achieved a classification accuracy of 98%, outperforming other deep learning architectures like ResNet50V2 and InceptionV3 in the logo classification task.

2.3.5 *YOLOv10*

The study [1] applied YOLOv10 to detect pediatric wrist fractures in X-ray images. They introduced a dual-label assignment system during training, which integrates both one-to-one and one-to-many matching strategies. This approach allows the model to handle overlapping objects more effectively by assigning multiple ground truth labels to a single predicted bounding box. Consequently, each detection can correspond to multiple classes, enhancing the model's ability to distinguish between closely located fractures and other abnormalities. This modification led to a mean average precision (mean Average Precision (mAP)@50-95) of 51.9%, surpassing the previous YOLOv9 benchmark of 43.3%. The results demonstrate the model's effectiveness in medical imaging tasks, particularly in supporting diagnostic accuracy for bone fractures.

In the context of agricultural yield estimation, the study [39] evaluated YOLOv8 through YOLOv11 for detecting and counting green fruitlets in commercial orchards. Out of 22 tested configurations, YOLOv9 gelan-base achieved the highest mAP@50 score of 0.935, while YOLOv11n was noted for its inference speed at just 2.4 milliseconds. YOLOv10 models showed a strong trade-off between speed and accuracy, highlighting their potential for real-time object detection in outdoor environments with complex backgrounds.

Another study, [38], examined the performance of YOLOv9, YOLOv10, and Real-Time Detection Transformer (RT-DETR) models in real-time weed detection for smart-spraying systems. The research revealed that YOLOv10 models, particularly the smaller variants like YOLOv10n, achieved fast inference times—down to 7.58 ms per frame on an NVIDIA GeForce RTX 4090 GPU, while maintaining competitive accuracy. Thus making YOLOv10 a viable option for deployment in resource-constrained embedded computing devices commonly used in agricultural applications. The study outlines YOLOv10's suitability for real-time weed detection, by balancing speed and accuracy, which is crucial for efficient herbicide application in precision agriculture.

3

METHODOLOGY AND SYSTEM ARCHITECTURE

This chapter consolidates the methodology and system architecture used across the two developed systems: the clam classification/detection pipeline and the browser-based face recognition application. Each project is presented as a subsection. For each project the chapter first describes the evolution and final form of the requirements, then the development methodology, tooling and dataset preparation, the component-level architecture, and finally the evaluation strategy and success metrics. Additionally, each project's subsection includes the specifications of the hardware used for testing and development.

3.1 CLAM CLASSIFICATION AND DETECTION PIPELINE

This section describes the design and implementation of the clam classification and detection pipeline, outlining its requirements, development methodology, architecture, and evaluation process.

3.1.1 *Requirements and evolution*

The clam project began as a proof-of-concept with an incremental, milestone-driven development approach rather than a single, fixed specification. Early milestones focused on non-AI preprocessing to isolate candidate clam objects from the background; subsequent milestones introduced classification, improved normalization, dataset refinement and finally object detection suitable for edge deployment. Alongside these, there was also an initial requirement that the AI model developed would have to be converted to the TensorFlow Lite format, since the interpreter for this format is optimized for edge devices.

The final functional requirements were:

- classify clams by species (multi-class species label) and by conformity (e.g. identify broken shells, labelled "partida"),

- detect and localize each clam (bounding box / centroid),
- estimate physical size (for quality classification / routing),
- output information compatible with conveyor-belt actuation (servo control) so items can be routed to containers.

Key non-functional requirements:

- low inference latency suitable for edge devices (the initial prototype exhibited roughly one-second latency on a standard Personal Computer (PC) which was considered too slow),
- high classification accuracy (the presented demos ultimately reached accuracies in the high nineties),
- appropriate confidence levels between correct and incorrect predictions (meaning it should be clear when the model is not sure about its prediction),
- use of locally available compute resources (development and early training were performed on a single local PC with limited Graphics Processing Unit (GPU) capability).
- convertion of the model to TFLite format.

During the project the original constraint to train models strictly from scratch was relaxed in favour of transfer learning due to dataset size and compute limitations. Early attempts used conventional computer-vision preprocessing (thresholding, CCL like approaches) to reduce irrelevant background. Later stages moved to an AI-based object detector (YOLO) which also provided classification, allowing a simplification of the architecture by merging detection and classification into a single model.

3.1.2 *Development methodology, tools and dataset preparation*

Development followed an iterative milestone approach: small, testable deliverables were implemented and evaluated before changing or adding requirements for the next milestone. This approach matched the proof-of-concept nature of the work.

Tools and libraries used:

- Development environment: Visual Studio Code and Python (with virtual environments) for writing scripts and notebooks.
- Deep learning frameworks: TensorFlow was for both the VGG and Resnet models, to ensure ease of conversion to TFLite format; for the object dection

the project moved to PyTorch, utilizing Ultralytics ecosystem (the project used an Ultralytics YOLO model), a choice which was made with the knowledge that there exists support to convert YOLO models to TFLite [50].

- Annotation: MakeSense (web tool) was used to create and export bounding boxes in a format compatible with Ultralytics' YOLO expectations.
- Scripted tooling: custom Python scripts were written to compute evaluation statistics (per-class accuracy, confusion matrices, counts of true/false positives/negatives, average confidences).
- Edge integration prototype: a Python script was planned to be used to sample the camera feed at intervals, input crops/frames to the YOLO model, and apply servo configuration outputs to guide clam routing.
- Model conversion: TensorFlow Lite conversion and post-training quantization were performed in Google Colaboratory [36] rather than locally due to Python library version incompatibilities in the local workstation.

Dataset preparation highlights:

- early preprocessing used thresholding / binary conversion and CCL–like logic to place preliminary bounding boxes,
- for the YOLO pipeline the dataset was reprocessed and re-annotated (bounding boxes) using MakeSense to match the detector format,
- the augmentation strategy was class-aware: most augmentations focused on the minority class ("partida") to rebalance the present extreme class imbalance; a limited subset of majority-class images were augmented to avoid creating patterns the model could pick up on (i.e. all clams at an odd angle belong to the minority class),
- two images originally labelled "normal" were re-classified as "partida" after manual review (an $\approx 18\%$ increase in minority-class volume), due to visibly having a break in their shell,
- negative samples (images without clams) were added from public datasets (e.g. Kaggle) to improve the detector's robustness to background objects.

Due to long training times, especially felt when training YOLO (taking over three and a half hours to complete 60 epochs). much of the research was done while models trained in the background. This introduced some delay between identifying potentially viable improvements to the model, and actually testing them, but helped mitigate the resource constraint somewhat.

Test devices and hardware

The clam project was developed and evaluated using the locally available development workstation. The workstation served as the primary environment for coding, experimentation and runtime validation; it provided the reference environment for performance and latency measurements referenced in the Evaluation subsection. The workstation specifications used during development are summarized in Table 1.

Table 1: Device used for clam classification pipeline development and testing

Device	Workstation (ASUS TUF FX505GE)
CPU performance	2.20 GHz base / up to 4.10 GHz turbo
RAM	16 GB (ample for development and dataset preparation)
Role	Sole development and testing machine

The workstation served as the sole development and testing environment; details are listed in Table 1.

3.1.3 *Architecture and components*

Figure 9 presents the system context view of the clam project: it situates the system within its operational environment and shows primary external actors and data flows without exposing internal containers.

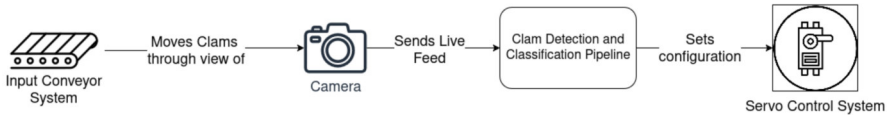


Figure 9: System Context diagram for the clam detection and classification pipeline. The diagram shows the Input Conveyor System feeding clams under the Camera, which provides image data to the Clam Detection and Classification Pipeline; the pipeline issues routing commands to the Servo Control System.

At a high level the system comprises:

- **Camera / image source:** acquires frames of the conveyor belt.
- **Inference node (edge device):** hosts the YOLO model and runs the inference pipeline. The model chosen for deployment was an Ultralytics YOLO variant (YOLO 11n was used in the final prototype) tuned for edge performance, being converted to TFLite format.

- **Actuation interface (servo control):** translates outputs (species label, conformity flag, size, bounding box/centroid) into servo commands to route clams to containers.
- **Supporting scripts / scheduler:** a Python script samples the camera feed at intervals, passes frames to the detector, post-processes detections (size estimation, confidence thresholds), and emits actuation commands.

Moving one level deeper, to the container view, Figure 10, the main components are the Image Acquisition Module, Detection Module, Classification Module, and Servo Configuration Module. Design decisions and simplifications:

- merge detection and classification into a single YOLO model (single-step inference) to reduce component handoff and lower end-to-end latency,
- select a light-weight base architecture for transfer learning (VGG16 was chosen in early transfer-learning experiments over heavier alternatives in order to reduce inference cost on constrained hardware),
- prefer smaller YOLO variants (edge-oriented v8/v11 family, specifically the nano versions) to balance accuracy and speed.

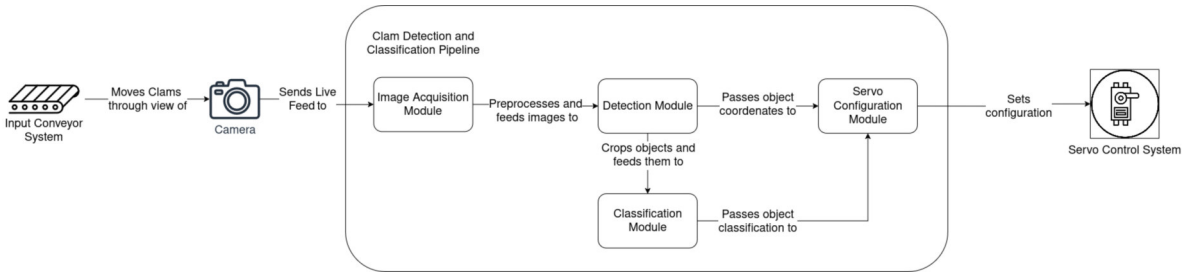


Figure 10: Container diagram of for the clam detection and classification pipeline. The Image Acquisition Module receives the live feed from the Camera, preprocesses and feeds images to the Detection Module, which crops objects and feeds them to the Classification Module; the Classification Module passes object classifications to the Servo Configuration Module, which sets the configuration for the Servo Control System.

3.1.4 *Evaluation and success metrics*

Evaluation tooling and metrics included:

- per-class and overall accuracy,
- confusion matrices and class-wise TP/FP/TN/FN counts,
- average confidence for correct and incorrect predictions,

- detector-specific metrics (bounding-box precision / recall / mAP) as provided by the Ultralytics evaluation utilities,
- practical runtime/inference measurements (latency) on the development workstation (Table 1) to confirm suitability for edge deployment.

Success criteria for the clam project focused on:

- achieving high classification accuracy (target: high 90s for demo-level performance),
- reducing inference time well below the initial ∼1s baseline on target hardware,
- producing sufficiently precise size and location estimates to enable accurate servo actuation on the conveyor.

3.2 BROWSER-BASED FACE RECOGNITION APPLICATION

This section presents the browser-based face recognition application, covering its evolving requirements, development workflow, system architecture, and evaluation strategy.

3.2.1 *Requirements and evolution*

The face recognition project was developed as a proof-of-concept with milestone-driven requirements. The initial requirement was to identify a viable way of running face recognition natively in the browser (so that processing happened on the client side rather than the server). During early exploration the Face API was identified as the most commonly used state-of-the-art solution for in-browser face recognition and was therefore selected. Only after this selection was the additional requirement introduced to make the webpage usable on lower-spec tablets with intermittent connectivity.

Final functional requirements:

- run face-recognition locally in the browser on low-spec tablets (no reliance on constant server connectivity),
- provide fallback authentication methods: a 4-digit Personal Identification Number (PIN) entry and voice-based authentication,
- enforce liveliness/expression checks to mitigate photo/video spoofing,

- store check-in/out events locally and synchronize with the server when connectivity is available.

Non-functional requirements (added after the in-browser approach was chosen):

- low resource usage to run on older tablets,
- robust offline initialization (the webpage and model artifacts must be usable while disconnected).

3.2.2 *Development methodology, tools and dataset*

The development approach includes the:

- incremental testing on representative hardware (tablet, smartphone, PC); emphasis on ensuring the app worked locally on the constrained tablet hardware,
- manual operational testing (small set of registered users) rather than large-scale metric-driven evaluation, because the primary concern was practical behaviour on the target tablet.

Regarding the tools and libraries used:

- Development: Visual Studio Code and standard web tooling (HyperText Markup Language (HTML)/Cascading Style Sheets (CSS)/JavaScript (JS)).
- Browser models: Face API models in TFJS format (which can be stored in browsers' cache) were used to run recognition directly in the browser. Face API was selected after the initial feasibility study as the most commonly used state-of-the-art client-side solution.
- Local Hypertext Transfer Protocol Secure (HTTPS) development: ngrok ([41]) (used to facilitate HTTPS testing of the webpage during development because mobile browsers enforce secure contexts).
- Offline / caching: service workers were used to cache the webpage and model artifacts so the tablet could initialize even when disconnected.
- Local storage: browser storage was used to log check-in/out events locally until a network connection was available for synchronization.

As for the face dataset and test samples:

- The testing dataset was intentionally small (a few profile-like images obtained from public sources plus four pictures of workers provided by the hosting

company); the models used were pre-trained and the small dataset served only for registration/test purposes rather than for model training,

- Commonvoice.mozilla.org [29] was identified (but not used in this work) as a potential source for voice samples if voice authentication training were required.

Test devices and hardware

Testing for the browser-native face recognition application used the shared development workstation and two additional consumer devices: a target deployment tablet and a modern smartphone. The device specifications are summarised in Table 2; these devices were the basis for the runtime observations described in the Evaluation subsection.

Table 2: Relative performance of devices used in the browser-based face recognition project

Device	**CPU / RAM performance and role**
Workstation (ASUS TUF FX505GE)	2.20 GHz base / up to 4.10 GHz turbo; 16 GB RAM
Smartphone (Xiaomi 220117sy)	up to 2.05 GHz; 6 GB RAM (+2 GB extension)
Tablet (Huawei MediaPad T5)	2.36 GHz; 3 GB RAM — Target for deployment

3.2.3 *Architecture and components*

Figure 11 presents the System Context view of the face recognition project: it situates the system within its operational environment and shows primary external actors and data flows without exposing internal containers.

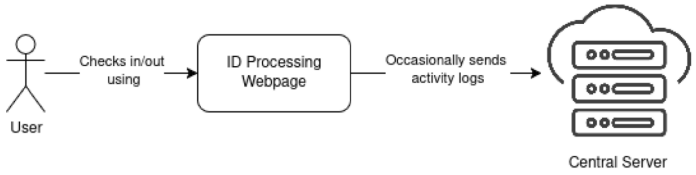


Figure 11: System Context diagram for the browser-based face recognition system. The User interacts with the Face Recognition Web App, which operates locally on the tablet and synchronizes with the Central Server when connectivity permits.

At the system level, the main external relationships are:

- **User:** provides authentication input (face, PIN, or voice) through the tablet.

- **Face Recognition Web App (system under design):** runs locally in the browser, handling recognition and authentication while caching models and logs.
- **Central server (external system):** stores the canonical face list and aggregated history, synchronizing with the client when connectivity is available.

Moving one level deeper, to the container view, Figure 12), the main components of the tablet client are described.

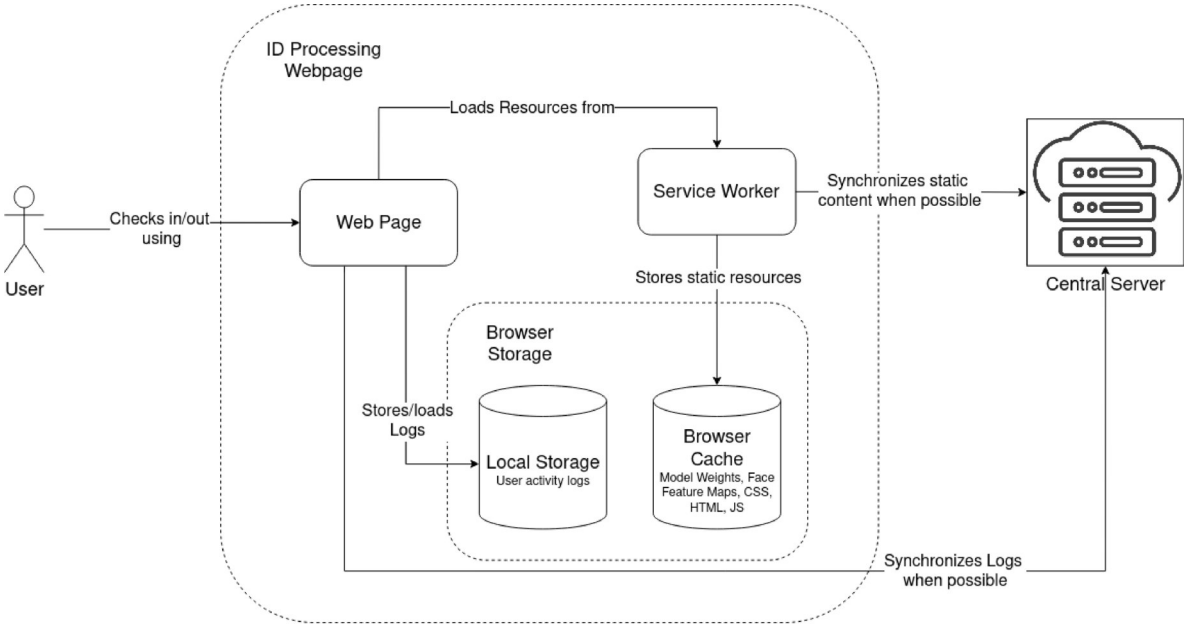


Figure 12: Container diagram for the browser-based face recognition system. The Web Page interacts with the User for check-in/out, loads resources from the Service Worker, and stores/loads logs in Local Storage. The Service Worker stores static resources in the Browser Cache and synchronizes static content and logs with the Central Server when possible.

The main components of the tablet client are:

- **Tablet client (web app):** loads the page, local models (TFJS), and handles user interactions (face capture, PIN entry, optional voice capture).
- **Service worker / cache:** caches webpage assets and model files to allow offline initialization.
- **Local storage / queue:** stores check-in/out events and logs offline, synchronizing once connectivity returns.
- **Central server:** hosts the webpage (optional), propagates updates to trusted face lists, and records check-in/out history.

Client–server interaction model:

- the client initiates all recognition locally using cached models; when online, it fetches updates (trusted faces, configuration) and uploads queued events,
- synchronization attempts occur periodically to tolerate unstable connectivity.

Liveliness / spoof prevention:

- an expression-detection model was chosen for liveliness checks after landmark-based blink detection proved ineffective,
- multi-factor authentication (face + PIN + voice) improves robustness against spoofing.

3.2.4 *Evaluation and success metrics*

Testing was primarily operational and device-focused:

- register a small set of users and verify the browser-based Face API correctly detects/encodes their faces,
- manual verification runs under typical lighting and device conditions to confirm accuracy and liveliness detection,
- measure latency and resource usage on the third-party tablet (Huawei MediaPad T5, Table 2) to confirm acceptable responsiveness,
- track confidence levels, recognition accuracy, and synchronization behaviour (queued events) as metrics of success.

3.3 COMMON CONSTRAINTS, TRADE-OFFS AND REFLECTIONS

Both projects were developed on a single local PC with limited compute. That constraint drove multiple design choices:

- preference for transfer learning and smaller model variants to reduce training time and inference cost,
- offline/edge-focused designs to avoid reliance on server compute,
- pragmatic dataset augmentation strategies (class-aware augmentation) rather than large-scale synthetic generation.

The milestone-driven, proof-of-concept approach encouraged focusing on the smallest useful integrations first (working demo), then iterating towards production-oriented concerns (latency, robustness, offline behaviour).

4

IMPLEMENTATION AND DEVELOPMENT

This chapter describes the practical work of turning earlier designs into working systems. It focuses on the two delivered projects — the clam classification/detection pipeline and the browser-based face recognition application — and explains how each was developed, evaluated and deployed for constrained hardware.

Each project is presented as a self-contained subsection. For each one I first summarise the evolved requirements, then describe the development methodology, tooling and dataset preparation, followed by the component-level architecture and implementation details. Each subsection concludes with evaluation procedures, key results and the hardware used for testing.

4.1 DEVELOPMENT PROCESS

This section details the step-by-step development process for both projects, covering detection, classification, dataset preparation, and the implementation of different model architectures.

4.1.1 *Clam classifier and detector*

This subsection presents the clam classification and detection pipeline is described in detail, including the evolution from preprocessing-based methods to an AI-based object detection and classification system.

Object Detection and segmentation

The first step in developing the clam detection pipeline involved traditional image-processing techniques to isolate individual clams before attempting AI-based segmentation.

Connected-Component Labeling

Before employing AI-based methods, an initial effort was made to detect and isolate the target objects (clams) in images using CCL. The main challenge was that each image contained multiple objects, but only the center-most object was of interest. The goal was to automatically crop the images to preserve the central object while removing others, retaining as much of the original image as possible.

The approach involved the following key steps:

- **Preprocessing:** Each image was converted to grayscale and then binarized via thresholding. This process generated a binary (black and white) image that distinguished foreground objects from the background.
- **Morphological Operations:** To connect parts of objects that were fragmented due to noise or shadows, dilation followed by morphological closing was applied. These operations helped merge closely positioned or partially disconnected components, preventing a single object from being mistakenly segmented into multiple parts.
- **Connected Components Analysis:** The binary image was analyzed to identify connected components representing candidate objects. For each component, bounding box coordinates, area, and centroid location were computed.
- **Selection of the Center-most Object:** Since the target clam was always approximately centered in the images, and had a consistent minimum size, the component closest to the image center with an area exceeding a specified threshold was selected as the main object. This filtered out smaller or peripheral objects.
- **Cropping Strategy:** To isolate the main object, cropping boundaries were defined by examining the spatial positions of other detected components relative to the main object. The crop was adjusted to exclude objects completely above, below, or to the sides of the main object, thus removing them while preserving the main object's context and maximizing retained image area.
- **Metadata Preservation:** The cropped images were saved with their original Exchangeable Image File Format (EXIF) metadata intact, ensuring no loss of important image information. This is relevant since the labels of the dataset were stored there.

This *Threshold-based Binary Segmentation* method was applied to the full base dataset of 749 target images. As detailed in the Section 5.3.1, this technique worked well for a vast majority of the dataset, though it did have a consistent limitation,

the detected objects shadow was also accounted as part of the object, as can be seen in Figure 29b.

To progress this technique from detection to segmentation and to attempt to minimize the impact of the object's shadow, the function `cv2.findContours` was used, changing the bounding box into a contour around the object's edges. The use of this feature seemed to require a high setting for strictness, as it was not providing significant improvements with the shadow situation, also removing small portions of the object (see Figure 29a). Increasing this setting did remove a significant portion of the shadow, but it also exarcebated the portions of the objects that were being removed, affecting some objects more than others (see Figure 29).

Since adjusting the strictness of `cv2.findContours` could not provide the desired result consistently, and adjusting the way the preprocessing was done in regards to the transforming the image to a black and white binary did not seem like a viable way to get rid of the shadow either (many of the objects were already black and white, and the shadow would fall within their color spectrum), there were no more clear ways to improve this method, thus no further efforts were made to do so.

IMPLEMENTATION DETAILS The method was implemented in Python 3, making use of three main libraries:

- **Open Source Computer Vision Library (OpenCV):** Provided the core image-processing functions, including grayscale conversion, binarization by thresholding, morphological operations (dilation and closing), connected-components analysis, and contour extraction. OpenCV was also used to draw visual indicators (e.g., green contours) and to save intermediate outputs such as binarized images.
- **Numerical Python (NumPy):** Used to create and manipulate the morphological kernels (e.g., the 7×7 structuring element) and to support numerical calculations such as distances between component centroids and the image center.
- **Python Imaging Library (PIL):** Employed for metadata handling and final image saving. Since the dataset's labels were stored in the EXIF metadata, PIL's `Image` module was used to read, preserve, and reattach this information when cropped outputs were saved.

The full pipeline was implemented as a custom Python function that processes all images in a given directory. This function performs the preprocessing, segmentation,

component analysis, cropping, and output saving automatically, ensuring consistent treatment across the dataset of 749 images.

Object Classification

Once clams could be reliably detected and cropped, the next focus was on classifying them by species using deep learning models trained from scratch.

Pre Transfer Learning

As requested, the initial efforts to develop a model capable of classifying the clams by species was designed to be trained from scratch. The exact amount of layers and the number of their neurons varied a lot in this stage, though the input image size remain consistent at 224x224. To account for the significant imbalance in the dataset (check Section 4.1.1), a custom implementation of focal loss function was used, ensuring loss favored minority classes. It differed from the traditional implementation of the function by extending it to the multi-class case and allowing the inclusion of explicit per-class weights, rather than relying only on a single scalar balancing factor. This removed the need to limit the data points of the majority class to balance the dataset. Both in terms of regularization, L1 Regularization (Lasso) (L1) and L2 Regularization (Ridge) (L2) were used at this stage, along with with dropout, with various values. Further, to remove the possibility of the model picking up on patterns in the dataset order, its order was shuffled every epoch. The optimizer used was Adaptive Moment Estimation (Adam), and the layers used Rectified Linear Unit (ReLU) for their activation function, except for the head of the model, which used softmax, as it outputs the actual predictions of the model. The models created at this stage had a lot of difficulty learning, leading to a lot of experimentation with the dataset in terms of augmentation (check Section 4.1.1), and the testing of more niche techniques, such as label smoothing (since one class would typically dominate significantly) and using a lower batch size (16) in training to introduce more noise. Since the model did not seem to be able to learn at this point, most likely due to being too small, VCC16 was selected as the basis for what the network's architecture should look like, as it was the smaller version of an outdated, but renown model, which was developed using only basic layers, as illustrated in Table 3, meaning it should be easier to understand and adjust for my use case. Due to its sheer size however, it was not feasible to train it from scratch, and attempts at making smaller versions of it that followed the same overall structure yielded no positive results still. As such, the project progressed to the next stage, using it as a base for transfer learning.

Table 3: VGG16 Architecture Overview [43]

Layer Type	**Output Shape**	**Number of Filters / ts**
Input	$224 \times 224 \times 3$	-
Conv3-64	$224 \times 224 \times 64$	64
Conv3-64	$224 \times 224 \times 64$	64
MaxPool	$112 \times 112 \times 64$	-
Conv3-128	$112 \times 112 \times 128$	128
Conv3-128	$112 \times 112 \times 128$	128
MaxPool	$56 \times 56 \times 128$	-
Conv3-256	$56 \times 56 \times 256$	256
Conv3-256	$56 \times 56 \times 256$	256
Conv3-256	$56 \times 56 \times 256$	256
MaxPool	$28 \times 28 \times 256$	-
Conv3-512	$28 \times 28 \times 512$	512
Conv3-512	$28 \times 28 \times 512$	512
Conv3-512	$28 \times 28 \times 512$	512
MaxPool	$14 \times 14 \times 512$	-
Conv3-512	$14 \times 14 \times 512$	512
Conv3-512	$14 \times 14 \times 512$	512
Conv3-512	$14 \times 14 \times 512$	512
MaxPool	$7 \times 7 \times 512$	-
Flatten	25088	-
Fully Connected	4096	4096
Fully Connected	4096	4096
Fully Connected (Output)	1000 (or number of classes)	1000

Transfer Learning With VGG16

At this stage of the project, VGG16 was adopted as the base model for transfer learning. The pretrained convolutional layers of VGG16 were used as a feature extractor, and several custom layers were added on top to adapt the network to the specific task of classifying clams into three species. These layers were initially just the ones already used in the previous stage, when making the light weight models, essentially just mashing the two together, which included the same normalization techniques, and the same loss function was still used. While this did produce a model capable of learning the task at hand, it fell short, never surpassing an accuracy distribution of 80-60-60 between the three classes, even after extensive tinkering with

the hyper parameters, class weights for the loss function, and dataset (check Section 4.1.1). The use of label smoothing and small batch size also remained present.

At this point the project went on hiatus, and when it was picked backup, the choice was made to look for new normalization options. At this stage, layer normalization and batch normalization were both tried, and batch normalization was elected as the appropriate option to use between the two, both due to displayed better improvements to the model, and due to being generally accepted as the appropriate option between them for image classification. The improvements seen with the addition of batch norm were sufficient to no longer having a need to continue looking for more normalization, as each class was now consistently achieving above 90% accuracy. The choice to stop investigating further improvements was also driven by the time constraint of being close to a presentation deadline here. Additionally, previously used techniques which were more niche were no longer used, namely the label smoothing, and the dataset was also simplified (check Section 4.1.1).

Using multiple batch normalization layers had very steeply diminishing returns, while also making the model much slower, for this reason, only a single layer was used. Since batch normalization benefits from a larger batch size, it was increased to 64 (from 16), and since it accelerates convergence, although it typically makes higher learning rates safer, it was actually reduced by 25%, from 0.001 to 0.0075, as it was already converging quite quickly, taking around 10 epochs after the reduction. After this training, fine tunning was done, unfreezing the last block of layers (block 5) of the base model (VVC16) with the learning rate reduced to a tenth of its normal amount (becoming 0.00075), and trained for an additional 5 epochs.

In the final version of the model, L1 was no longer in use, L2 was still applied to each layer added after the base model, with a lambda value of 0.0005 apart from the dense layers after the flatten layer, which used a value of 0.001. Dropout followed a similar distribution, being set to 0.33 for most layers, but 0.5 in the final convolution layers, also being present right after the base model a a value of 0.5. This version achieved very promising results, as detailed in 5.3.2, and as such, the changes made here would influence the next stage of the project. The architecture of this model is displayed in the Table 4

Transfer Learning With ResNet

This development step was very short lived, due to the results found with YOLO regarding classification, as detailed in Section 5.3.2. ResNet, whose architecture is represented in Table 5, was chosen to replace VCC16 as the base model at this stage

Table 4: Architecture of the Transfer Learning Model Based on VGG16

Layer (Type)	**Output Shape**	**Parameters**
VGG16 (Functional)	(None, 7, 7, 512)	14,714,688
BatchNormalization	(None, 7, 7, 512)	2,048
Dropout	(None, 7, 7, 512)	0
Conv2D	(None, 7, 7, 256)	1,179,904
Dropout	(None, 7, 7, 256)	0
Conv2D	(None, 7, 7, 256)	590,080
Dropout	(None, 7, 7, 256)	0
Conv2D	(None, 7, 7, 256)	590,080
Dropout	(None, 7, 7, 256)	0
Flatten	(None, 12544)	0
Dense	(None, 1024)	12,846,080
Dropout	(None, 1024)	0
Dense	(None, 1024)	1,049,600
Dropout	(None, 1024)	0
Dense (Output)	(None, 3)	3,075

due to the need to reduce inference time, since it is both much lighter, and a more recent model, having had access to new techniques that VCC did not. Only two changes were made to the model's structure aside from changing the base model, the addition of a zero layer, which adds a single line of zeros around the input received by the model (this ensures details in the edge of the image are given the same importance as details in other positions), and the use of the MaxNorm weight constraint, which helps stabilize the training, being commonly used with dropout. The choice to add the zero layer at this stage was due to it being meant to integrate with YOLO, receiving cropped images constructed from the former's output, which were expected to be very tight fits. The larger change for this stage form the previous one was the return to a more complex dataset regarding augmentation (check Section 4.1.1). While the model did achieve the goal of having low latency, without sacrificing accuracy (check Section 5.3.2), as previously stated, YOLO made its use unnecessary, thus it was pursued no further.

YOLO

YOLO was selected for the object detection component of the project, but since it also performs classification, the model was tested with a data set to perform both, but predicting on wether the clams had non comformaties to indicate their shells

Table 5: ResNet-50 Architecture Overview [14]

Layer / Block	**Output Shape**	**Number of Filters / Units**
Input	$224 \times 224 \times 3$	-
Conv1 (7x7, stride 2)	$112 \times 112 \times 64$	64
MaxPool (3x3, stride 2)	$56 \times 56 \times 64$	-
Conv2_x (3 Bottleneck blocks)	$56 \times 56 \times 256$	64, 64, 256
Conv3_x (4 Bottleneck blocks)	$28 \times 28 \times 512$	128, 128, 512
Conv4_x (6 Bottleneck blocks)	$14 \times 14 \times 1024$	256, 256, 1024
Conv5_x (3 Bottleneck blocks)	$7 \times 7 \times 2048$	512, 512, 2048
Average Pooling	$1 \times 1 \times 2048$	-
Fully Connected (Output)	1000 (or number of classes)	1000

being broken. Given that YOLO performs object detection, it was first necessary to make the labels for the dataset, and since the dataset needed to be adjusted specifically for the new task of having multiple objects in view at once, as well as adjusting for the imbalance in data points between classes, a new dataset was created from the original one, as detailed in Section 4.1.1. Since Ultralytics's library is built on Python Machine Learning Library (PyTorch), the development changed to PyTorch at this stage, introducing some complexity in converting models to TFLite. Since YOLO has many different versions to pick from, it was necessary to test the ones that seemed most appropriate for the scenario. Initially, YOLO8 nano and small as well as YOLO11 nano and small were planned to be tested, but during the training of the YOLO8 small version it became apparent that small models were too large, so neither small model was trained to completion. There were two main issues with the small models, the first being that that they require, and second that their training time was a lot longer, taking many more hours to train, even with lower settings. For both of the nano models, the image size used was 640x640, as is standard with YOLO, and the batch size of 48, which was the highest the workstation could handle whilst training them. Since YOLO uses batch norm, the batch size can have a significant impact, as is the case for image size, for the small model, these values had to be set to 36 and 480x480, still exhausting the workstation's resources.

Version 11 nano was selected due to its speed advantage over version 8 nano. After this stage, since the results for classification far surpassed expectations (see 5.3.2), it was decided that YOLO would also perform the classification of species of clam, which required adjusting the labels in the dataset to include the species when their shells were not broken. At this stage, there was also more experimentation

with the training process of the model, setting the dropout rate to 0.33 (which affects each layer), a higher learning rate (from 0.01 to 0.015), fine tuning, and transfer learning.

Transfer learning did not appear to provide any advantage over regular training in the training processes, having been tested with several different layer freezing target, still taking just as long to converge and still having similar statistics. For this reason it was not used in the final solution. In the final solution, the model was initial trained for 60 epochs, then fine tuned for another 30, with a learning rate a tenth of the one used in training (0.0015), set to freeze the first 5 backbone blocks.

Working with the models proved challenging without the use of Ultralytics' library, due to it having a confusing output pattern, so even after converting the models to TFLite, they are still used with it. Since the built in Non-Maximum Suppression (NMS) function would at times display multiple boxes stacked on top of one another if they were of different classes (see Figures 13 and 14), a custom one was developed, which granted finer control over the results received, now being able to display a single box, as well as easily check how high the confidence of each predictions were, to decide when a broken shell might have been misidentified as one of the other classes, or more easily mark a clam for manual review if multiple classes are of a high confidence.

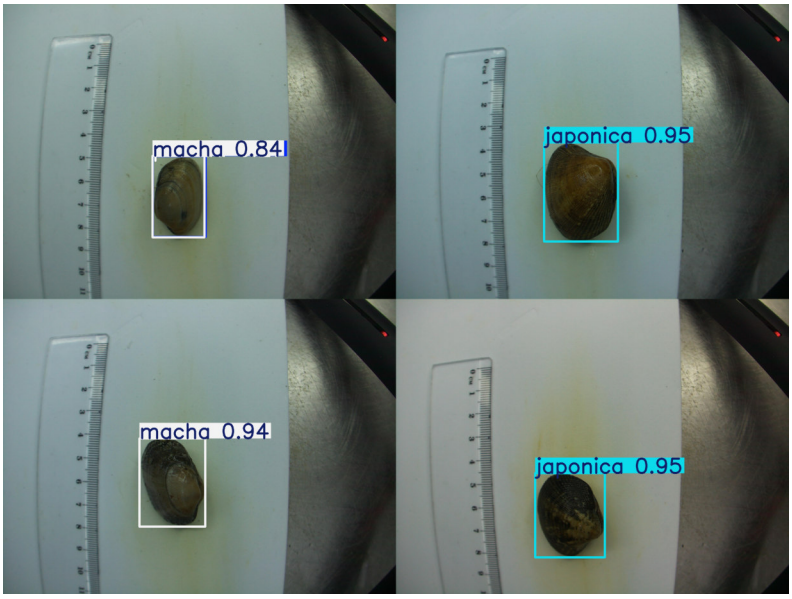


Figure 13: Example of stacked bounding boxes on top left clam.

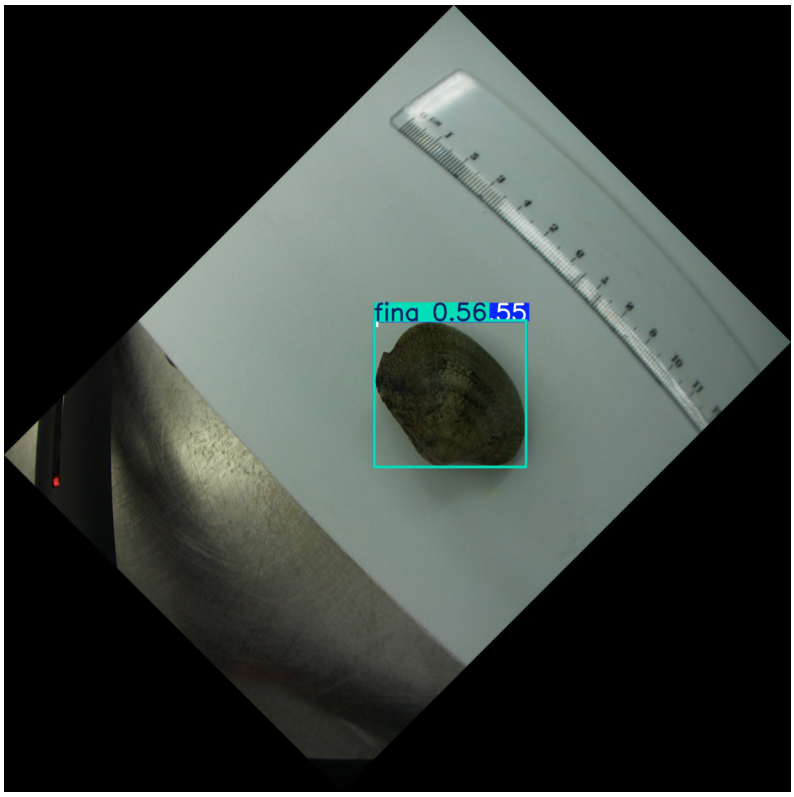


Figure 14: Example of stacked bounding boxes on a single clam.

Dataset

The dataset available for training the model consisted of 749 images of 4 different clam varieties, two of which belong to the same species. As such, the set was divided into 3 species: "fina", "macha", and "japonica", being unbalanced in a relative ratio of approximately 1:1.03:1.59, with "japonica" being the most represented and "fina" the least.

During the initial VCC16 experiments, a wide variety of augmentations was tested: kernel-level changes, greyscaling, rotations, flips, zooming in/out, cropping so that only certain regions of the object remained, and additive noise, at times combined. However, these more aggressive transformations produced negligible or negative effects on validation behaviour. Because the VCC16 model only began showing promising results very close to the demo deadline (where the requirement was simply to present a proof of concept that an AI could distinguish clam species), there was no time to revisit a more diverse augmentation strategy. For that reason, the augmentation set for VCC16 was pared back to a minimal, conservative subset, containing just the zoomed in and bounding box bearing imagesproduced by the

CCL done in the earlier stage of the project (see the original bounding box samples in Figure 29b for an example), in order to minimize potential non obvious destabilizers to the model's performance.

When experimentation moved to ResNet, the dataset itself was expanded: in addition to the original images, it also included a manually cropped version of the dataset. These crops were more tightly fitted around the clams than the CCL-based crops, and were prepared manually as part of the later work on the YOLO dataset.

The augmentation set was also broadened again, though with a more carefully tuned approach. The transformations applied in this phase were:

- Zoom in (no zoom out),
- Stretching and compressing,
- Fixed-angle rotations at $0°, 45°, 135°, 225°, 315°$,
- Random blackout squares via `apply_blackout` (covering 10–30% of the image with fewer but larger squares),
- Flipping (horizontal, vertical, and both).

These augmentations were sometimes applied in combination, to increase variability in the dataset. Figures 15, 16 and 17 show cropped examples of such augmentations applied to the ResNet dataset.

For the YOLO models, augmentation was deliberately kept closer to realistic production scenarios, primarily on account of the acute imbalance between the two classes that were being used in the initial test ("normal" and "partida", at a staggering 57.6 : 1 balance before augmentation). Augmentations included rotation, flipping, and placing multiple clams within a single image to simulate the conveyor-belt setting, and were applied to all samples of the "partida" class multiple times, while only to some of the "normal" class, still producing more augmentation of the "normal", to ensure the model did not learn to look for artifacts of augmentation (i.e. black corners from rotation) to find instances of "partida". Additionally, since the sample size was so small for the minority class, data leakage became a more serious concern, so a single sample and all of its augmentations was deliberately reserved to the test dataset in early experimentation, to ensure proper generalization. After this was verified, this constraint was removed, as the model would benefit more from having more variety in datapoints for the minority class, than it would in having proof of generalization at this stage, as it would likely be tested with real world clams in the next demo.

In the first iteration of this dataset, some augmented images included overlapping clams, but this was done based on a misunderstanding of the intended conveyor belt system's functionality. In practice clams would never be stacked. Once this was clarified, those overlapping images were removed, as the models struggled specifically with them, while performing adequately on the non-overlapping augmented samples (see Figure 18).

Regarding the split between training, validation and testing sets, for both the VCC16 and ResNet models, the dataset was divided into training, validation, and test sets with an 80/10/10 ratio; however, since the results of validation were not leveraged in any way during training, that set was merged with the test for evaluations in those frameworks. In contrast, while the YOLO dataset followed the same 80/10/10 split, it retained a strict separation between validation (used only for training feedback) and the test set (reserved exclusively for final evaluation).

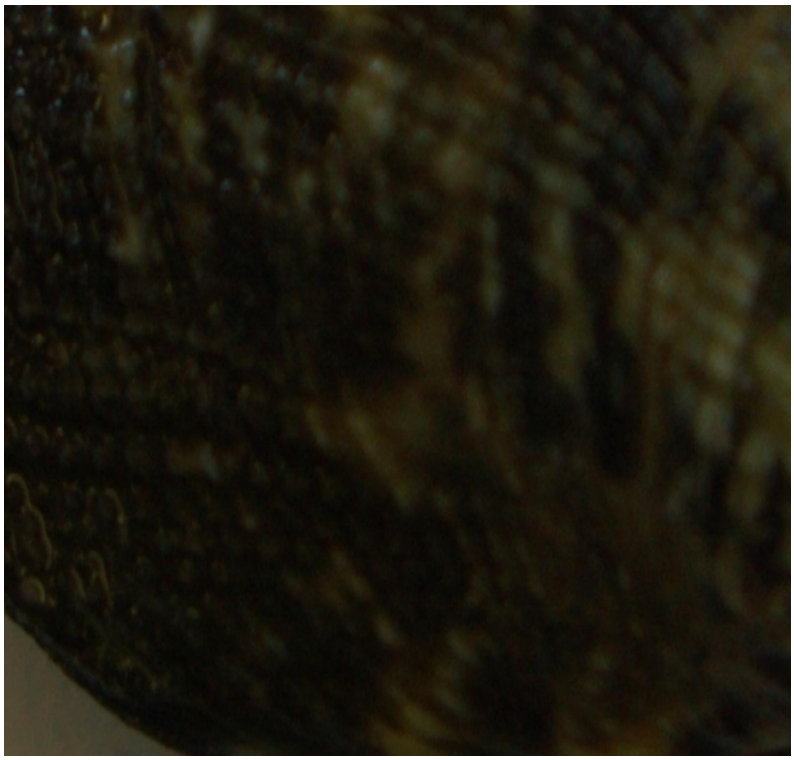


Figure 15: Example of zoom and rotation augmentation of cropped image in the ResNet dataset.

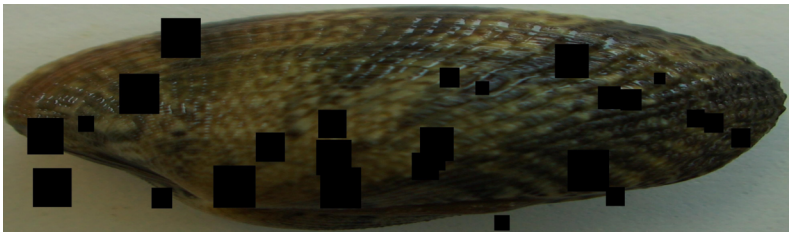


Figure 16: Example of stretching/compressing combined with flipping of cropped image in the ResNet dataset.

Figure 17: Example of blackout square augmentation of cropped image in the ResNet dataset.

4.1.2 *Browser-based face recognition application*

Requirements and evolution

The application was developed as a browser-native AI solution: a web page that performs detection, recognition and liveliness checks locally in the client (no mandatory server-side inference). The implemented authentication policy requires two factors:

(a) Overlapping clams (example 1)

(b) Organized clams (example 1)

(c) Overlapping clams (example 2)

(d) Organized clams (example 2)

(e) Overlapping clams (example 3)

(f) Organized clams (example 3)

Figure 18: Comparison of overlapping clam configurations (left column: m1–m3) versus organized, non-overlapping clam arrangements (right column: f1–f3).

- **Primary factor:** face ID or employee number entry.
- **Secondary factor:** if the primary factor is face ID, the secondary must be a liveliness (expression) check; otherwise the secondary may be an employee PIN or voice recognition.

To ensure the same person completes both factors the application retains the last confirmed face identity and continues face identification while the user completes the secondary factor, providing a short grace period before discarding that identity context. When voice authentication is selected the continuous face recognition is temporarily suspended while a short audio sample is captured; face recognition resumes if voice validation fails.

Detection, recognition and liveliness checks

The client performs all inference in the browser using Face API (TFJS formats). The implemented detection/recognition and liveliness strategy evolved as follows:

- **Detection model:** the system initially used SSD MobileNetV1 and was later switched to the lighter Tiny Face Detector to lower runtime on constrained hardware. The switch reduced computational cost with negligible impact on practical accuracy for the intended close-range use (users stand close to the tablet).
- **Recognition model:** face identification was implemented using the Face API descriptor model (`faceRecognitionNet`).
- **Landmarks:** `faceLandmark68TinyNet` was used for landmark extraction (chosen for performance).
- **Liveliness:** an initial approach based on detecting eye movement between frames proved unreliable on available landmark signals (tested with both `faceLandmark68Net` and `faceLandmark68TinyNet`). Liveliness was therefore implemented using expression detection via `faceExpressionNet`. Practical testing found three expressions to be consistently detectable: *neutral*, *happy* and *surprised*. The implemented challenge records the detected expression at identification, prompts the user to change to one of the other two expressions, then requests a final (different) expression — providing a two-step expression challenge for liveliness confirmation.
- **Voice authentication:** speaker-identification in noisy queue environments proved infeasible with available in-browser models and data; voice mode was

not integrated with a working AI model. The UI includes audio capture and a simulated failure flow, but no trained in-browser voice recogniser was deployed.

Dataset

The dataset was small and operational in nature: a handful of publicly available profile-like images plus two photographs of different people from the hosting company. No voice dataset was sourced since this functionality was set aside very early in development. The minimalist dataset for this project is the result of using pre trained models rather than training my own, meaning I needed only to test them, and between differences in facial hair, wearing glasses and camera quality, these few pictures felt sufficient.

Implementation details and runtime measures

Key implementation and runtime decisions were made to improve performance on constrained devices. Processing was restricted to a Region of Interest (ROI) in the central portion of the camera frame, which was visually indicated in the user interface. This ensured that model inputs primarily contained the user's face with less background.

In addition, input resolution was reduced and the effective processing frame rate was throttled. Expression checks were sampled at a lower rate than face identification to reduce load, with expression sampling pushed down to about 3 fps in extreme tests. To further improve runtime efficiency, lighter model variants, specifically the Tiny detector and Tiny landmarks models, were employed to reduce inference time.

Camera and flow behaviour were also adjusted. The camera remains active during PIN entry, and a short re-confirmation interval enforces that the previously identified face still corresponds to the person entering the PIN. When voice capture is active, continuous face identification is paused for the duration of audio capture to reduce resource contention.

Offline operation and logging were enabled through a service worker, which caches model artifacts, the trusted-face descriptor list, and queued attendance logs. Logs are stored locally with timestamps and retried to the server periodically, with the synchronization interval configured to one hour. Logs are cleared only upon successful server acknowledgement, ensuring reliability in environments with intermittent connectivity.

Finally, HTTPS and service-worker setup required special handling on mobile devices. During development, ngrok was initially used to create an HTTPS tunnel to serve the web page, which allowed mobile devices to grant access to their cameras, as their browsers are more strict about insecure contexts. However, this tunnel was not sufficient for service-worker registration on those devices. To resolve this, a self-signed certificate was created and manually added to the list of trusted certificates on each device. Once trusted, the browsers could successfully install the service worker when visiting the page. This ensured full offline operation with cached models and logs, independent of server connectivity once the page was initially loaded, and also prevented models from being reloaded on every visit, thereby speeding up page initialization.

Leveraging the presence of the service work, and in order to reduce the initial load time of the page and enhance scalability, an adjustment was made to the page load logic so that it did not have to perform the extraction of the facial features from the full dataset every time the page loads, which was the case in downloaded Face API sample. Instead the face Identifier (ID) page was made to load them from a file, requiring that they had already been extracted beforehand. The extraction logic was moved to a different page, designed to manage the dataset the face ID webpage uses, allowing the upload of pictures and use of the camera to take more, before extracting their features and saving them to a file, which the server would distribute to the fronted when it first loads or every hour (at the front end's request), if it had been changed more recently than the version cached in the browser.

4.2 CHALLENGES FACED

The development uncovered several recurring challenges. Each is listed with the observed impact and, where applicable, the mitigation attempted during the internship.

- **Limited compute and legacy hardware:** both training and on-device inference were constrained by available hardware. Mitigations: transfer learning, lighter model variants, ROI cropping, resolution/FPS throttling. Despite these, the older tablet still exhibited unacceptable latency for expression checks.
- **Small and imbalanced dataset (clams):** dataset size and class imbalance ("japonica" dominant; few "partida") complicated training from scratch and risked biased models. Mitigations: class-aware augmentation, class weights,

and careful validation; a custom loss was considered but not implemented within the internship timeframe.

- **Segmentation limitations and shadows (CCL):** threshold-based segmentation frequently included shadows or failed on overlapping objects, producing imperfect crops. Mitigations: morphological operations, polygonal contour refinement and alternate cropping strategies — these improved results in many cases but remained inconsistent.
- **Device-specific performance variability:** the app behaved well on modern phones and workstations but degraded on the older tablet (frame drops, 1–2 s delays). Impact: the face+expression two-factor flow was impractical on the tablet. Mitigations: multiple runtime optimisations were applied but could not fully overcome hardware limits.
- **Environmental noise for voice authentication:** the expected deployment (queues, chatter) made reliable speaker isolation difficult; combined with the lack of suitable in-browser speaker-ID models, voice mode was postponed. Mitigation: UI support and simulated failure flows were implemented so voice could be integrated later.
- **Scope changes and scheduling (Farm2Fork delay):** the original blockchain deliverable was delayed, necessitating a shift to AI projects. Impact: planned blockchain work was not completed during the internship; development effort was redirected to demonstrators that provided immediate value.
- **Privacy and deployment constraints (face system):** biometric systems raise data minimisation, consent and retention concerns. Action taken: the project design kept identifiers and logs local (service-worker cache, local queue) and flagged privacy/ethical constraints in the documentation; further certification and policy work remain outstanding.
- **Potential data leakage when mixing datasets:** mixing augmented and base datasets without explicit leakage controls could bias results. Awareness: the limitation is noted in the results and advised as an item for follow-up validation.
- **Tooling and model-availability gaps:** some desired capabilities (speaker-ID in-browser, highly-optimised custom models for legacy tablets) were outside the available toolset and time budget; these remain future work items rather than current deliverables.

4.2.1 *Browser-based face recognition application*

This subsection presents the browser-based face recognition application, detailing its evolution from a simple face-detection prototype to a fully browser-native, two-factor authentication system. The discussion covers the functional requirements, the development methodology, model selection, UI and runtime logic, and the strategies employed to operate efficiently on constrained client devices. Each part of the system is described with reference to practical implementation decisions, performance optimizations, and deployment considerations, providing a comprehensive view of the engineering trade-offs involved.

Requirements and evolution

The application was developed as a browser-native AI solution: a web page that performs detection, recognition and liveliness checks locally in the client (no mandatory server-side inference). The implemented authentication policy requires two factors:

- **Primary factor:** face ID or employee-number entry.
- **Secondary factor:** if the primary factor is face ID, the secondary must be a liveliness (expression) check; otherwise the secondary may be an employee PIN or voice recognition.

On page load the system checks camera availability. If the camera is not available the primary mode defaults to employee-number entry; if available the primary mode defaults to face ID. The page appearance during initial checks is shown in Figure 19. To ensure the same person completes both stages, the application retains the last confirmed face identity and continues face identification while the user completes the secondary factor, providing a short grace period before discarding that identity context. When voice authentication is selected the continuous face recognition is temporarily suspended while a short audio sample is captured; face recognition resumes if voice validation fails.

Detection, recognition and liveliness checks

All inference runs in the browser using Face API (TFJS formats). The detection/recognition/liveliness strategy and model choices evolved during development as follows:

- **Detection model:** initially SSD MobileNetV1 was used; this was later switched to the lighter Tiny Face Detector to lower runtime on constrained hardware with negligible practical accuracy loss for the intended close-range use.
- **Landmarks and recognition:** landmarks are extracted with `faceLandmark68TinyNet` (chosen for performance) and face identification uses the Face API descriptor model (`faceRecognitionNet`).
- **Liveliness:** eye-movement heuristics between frames proved unreliable on available landmark signals; liveliness was therefore implemented using expression detection via `faceExpressionNet`. Practical testing found three expressions to be consistently detectable: `neutral`, `happy` and `surprised`. The liveliness challenge records the detected expression at identification and then requests two distinct expressions in Stage 2 (both different from the initially recorded expression) to confirm liveliness.
- **Voice authentication:** speaker-identification in a noisy queue environment proved infeasible with available in-browser models and data; voice mode was not integrated with a working AI model. The UI contains audio capture and a simulated failure flow for future integration.

Logic and UI flow

The interaction is organised in two stages (Stage 1 — primary identification; Stage 2 — secondary confirmation) with explicit UI and runtime rules designed to keep the same user across steps.

STAGE 1 — PRIMARY IDENTIFICATION

- **Face primary mode:** the detector searches a restricted *active area* centred in the camera frame (the on-screen rectangle). The rectangle provides direct feedback: yellow when no face is detected (Figure 20), green when a recognized face is present (Figure 21), and red when an unrecognized face is detected (Figure 22). If multiple faces are present the system selects the *largest* detected face (by bounding-box area) for identification and ignores the others, which together with the restricted active area reduces the risk of IDing a bystander.
- **Employee-number primary mode:** selecting this mode disables the face model (the active-area rectangle is not displayed) and the user enters a 4-digit employee number on the on-screen numpad (Figure 23). While the page is in

Stage 1 all buttons other than the camera-mode and number-mode selectors are disabled; when a successful primary ID is obtained the application enables Stage 2 controls (the numpad becomes active and other secondary options unlock as allowed by policy).

STAGE 2 — SECONDARY IDENTIFICATION Stage 2 offers three mutually exclusive secondary checks: PIN, expression-based liveliness, or voice (simulated). The allowed secondary options depend on the chosen primary mode:

- If **face** was the primary mode, Stage 2 may be completed with a secret 4-digit PIN, an expression-based liveliness check, or voice ID. While the user completes Stage 2 the camera remains active and continues attempting to re-identify the same person from Stage 1. A grace mechanism permits up to **3 seconds** of failing to re-identify the person before the page resets to Stage 1; during this grace period the green rectangle blinks and the screen text indicates the grace state (Figure 24). When the grace period ends the page returns to Stage 1 (see Figure 20 for the related state).
- If **employee-number** was the primary mode, Stage 2 starts with the PIN option active and voice ID enabled; face-ID re-confirmation is not engaged by default in this flow, although the camera feed may still be visible. Selecting a different primary-mode button while in Stage 2 resets the interaction back to Stage 1.

Additional UI rules: pressing the voice button in Stage 2 suspends the face-recognition loop (to avoid resource contention), disables the numpad during capture, and pauses the 3-second re-identify grace timer; the current implementation animates an audio capture bar and then fails (see Figure 26), returning to the previous Stage 2 state.

Expression selection and sampling

The expression logic relies on the Face API's prediction set. The application considers the following possible expressions: `happy`, `surprised`, `neutral`, `sad`, `angry`, `disgusted`, `fearful`, and `contemptuous`.

From this set, the application uses only `neutral`, `happy`, and `surprised` for the two-step liveliness challenge because the others were not consistently triggered. Captured examples of the three chosen expressions are shown in Figure 25. Expression

checks are sampled at a reduced rate (to reduce CPU load) and, by design, the expression model is invoked only during Stage 2 when a liveliness check is selected.

Model usage, warm-up and scalability

The application coordinates several Face API models:

- **Tiny Face Detector** — face bounding boxes (used for the active-area and choosing the largest face).
- **Face Landmark Model** (`faceLandmark68TinyNet`) — 68 keypoints for alignment and post-processing.
- **Face Recognition Model** (`faceRecognitionNet`) — produces a 128-dimensional descriptor for matching.
- **Face Expression Model** (`faceExpressionNet`) — used during Stage 2 for liveliness.
- **FaceMatcher utility** — compares descriptors against known faces with a configured `similarityThreshold`.

Model artifacts are loaded on page open which produces a short initial warm-up delay of a few seconds but yields faster subsequent inference. An alternative configuration (skip warm-up to shorten first load) was considered but rejected because it slows all future inferences — an undesirable trade-off for a real-time UI.

Face matching currently iterates through the full registry of known faces and compares descriptors sequentially until a match meeting the similarity threshold is found. This linear search approach works for the small operational dataset used in development but raises scalability concerns if the registry grows significantly; indexing or approximate nearest-neighbour methods would be required for large deployments.

Implementation details and runtime measures

To improve runtime on constrained hardware the application applies several practical optimisations:

- Processing is restricted to a central ROI (the active-area rectangle) so model inputs contain mainly the user's face.

- Input resolution is reduced before inference and effective processing frame rate is throttled; expression checks are sampled at a lower rate (in extreme tests down to ∼3 fps).
- Lighter model variants (Tiny detector, Tiny landmarks) are used to reduce inference time.
- Camera behaviour is adjusted: the camera remains active during PIN entry and a short re-confirmation interval enforces that the previously identified face still corresponds to the person entering the PIN.
- Offline operation is enabled via a service worker that caches model artifacts, the trusted-face descriptor list and queued attendance logs. Logs are persisted locally with timestamps and retried to the server periodically (configured to one hour); logs are cleared only after successful server acknowledgement.
- During development ngrok was used to provide an HTTPS tunnel for camera access on mobile devices, but service-worker registration on mobile required a trusted certificate. A self-signed certificate was manually installed on test devices so the service worker could register reliably and cache models for offline use.

Figures

To help visualise the implemented workflow, the following figures illustrate key states of the browser-based face recognition application. Figures 19–26 show the page during initialization, the active-area feedback during Stage 1 and Stage 2 interactions, examples of detected expressions used in the liveliness challenge, and the simulated voice capture animation. These images provide a concrete view of how detection, recognition, liveliness checks, and UI feedback are integrated in practice.

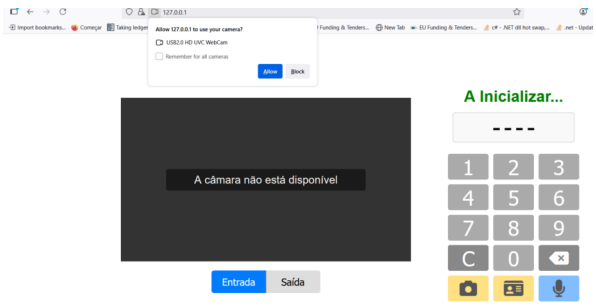


Figure 19: Initial page appearance while camera and models are being checked/loaded.

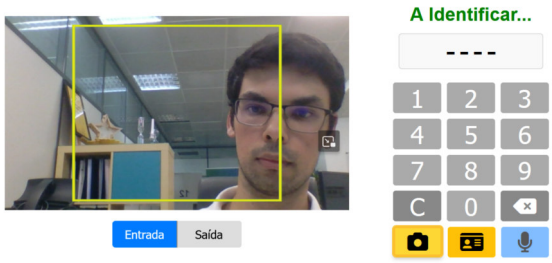


Figure 20: Active-area rectangle shown in yellow when no face is detected (Stage 1).

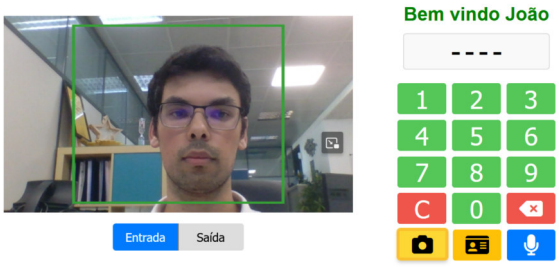


Figure 21: Active-area rectangle shown in green when a recognized face is detected (successful Stage 1 face ID).

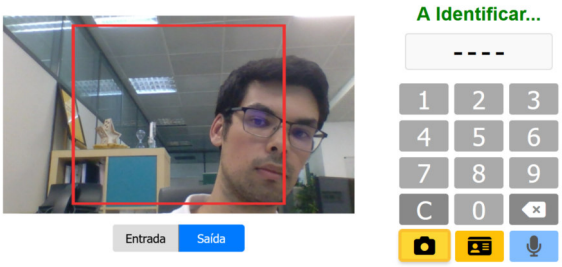


Figure 22: Active-area rectangle shown in red when an unrecognized face is detected.

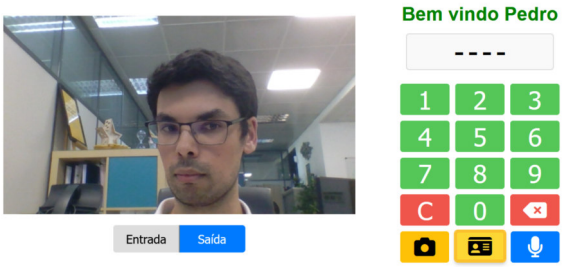


Figure 23: Employee-number primary flow: user entering the 4-digit number on the numpad.

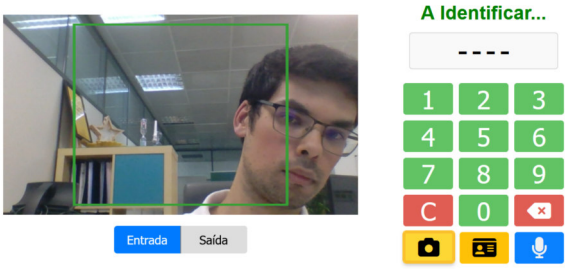
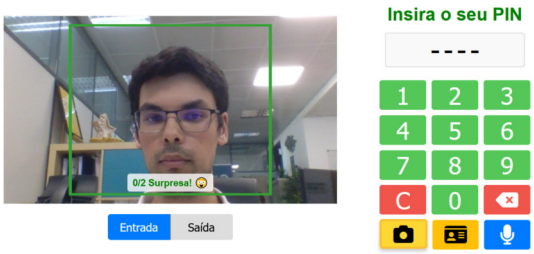
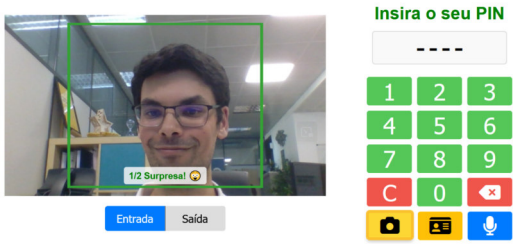


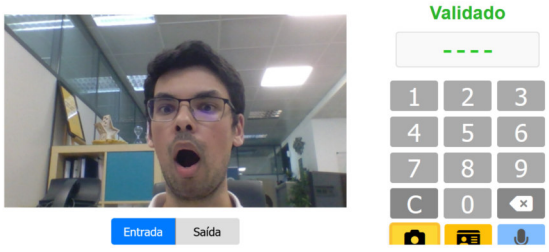
Figure 24: Stage 2 grace period example: the green rectangle blinks to indicate transient failure to re-identify the same person.



(a) Neutral (example).



(b) Happy (example).



(c) Surprised (example).

Figure 25: Example images of the three expressions used in the liveliness challenge.

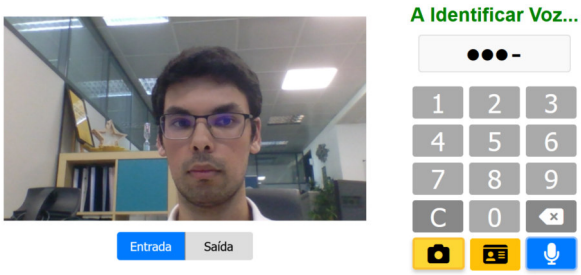


Figure 26: Simulated voice-capture animation shown while the (currently unimplemented) voice collection runs.

5

TESTING, RESULTS, AND EVALUATION

This chapter presents the testing procedures applied to the developed solution, followed by the results obtained and a critical evaluation of their significance. It aims to validate the system's performance, reliability, and compliance with the defined requirements.

5.1 EXPERIMENTAL SETUP

The specific experimental choices, dataset preparation procedures (including the connected-component labeling preprocessing), and the evaluation splits used for these tests are described in the Development chapter. This chapter contains the test results, metrics, figures and their interpretation for the evaluated models and preprocessing methods (VGG16 — final model with batch-normalization and fine-tuning, ResNet, YOLO11-nano and YOLO8-nano), along with the CCL segmentation outcomes and associated visual artifacts. The context for this chapter is the face-recognition project (browser-native, in-browser face detection/recognition and expression inference); all references to dataset items, segmentation and examples below relate to face images rather than the original clam dataset descriptions.

5.2 TESTING METHODOLOGY AND TOOLS

Tests produced the following metrics and artifacts for each experiment:

- Classification: overall accuracy, loss, validation accuracy/loss, per-class F1-scores (when available), confusion matrices, and confidence-distribution statistics (average confidence for correct vs incorrect predictions).
- Detection (YOLO): precision (P), recall (R), mean Average Precision at IoU thresholds (mAP@0.5 and mAP@0.5:0.95), per-class breakdowns, and per-image timing (preprocess / inference / postprocess).

- Ancillary runtime measures: inference latency (reported per-image where available) and proposed operational confidence thresholds.
- Visual outputs: binary segmentation masks and explicit success/failure examples (for the CCL stage), cropped/segmented comparison images, confusion matrices and confidence boxplots, and YOLO detection test reports (DetMetrics).

5.3 RESULTS AND METRICS

This section presents quantitative outcomes, key performance metrics, and visual artifacts from testing all developed models and preprocessing methods, providing a foundation for subsequent analysis and interpretation.

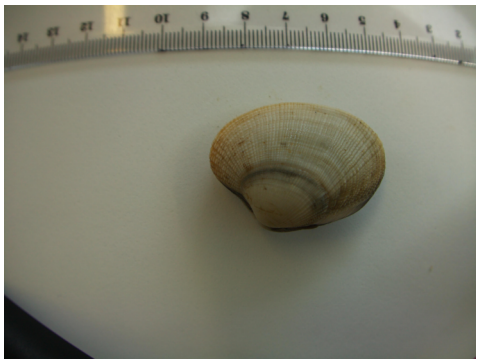
5.3.1 *Connected-Component Labeling segmentation*

The initial approach prior to implementing object segmentation was applied to the full base dataset of 749 images with the following outcomes:

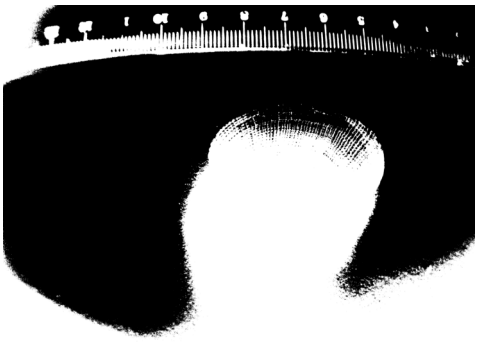
- **Successful crops:** 736 images successfully isolated the primary object, as ilustrated in Figure 28.
- **Failures:** 13 images failed, due to nearby/overlapping objects or shadow merging, as displayed in Figure 27.
- **Overall success rate:** approximately **98.3%**.

A noted limitation at this stage was that the detected bounding box often included portions of the object's shadow, because binary segmentation could not reliably differentiate shadows from the object itself. This resulted in inaccurate bounding boxes, and was a consistent issue throughout the entire dataset.

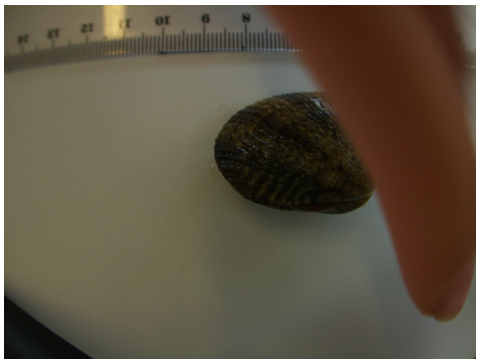
The refined method (object segmentation) made excellent improvements in a few cases (see Figures 30e and 30f), but for the most part introduced new problems whilst also not fully solving the issue with the object's shadow. The introduced issue was that parts of the main object (the clam) would be excluded, primarily affecting white parts in multi color shells, which is a majority of the dataset. Figure 29 displays what the average result looked like with a less aggressive setting for segmenting the object, while Figure 30 displays what the final version achieved.



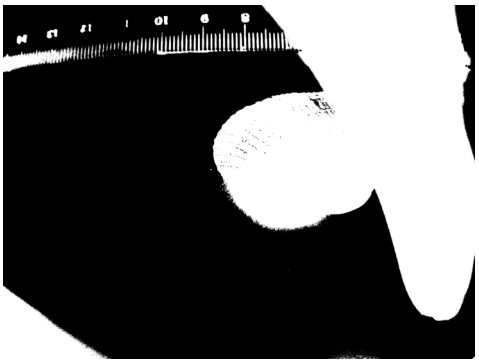
(a) Failed detection example 1 (original)



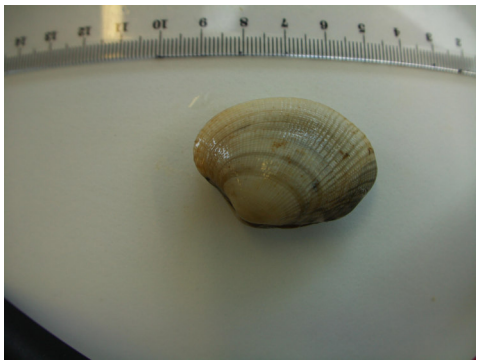
(b) Failed detection example 1 (binary mask)



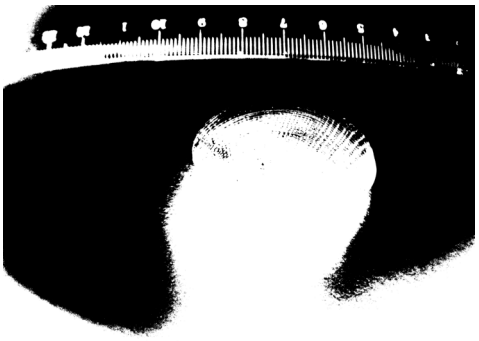
(c) Failed detection example 2 (original)



(d) Failed detection example 2 (binary mask)



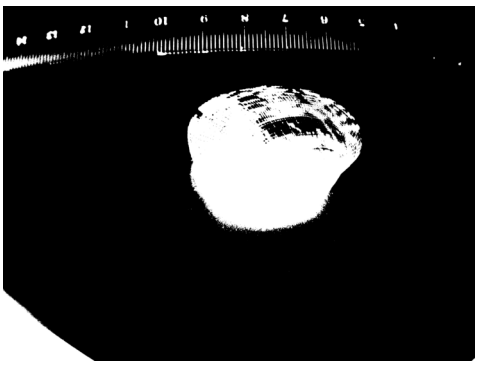
(e) Failed detection example 3 (original)



(f) Failed detection example 3 (binary mask)

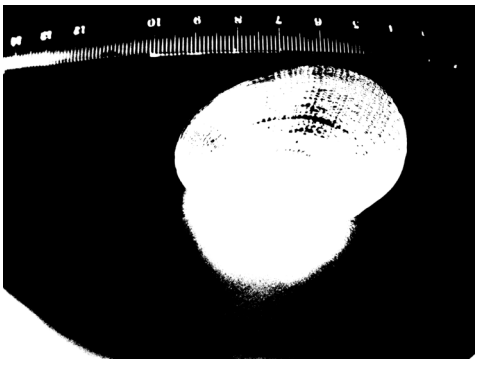
Figure 27: Examples of failed CCL-based face detection. Incorrect object merging, shadow misclassification, or boundary inaccuracies led to segmentation failures.

(a) Successful detection example 1 (cropped result)



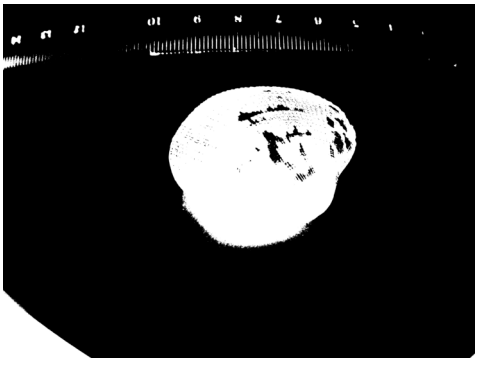
(b) Successful detection example 1 (binary mask)

(c) Successful detection example 2 (cropped result)



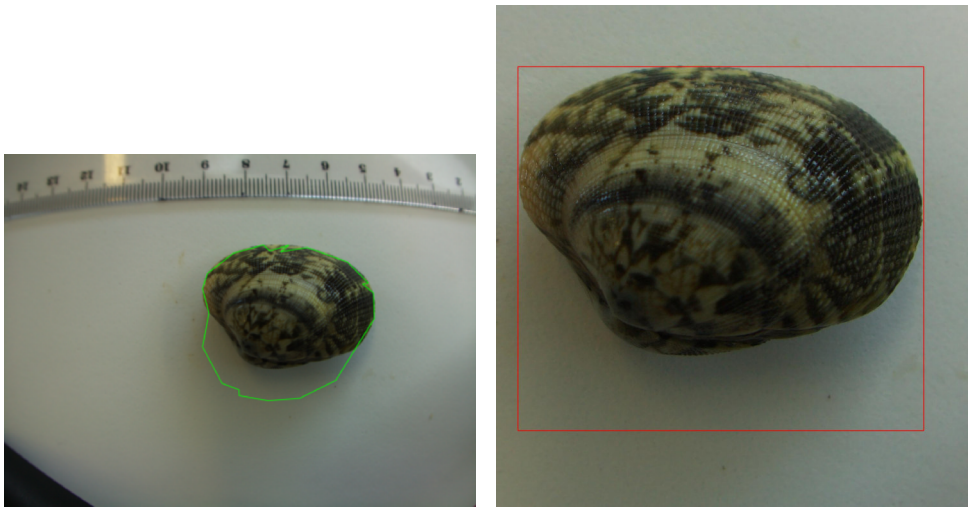
(d) Successful detection example 2 (binary mask)

(e) Successful detection example 3 (cropped result)



(f) Successful detection example 3 (binary mask)

Figure 28: Examples of successful CCL-based face detection. Binary masks accurately isolate the primary face, and cropping preserves contextual boundaries.



(a) Refined segmentation result (b) Original bounding box

Figure 29: Comparison between the original bounding box method (right) and the refined segmentation approach (left).

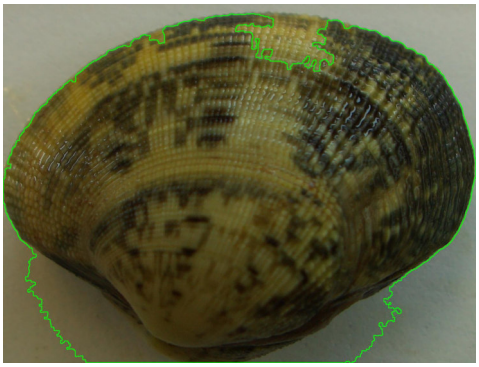
5.3.2 *Image classification — VGG16 (final) and ResNet*

This subsection reports classification results for the VGG16 and ResNet pipelines, comparing model performance, dataset effects, and the impact of fine-tuning on accuracy, loss, and per-class metrics.

Impact of fine-tuning (VGG16)

To evaluate the effect of fine-tuning on the VGG16 pipeline, its performance before and after the fine-tuning stage was compared. The fine-tuning stage produced measurable improvements in multiple metrics (the training/fine-tuning procedure details are described in the Development chapter).

- **Overall accuracy** increased from **90.35%** before fine-tuning to **93.43%** after fine-tuning.
- **Loss** decreased from **1.8446** to **1.0338**.
- **Validation accuracy** increased from **83.28%** to **89.63%**, and **validation loss** decreased accordingly.
- **Per-class F1 improvements:** Japonica 0.84 → 0.90; Macha 0.88 → 0.92; Fina 0.79 → 0.87.



(a) Refined segmentation result 1

(b) Original bounding box 1

(c) Refined segmentation result 2

(d) Original bounding box 2

(e) Refined segmentation result 3

(f) Original bounding box 3

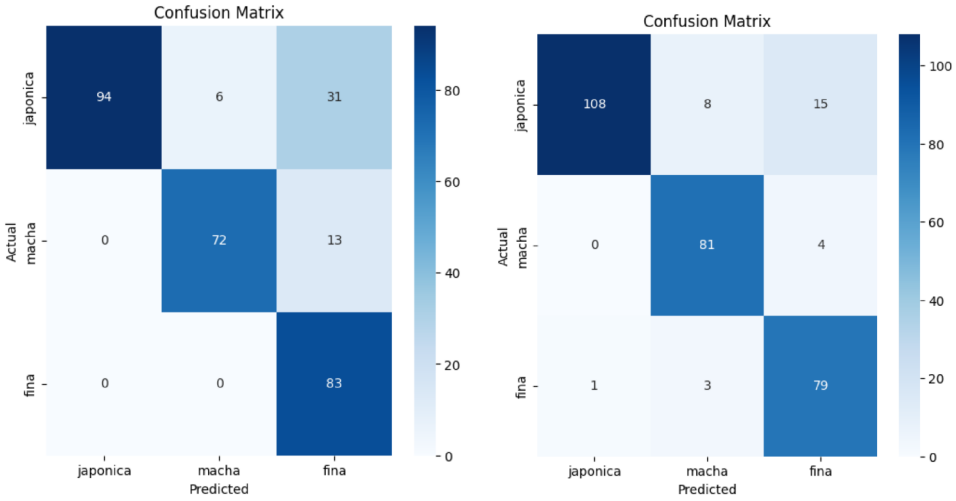
Figure 30: Comparison between the original bounding box method (right) and the refined segmentation approach with cropping (left).

These gains give clear indication that the model benefited from the additional refinement stage.

Table 6 summarises the before/after metrics and Figure 31 shows the validation confusion matrices for VGG16 pre/post fine-tuning.

Table 6: Performance Comparison Before and After Fine-Tuning (VGG16)

Metric	Before Fine-Tuning	After Fine-Tuning
Accuracy	0.9035	**0.9343**
Loss	1.8446	**1.0338**
Validation Accuracy	0.8328	**0.8963**
Validation Loss	1.8549	**1.0749**
Japonica F1-score	0.84	**0.90**
Macha F1-score	0.88	**0.92**
Fina F1-score	0.79	**0.87**



(a) Model validation pre fine tuning

(b) Model validation post fine tuning

Figure 31: Comparison of model validation before and after fine tuning (VGG16).

Effect of dataset composition on model performance

To quantify how dataset composition affects generalization, two training regimes were compared: one trained on the original base dataset and another trained on a mixed (base + augmented) dataset. Each model was tested on both base and mixed test sets. Results are summarised in Table 7 and illustrated in Figures 32 and 33.

BASE TESTING SET RESULTS

- Model trained on **base** dataset, tested on base: **95.95%** accuracy; average confidence for correct predictions = **0.74**; incorrect average confidence = **0.51**.
- Model trained on **mixed** dataset, tested on base: **98.65%** accuracy; average confidence for correct predictions = **0.89**; incorrect average confidence = **0.62**.

MIXED TESTING SET RESULTS

- Model trained on **base** dataset, tested on mixed: **62.31%** accuracy; correct avg. confidence = **0.65**; incorrect avg. confidence = **0.50**.
- Model trained on **mixed** dataset, tested on mixed: **91.62%** accuracy; correct avg. confidence = **0.82**; incorrect avg. confidence = **0.58**.

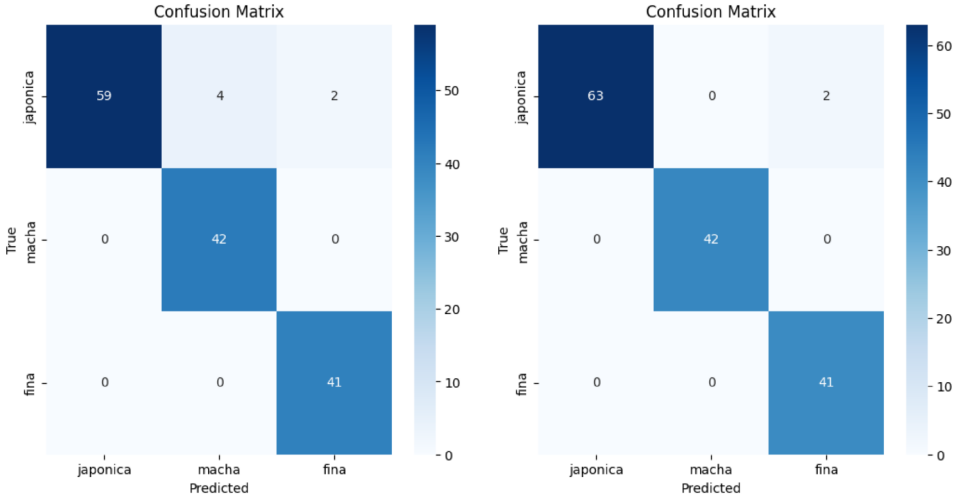
OVERALL IMPLICATIONS The mixed-trained model outperformed the base-trained model on both distributions (98.65% vs 95.95% on base; 91.62% vs 62.31% on mixed), demonstrating that dataset diversity and augmentation improved generalization in these experiments. The small volume of incorrect predictions by the mixed-trained model on the base set makes it impossible to give an estimate of what confidence level would be a good threshold for trusting its predictions. Based on the Figure 32d 70% and up might be a good level. Possible dataset-mixing leakage was acknowledged in the Development notes and should be audited in follow-up validation.

Table 7: Comparison of model performance across training and testing combinations.

Training Set	**Testing Set**	**Accuracy**	**Avg. Conf. (Correct)**	**Avg. Conf. (Incorrect)**
Base	Base	95.95%	0.74	0.51
Mixed	Base	98.65%	0.89	0.62
Base	Mixed	62.31%	0.65	0.50
Mixed	Mixed	91.62%	0.82	0.58

ResNet results and comparison with VGG16

The ResNet experiments were evaluated on two test distributions: the augmented ResNet test set and the original base test set. Per-run accuracy and prediction-confidence statistics are summarised below and illustrated with confusion matrices and confidence-distribution boxplots.



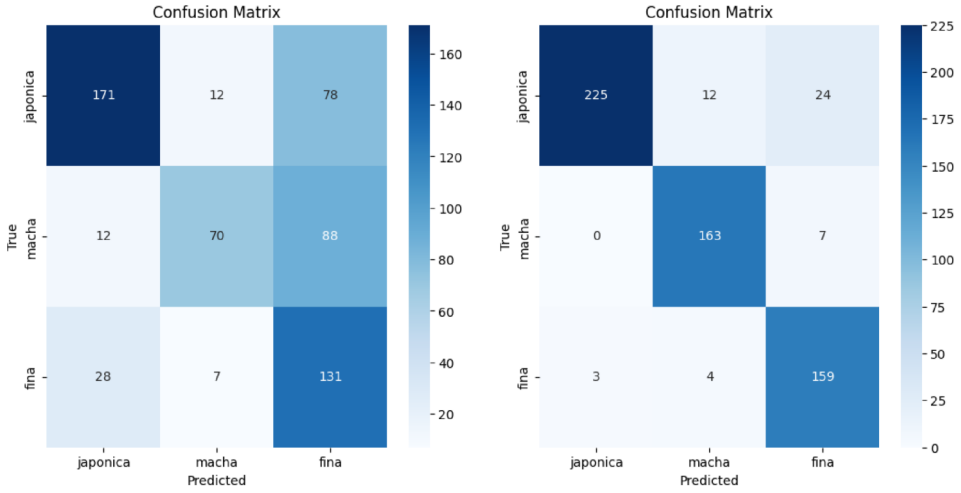
(a) Model validation (base→base)

(b) Model validation (mixed→base)

(c) Training/validation loss (base→base)

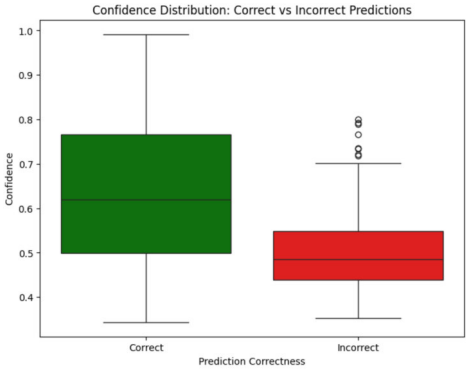
(d) Training/validation loss (mixed→base)

Figure 32: Model validation and training loss on the base dataset (VGG trained with base vs. mixed datasets).

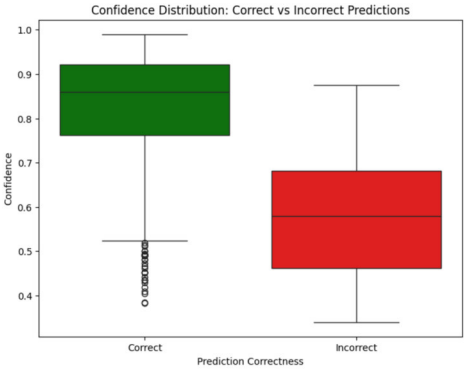


(a) Validation results (base→mixed)

(b) Validation results (mixed→mixed)



(c) Loss (base→mixed)

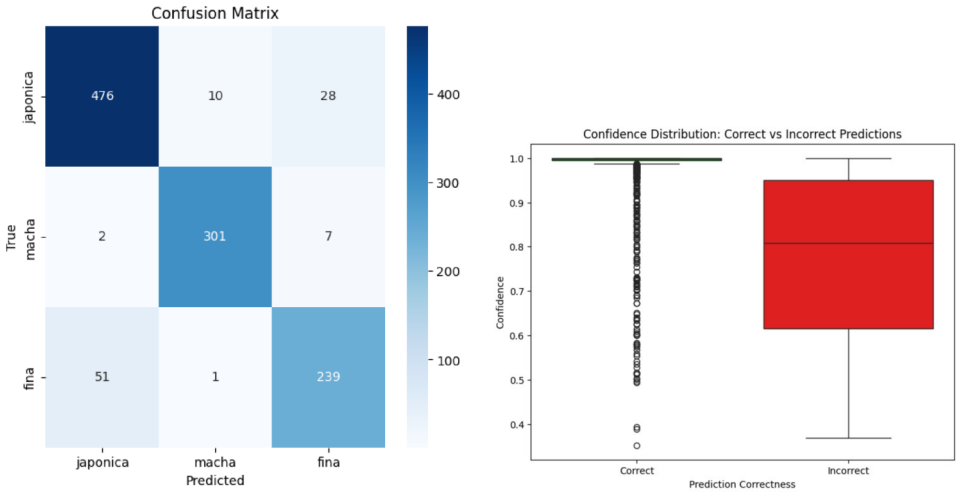


(d) Loss (mixed→mixed)

Figure 33: Model validation and training loss on the mixed dataset (VGG trained with base vs. mixed datasets).

AUGMENTED RESNET TEST SET

- Overall accuracy: **91.12%**.
- Correct predictions: Avg. confidence = **0.97**; max = 1.00; min = 0.35.
- Incorrect predictions: Avg. confidence = **0.78**; max = 1.00; min = 0.37.



(a) ResNet, augmented set confusion matrix (b) ResNet, augmented set confidence distribution

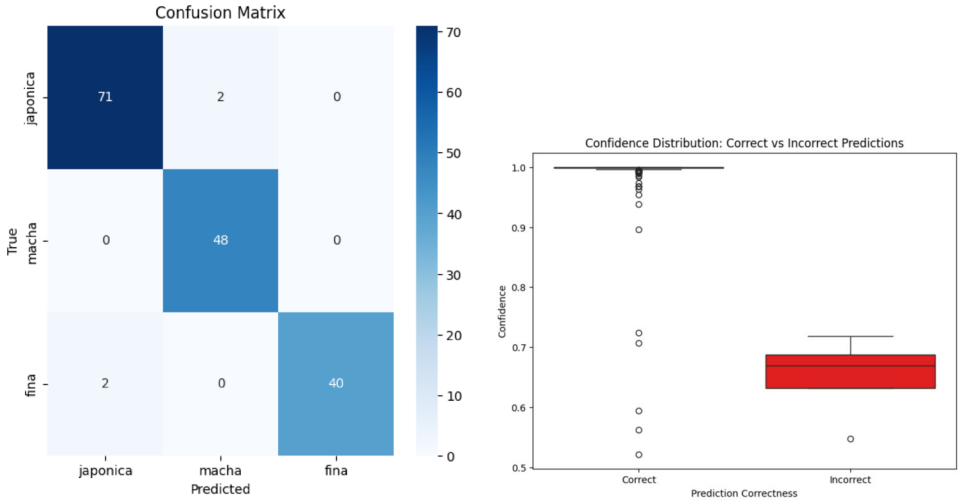
Figure 34: Model validation and training loss on its augmented dataset (ResNet trained with its augmented dataset)

BASE (ORIGINAL) TEST SET

- Overall accuracy: **97.55%**.
- Correct predictions: Avg. confidence = **0.98**; max = 1.00; min = 0.52.
- Incorrect predictions: Avg. confidence = **0.65**; max = 0.72; min = 0.55.

COMPARISON WITH THE FINAL VGG16 RESULTS

- On their respective **augmented** test sets: ResNet = **91.12%** vs VGG16 = **93.43%**.
- On the **base** test set: ResNet = **97.55%** vs VGG16 = **98.65%**.
- ResNet inference latency was typically under **0.1 s per image**, giving a practical inference-speed advantage; VGG16 (after fine-tuning) yielded slightly higher absolute accuracies in these reported comparisons, but with latencies around **1 s per image**.



(a) ResNet, base test set confusion matrix (b) ResNet, base test set confidence distribution

Figure 35: Model validation and training loss on the base (ResNet trained with its augmented dataset)

- ResNet showed very high average confidences for correct predictions (0.97–0.98); incorrect ResNet predictions had higher average confidence on the augmented set (0.78) than on the base set (0.65).

NOTES AND LIMITATIONS All comparisons use only the measured metrics available in this report. ResNet metrics reported here are overall run accuracies and confidence summaries; per-class ResNet F1-scores were not recorded for these runs. Differences in dataset composition (augmented vs. base) materially influence accuracy and confidence and should be taken into account when interpreting relative model performance.

Sumamry Comparison between ResNet and VGG16

Reported comparative points (using the available measured metrics):

- On their respective **augmented** test sets: ResNet achieved **91.12%** vs VGG16 **93.43%**.
- On the **base** (original) test set: ResNet achieved **97.55%** vs VGG16 **98.65%**.
- ResNet inference latency was typically under **0.1 s per image**, producing faster per-image runtime on average compared to the VGG16 runs typically around **1 s per image**.

- Confidence distributions: ResNet exhibited very high average confidences for correct predictions (0.97–0.98), with incorrect predictions showing higher average confidence on the more challenging augmented set (0.78) than on the base set (0.65).

YOLO models comparison (YOLO11-nano vs YOLO8-nano)

The YOLO detector evaluations produced per-class detection metrics and per-image timing. The comparison results are summarised in Table 8.

Table 8: Comparison of YOLO11-nano and YOLO8-nano test results (classes: `normal` / `partida`).

Aspect	**YOLO11-nano (Ultralytics)**	**YOLO8-nano (Ultralytics)**
test set (images / instances)	104 images, 170 instances	125 images, 201 instances
Aggregate metrics (Precision (P) / Recall (R) / mean Average Precision at Intersection-over-Union (IoU) threshold 0.5 (mAP@0.5) / mean Average Precision averaged across IoU thresholds from 0.5 to 0.95 in steps of 0.05 (mAP@0.5:0.95))	0.802 / 0.861 / 0.906 / 0.888	**0.947 / 0.933 / 0.971 / 0.936**
Per-class — `normal` (images / instances)	92 images, 156 instances; P = 0.960; R = 0.936; mAP@0.5 = 0.979; mAP@0.5:0.95 = 0.943	96 images, 177 instances; P = 0.944; R = 0.950; mAP@0.5 = 0.987; mAP@0.5:0.95 = 0.955
Per-class — `partida` (images / instances)	13 images, 14 instances; P = 0.643; R = 0.786; mAP@0.5 = 0.833; mAP@0.5:0.95 = 0.833	23 images, 24 instances; P = 0.950; R = 0.917; mAP@0.5 = 0.954; mAP@0.5:0.95 = 0.916
Per-image timing / speed (Central Processing Unit (CPU))	2.2 ms preprocess, **70.8 ms inference**, 0.5 ms postprocess per image	1.8 ms preprocess, **124.1 ms inference**, 0.4 ms postprocess per image
Results saved (run folder)	results saved to `runs/detect/val2`	results saved to `runs/detect/train103`
Decision impact	Faster inference on CPU and apparent training headroom led to prioritising YOLO11-nano for subsequent work (despite lower initial mean Average Precision (mAP)).	Although YOLO8-nano showed superior aggregate test metrics, its longer inference time made it less suitable given the latency constraint (target ∼0.1 s).

Final YOLO model tests

The final detector evaluation compared three TFLite builds of the same YOLO model family: (1) **non fine-tuned (32-bit float)**, (2) **fine-tuned (32-bit float)**, and (3) **fine-tuned (16-bit float)**. Table 9 summarises the measured per-run statistics.

Table 9: Comparison of final YOLO TFLite builds (reported values taken from provided evaluation output).

Variant	Images	GT	Pred	True Positive (TP)	False Positive (FP)	False Negative (FN)	Precision	Recall	Inf. mean (ms)	Process
Non fine-tuned (32-bit)	113	113	113	107	6	6	0.947	0.947	176.5	
Fine-tuned (32-bit)	113	113	113	107	6	6	0.947	0.947	174.4	
Fine-tuned (16-bit)	113	113	113	107	6	6	0.947	0.947	167.6	

PER-CLASS BEHAVIOUR The per-class breakdowns reported for each variant are:

- **Non fine-tuned (32-bit)**:
  - `Partida`: Ground Truth (GT) 21 / Pred 24 / TP 21 / FP 3 / FN 0 → P = 0.875, R = 1.000
  - `Japonica`: GT 45 / Pred 40 / TP 40 / FP 0 / FN 5 → P = 1.000, R = 0.889
  - `Macha`: GT 31 / Pred 31 / TP 31 / FP 0 / FN 0 → P = 1.000, R = 1.000
  - `Fina`: GT 16 / Pred 18 / TP 15 / FP 3 / FN 1 → P = 0.833, R = 0.938
- **Fine-tuned (32-bit)**:
  - `Partida`: GT 21 / Pred 22 / TP 20 / FP 2 / FN 1 → P = 0.909, R = 0.952
  - `Japonica`: GT 45 / Pred 43 / TP 43 / FP 0 / FN 2 → P = 1.000, R = 0.956
  - `Macha`: GT 31 / Pred 32 / TP 30 / FP 2 / FN 1 → P = 0.938, R = 0.968
  - `Fina`: GT 16 / Pred 16 / TP 14 / FP 2 / FN 2 → P = 0.875, R = 0.875
- **Fine-tuned (16-bit)**: (per-class numbers reported identical to the 32-bit fine-tuned run)
  - `Partida`: P = 0.909, R = 0.952
  - `Japonica`: P = 1.000, R = 0.956
  - `Macha`: P = 0.938, R = 0.968
  - `Fina`: P = 0.875, R = 0.875

OVERALL DETECTION STATISTICS. All three builds produced identical aggregate detection counts in the provided evaluations: 113 images, 113 GT instances, 113 predictions, 107 true positives, 6 false positives and 6 false negatives — yielding the same overall precision and recall of **0.947** for each reported build (see Table 9).

TIMING BEHAVIOUR AND FLOAT16 EFFECT. Two timing summaries were provided:

- *TFLite inference times (excluding warmup)*:

  - Non fine-tuned (32-bit): mean = 176.5 ms, median = 154.5 ms, min = 124.4 ms, max = 1762.9 ms (outlier).
  - Fine-tuned (32-bit): mean = 174.4 ms, median = 169.3 ms, min = 133.4 ms, max = 266.8 ms.
  - Fine-tuned (16-bit): mean = 167.6 ms, median = 164.7 ms, min = 131.2 ms, max = 242.9 ms.

- *Per-image processed times (script output, excluding first/warmup image)*:
  - Non fine-tuned (32-bit): mean = 155.6 ms, median = 156.3 ms, min = 109.4 ms, max = 203.1 ms.
  - Fine-tuned (32-bit): mean = 162.5 ms, median = 156.3 ms, min = 125.0 ms, max = 218.8 ms.
  - Fine-tuned (16-bit): mean = 142.9 ms, median = 140.6 ms, min = 109.4 ms, max = 187.5 ms.

Because the aggregate TP/FP/FN counts did not change across the three builds, the principal observable effects from the provided outputs are:

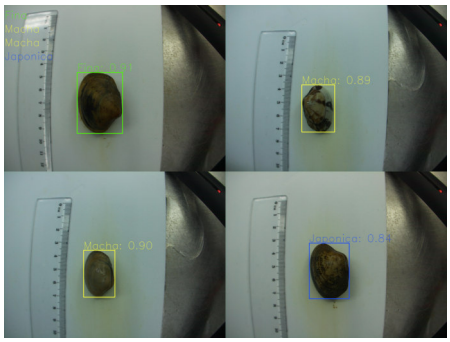
- **Per-class tradeoffs:** fine-tuning redistributed errors between classes (e.g., `Partida` precision improved while recall decreased slightly; `Japonica` false negatives were reduced).
- **Latency:** the 16-bit fine-tuned build shows the lowest processed-image mean/median times in the provided script outputs (mean 142.9 ms, median 140.6 ms) and a reduced worst-case in the TFLite inference summary compared with the extreme outlier present in the non fine-tuned inference log.

The figure 36 displays a few outputs from the 16-bit model, with the ground truth displayed on the top left of each image, in order of quadrant in the case of the figure with multiple clams (top left first, then top right, then bottom left and lastly bottom right).

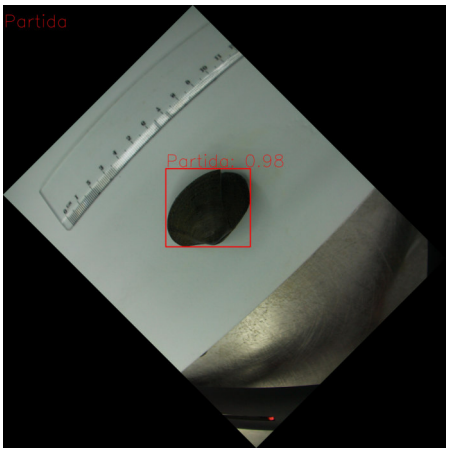
5.3.3 *Discussion and Interpretation*

This section summarizes the main interpretations drawn directly from the test results reported above.

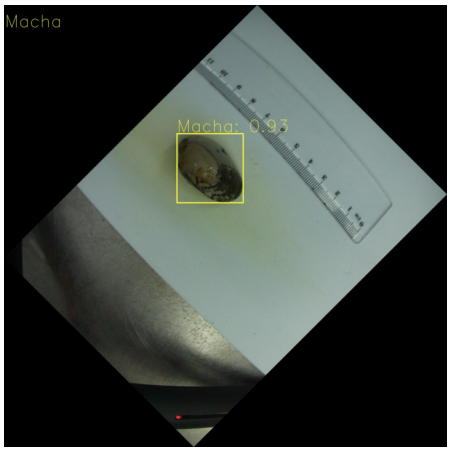
- **CCL segmentation:** threshold-based CCL proved highly effective as a lightweight preprocessing step (98.3% success isolating the primary face)



(a) Four clams.



(b) One rotated clam.



(c) One rotated clam.

(d) Original dataset image.

Figure 36: Sample output from the 16-bit model.

but suffered when faces had pronounced texture, multiple colors or shadows merged with object contours. These residual segmentation errors affected a small fraction of the dataset (13/749) but are relevant for downstream training fidelity.

- **Fine-tuning benefits (VGG16):** the controlled fine-tuning phase produced measurable improvements across accuracy, loss, validation metrics and per-class F1-scores (Table 6), supporting the use of a fine-tune stage following transfer learning convergence. A caution is noted that the timing of fine-tuning might have been conservative and could slightly overstate necessity versus extended base-stage training.
- **Data diversity improves generalization:** models trained on the mixed/augmented dataset generalized better to both base and mixed test sets, often outperforming models trained only on the base set even when evaluated on the base distribution (see Table 7). Possible dataset mixing/leakage is noted and should be audited in follow-up work.
- **ResNet vs VGG16 trade-offs:** ResNet yielded faster inference with strong confidence separation on the base test set; VGG16 (after fine-tuning) achieved slightly higher absolute accuracies in the reported comparisons. Confidence-distribution behaviour provides an operational threshold (0.80) for human-in-the-loop checks.
- **YOLO detector choice driven by latency:** YOLO8-nano produced higher mAP but higher inference latency; YOLO11-nano offered substantially lower CPU inference time with room for further training — this influenced the decision to prioritise YOLO11-nano for follow-up work. The accuracy in classification was astounding given the extreme imbalance in the dataset, this result was very much unexpected, and displays a very good ability to abstract by YOLO, in both models tested.
- **Final YOLO optimizations:** Fine-tuning redistributed errors across classes without altering aggregate precision/recall (remaining at 0.947), while quantization to 16-bit floats reduced mean inference latency (from 176 ms to 168 ms) and processed times (to 143 ms mean), improving runtime feasibility for on-device deployment and mitigating outliers in worst-case performance.

5.3.4 *Face ID*

This subsection reports the device-level evaluation and conclusions for the browser-native face recognition application described in the Development chapter. It reproduces the measured device behaviour, runtime observations and the conclusions drawn from those tests.

Evaluation and observed behaviour

The implementation was validated on the development workstation, a modern smartphone and tablet similar to the target device. Observations from testing the webpage on these devices were as follows:

- **Workstation performance:** the workstation ran the application with no noticeable frame drops or delays.
- **Smartphone performance:** the smartphone had some frame drops prior to optimizations to the frequency of expression detection, but showed no performance issues with the final settings.
- **Tablet performance:** the tablet device exhibited significant frame drops and delays of 1–2 s during expression checks. This latency made the face+expression two-factor flow on that tablet impractical, though not completely unusable.

All three devices were successfully able to load the page when the server was down, indicating correct functioning of the service worker.

Conclusions

The development effort produced a working browser-native multi-factor attendance application that meets the offline, in-browser requirement on modern devices but is constrained by the legacy tablet hardware. Principal conclusions:

- In-browser detection, recognition and expression inference is feasible on modern smartphones and workstations using Face API models.
- The full face+expression two-factor flow is not practical on the tested older tablet due to persistent latency and frame drops, despite multiple optimizations and lighter model variants.

- Expression-based liveliness checks provide a practical anti-spoofing measure where the device can process frames at acceptable rates; eye-movement heuristics were not effective with available landmark signals.
- Voice authentication was not integrated due to model availability and environmental constraints; the UI contains capture and failure-handling flows but no in-browser speaker-ID model.
- Service-worker caching successfully decouples operation from the server, enabling offline logging and later synchronization, as well as reducing initial load times and allows loading the page with no access to the server all together.

6

CONCLUSIONS AND FUTURE WORK

This chapter draws together the practical outcomes of the project and points to sensible next steps. It summarises the main contributions and empirical findings from the two delivered systems — the clam classification/detection pipeline and the browser-based face recognition application — emphasising what worked, what did not, and why.

Following the summary, the chapter outlines concrete proposals for future improvements, organised by project: recommendations for data collection, model refinement and alternative methods for the clam pipeline; and suggestions for stronger liveness checks, model compression and feature completion (voice ID, scalability and privacy) for the face recogniser. The aim is to give a compact appraisal of delivered value and a prioritized roadmap that can be followed to increase accuracy, robustness and deployability.

6.1 SUMMARY OF CONTRIBUTIONS

This section highlights the key achievements of the project, including improved classification accuracy through data augmentation, effective object detection with YOLO on resource-constrained devices, and the feasibility of a browser-native multi-factor face recognition system.

6.1.1 *Clams Classifier*

The use of CCL to perform object detection (in actuality segmentation) proved impossible due to the presence of objects' shadows. While this ruled out the use of this technique for this particular scenario, in a more controlled environment which can minimize shadows it could still be a viable technique, which would eliminate the need for a dedicated object detector (or segmentor) AI model. As such, the technique remains relevant to the topic of computer vision solutions for resource constrained devices.

The achieved performance improvement on the VGG based model between using just the original dataset, and the mixed dataset (which includes the original and the augmented one) highlighted the important role data augmentation plays in improving a model's capacity for abstraction, as the model trained with the mixed set had better performance on the original dataset than the model trained on the original dataset itself.

Furthermore, YOLO's capacity to both detect and classify objects in real time with high accuracies and low resource usage makes it a phenomenal solution to the issue of balancing accuracy with performance for edge devices. Since new YOLO models are constantly in development, introducing both optimizations to the existing architecture and new cutting edge techniques, the expectation is that they continue to improve significantly. This leads to the conclusion that YOLO will be among the leading solutions for AI based systems in resource constrained devices that require any form of object detection. Lastly, the ability displayed YOLO displayed to generalize even with an extremely unbalanced dataset further cements the belief that it will continue to be among the top choices for years to come.

6.1.2 *Facial Recognition*

While the results achieved do display that the envisioned system can be built and work with modern devices, the reality of many companies is that hardware is not up to date, especially edge devices with minor (resource wise) tasks, such as check in/out. For this reason, it may still take some years before such solutions can be widely adopted. That being said, TFJS paired with a service worker did allow for a very unique system, creating a potential for unlimited clients entirely independent of the server, all running the model at the same time, which although a bit redundant for most devices, since they could simply install an application to run a model locally, there are bound to be more niche use cases (besides the one explored in this document), that can make good use of it.

6.2 PROPOSALS FOR FUTURE IMPROVEMENTS

This section outlines potential enhancements, such as expanding datasets, refining model architectures, implementing dual-head predictions for clams, optimizing model performance for edge devices, and exploring improved liveness checks and voice recognition for facial authentication.

6.2.1 *Clams Classifier*

As previously stated, under more favorable conditions, CCL should be a viable alternative to the use of AI for object detection (or segmentation), however, it remains to be studied how strict these conditions need to be. Secondly, since the dataset was imbalanced and relatively small, acquiring a more complete one should allow for the training of a model with better performance, as well as for higher confidence when measuring its performance. This would be particularly impactful in the classification of broken shells, as the images available for this class are an extreme minority (less than 3%).
Furthermore, since YOLO12 ([49]) was not released before the last advancement of this project, it has not been tested with the dataset, having the potential to provide a better performance than the tested versions, though this is unlikely to be the case seeing how it is geared towards higher end devices, and thus some of its key functionalities are not available for this particular use case. However, the study [20] did find that this version introduced significant overhead without significant performance gains. As the model is to be used on an edge device testing YOLO12 is not a priority.
Last but not least, the model could be adapted to have two heads, one for predicting clam species and the other for prediction shell conformity. This would permit the development of a more sufisticated system of clam separation, organizing the ones discarded for shell non-conformity, so that a human can simply confirm whether they really are broken, not having to then also identify the species. This should also improve the model's performance in both tasks, as it would not confuse the model with classifying an image as not being of its species when it is, due to it being broken. This change might prove challenging when relying on ultralytic's software if this does not have built in support for this type of adjustment, as several components will then need to be altered (dataset, loss function, model architecture, etc).

6.2.2 *Facial Recognition*

Firstly, more time should be invested into figuring out a way to have a functional liveness check with the face descriptors, as this would provide a way to avoid video/image spoofing without the need to involve the facial expression recognition model. Secondly, since the Models were too heavy for the targeted device, it would

be worth looking into ways to further reduce their load or ways to optimize the web page or the device's performance.

Additionally, with the Face API library being entirely written in TFJS, it should not be too challenging to develop new models and use TFJS to utilize them in place of the existing ones. Furthermore, since the release of Face API there have been numerous advancements in this field (namely with the YOLO models) that should allow for the creation of lighter or more accurate models.

Finally, the voice recognition feature remains unimplemented, which should be feasible considering the work done in [44].

BIBLIOGRAPHY

[1] Ammar Ahmed and Abdul Manaf. «Pediatric Wrist Fracture Detection in X-rays via YOLOv10 Algorithm and Dual Label Assignment System». In: (2024). arXiv: 2407.15689. URL: http://arxiv.org/abs/2407.15689.

[2] Hatem Belhassen, Virginie Fresse, and El Bay Bourennane. «Comparative Study of Face and Person Detection algorithms: Case Study of tramway in Lyon». In: *Proceedings of International Conference on Advanced Systems and Emergent Technologies, IC_ASET 2019* August (2019), pp. 154–159. DOI: 10.1109/ASET.2019.8871003.

[3] Bitcliq Technologies, S.A. *BIG EYE Smart Fishing — Seafood Traceability*. Accessed: 2025-09-12. 2025. URL: https://www.bitcliq.com/bigeye/.

[4] Bitcliq Technologies, S.A. *Bitcliq Communications Kit (product descriptions, Big Eye Smart Farming)*. Accessed: 2025-09-12. 2021. URL: https://www.bitcliq.com/wp-content/uploads/2021/11/Bitcliq-Kit-Comunicacao-ENG-03.pdf.

[5] Bitcliq Technologies, S.A. *Bitcliq Technologies — Home / About*. Accessed: 2025-09-12. 2025. URL: https://www.bitcliq.com/en/home-eng/.

[6] Phakpoom Chinprutthiwong et al. «Security Study of Service Worker Cross-Site Scripting.» In: *ACM International Conference Proceeding Series* 3427290 (2020), pp. 643–654. DOI: 10.1145/3427228.3427290.

[7] Phakpoom Chinprutthiwong et al. «The service worker hiding in your browser: The next web attack target?» In: *ACM International Conference Proceeding Series* (2021), pp. 312–323. DOI: 10.1145/3471621.3471845.

[8] Damien Dablain et al. «Understanding CNN fragility when learning with imbalanced data». In: *Machine Learning* 113.7 (2024), pp. 4785–4810. ISSN: 15730565. DOI: 10.1007/s10994-023-06326-9. arXiv: 2210.09465.

[9] *Explore Ultralytics YOLOv8 - Ultralytics YOLO Docs*. URL: https://docs.ultralytics.com/models/yolov8/ (visited on 04/08/2025).

[10] *face-api.js*. URL: https://justadudewhohacks.github.io/face-api.js/docs/index.html (visited on 04/08/2025).

[11] Embalagem do Futuro and Bitcliq Technologies. *PPS14 "Farm2Fork Traceability" — project description*. Accessed: 2025-09-12. 2024. URL: https://embalagemdofuturo.pt/en/tag/pps14-en/.

[12] *GitHub - ckhroulev/connected-components: An implementation of a standard 2-scan connected component labeling algorithm using run-length encoding*. URL: https://github.com/ckhroulev/connected-components (visited on 06/24/2025).

[13] Souhail Guennouni, Ali Ahaitouf, and Anass Mansouri. «A comparative study of multiple object detection using haar-like feature selection and local binary patterns in several platforms». In: *Modelling and Simulation in Engineering* 2015 (2015). ISSN: 16875605. DOI: 10.1155/2015/948960.

[14] Kaiming He et al. «Deep Residual Learning for Image Recognition». In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 770–778.

[15] Lifeng He et al. «The connected-component labeling problem: A review of state-of-the-art algorithms». In: *Pattern Recognition* 70 (2017), pp. 25–43. ISSN: 00313203. DOI: 10.1016/j.patcog.2017.04.018.

[16] *How Data Augmentation Impacts Performance Of Image Classification*. URL: https://analyticsindiamag.com/deep-tech/image-data-augmentation-impacts-performance-of-image-classification-with-codes/ (visited on 06/24/2025).

[17] Mohammed Hussain et al. «Optimizing VGG16 deep learning model with enhanced hunger games search for logo classification». In: *Scientific Reports* 14.1 (2024), pp. 1–34. ISSN: 20452322. DOI: 10.1038/s41598-024-82022-5.

[18] «Inception ResNet v2 for Early Detection of Breast Cancer in Ultrasound Images». In: *Journal of Information System Exploration and Research* 2.2 (2024), pp. 93–102. ISSN: 2964-1160. DOI: 10.52465/joiser.v2i2.439.

[19] Sergey Ioffe and Christian Szegedy. «Batch Normalization : Accelerating Deep Network Training by Reducing Internal Covariate Shift». In: (). arXiv: arXiv:1502.03167v3.

[20] Nidhal Jegham et al. «Evaluating the Evolution of YOLO (You Only Look Once) Models: A Comprehensive Benchmark Study of YOLO11 and Its Predecessors». In: (2024). arXiv: 2411.00201. URL: http://arxiv.org/abs/2411.00201.

[21] Moran Ju et al. «The Application of Improved YOLO V3 in Multi-Scale Target Detection». In: *Applied Sciences 2019, Vol. 9, Page 3775* 9 (18 Sept. 2019), p. 3775. ISSN: 2076-3417. DOI: 10.3390/APP9183775. URL: https://www.mdpi.com/2076-3417/9/18/3775/htm%20https://www.mdpi.com/2076-3417/9/18/3775.

[22] Reshma Kamath. «Food Traceability on Blockchain: Walmart's Pork and Mango Pilots with IBM». In: *Journal of the British Blockchain Association* 1.1 (2018). Using data from 2017 pilots; Accessed: 2025-09-12, pp. 1–12. URL: https://www.researchgate.net/publication/326188675_Food_Traceability_on_Blockchain_Walmart%27s_Pork_and_Mango_Pilots_with_IBM.

[23] Soroush Karami, Panagiotis Ilia, and Jason Polakis. «Awakening the Web's Sleeper Agents: Misusing Service Workers for Privacy Leakage». In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021* February (2021). DOI: 10.14722/ndss.2021.23104.

[24] Kostiantyn Khabarlak. «Face Detection on Mobile: Five Implementations and Analysis». In: (2022). arXiv: 2205.05572. URL: http://arxiv.org/abs/2205.05572.

[25] Pengfei Li et al. «Making AI Less "Thirsty": Uncovering and Addressing the Secret Water Footprint of AI Models». In: (2025), pp. 1–10. arXiv: 2304.03271. URL: http://arxiv.org/abs/2304.03271.

[26] *LiteRT overview | Google AI Edge | Google AI for Developers*. URL: https://ai.google.dev/edge/litert (visited on 06/24/2025).

[27] Mohammed Aqib Zeeshan Md Sayem Iftekar. «The 'face-api.js' Library for Accurate Face Recognition in Web- Applications and Possible use Cases with Accuracy Metrics». In: *International Journal of Computer Applications* 186.21 (May 2024), pp. 10–15. ISSN: 0975-8887. DOI: 10.5120/ijca2024923507. URL: https://ijcaonline.org/archives/volume186/number21/the-face-apijs-library-for-accurate-face-recognition-in-web-applications-and-possible-use-cases-with-accuracy-metrics/.

[28] *Model conversion | TensorFlow.js*. URL: https://www.tensorflow.org/js/guide/conversion.

[29] Mozilla. *Common Voice*. Accessed: 2025-08-28. 2025. URL: https://commonvoice.mozilla.org/.

[30] Ismail Olaniyi Muraina. «IDEAL DATASET SPLITTING RATIOS IN MACHINE LEARNING ALGORITHMS :» in: February (2022).

[31] Behnam Neyshabur, Ryota Tomioka, and Nathan Srebro. «Norm-Based Capacity Control in Neural Networks». In: *Proceedings of the 28th International Conference on Learning Representations (ICLR) / MLR* (2015).

[32] Andrew Y. Ng. «Feature Selection, L1 vs. L2 Regularization, and Rotational Invariance». In: *Proceedings of the 21st International Conference on Machine Learning (ICML)* (2004).

[33] *peterkim (Sangjoon Kim) / - velog*. URL: https://velog.io/@peterkim/posts (visited on 06/24/2025).

[34] SGS Portugal and Embalagem do Futuro. *PPS14 - Traceability Farm to Fork — datasheet*. Accessed: 2025-09-12. 2022. URL: https://www.sgs.com/-/media/sgscorp/documents/corporate/technical-documents/technical-datasheets/ficha-de-projeto-embalagem-do-futuro.cdn.pt-pt.1.pdf.

[35] Joseph Redmon et al. «You Only Look Once: Unified, Real-Time Object Detection». In: (2015). arXiv: 1506.02640v5. URL: http://pjreddie.com/yolo/.

[36] Google Research. *Google Colaboratory*. Accessed: 2025-09-20. 2017. URL: https://colab.research.google.com/.

[37] Jessica M. Rudd and Herman "Gene" Ray. «An Empirical Study of Downstream Analysis Effects of Model Pre-Processing Choices». In: *Open Journal of Statistics* 10.05 (2020), pp. 735–809. ISSN: 2161-718X. DOI: 10.4236/ojs.2020.105046.

[38] Ahmet Oğuz Saltık, Alicia Allmendinger, and Anthony Stein. «Comparative Analysis of YOLOv9, YOLOv10 and RT-DETR for Real-Time Weed Detection». In: (2024). arXiv: 2412.13490. URL: http://arxiv.org/abs/2412.13490.

[39] Ranjan Sapkota et al. «Comprehensive Performance Evaluation of YOLOv10, YOLOv9 and YOLOv8 on Detecting and Counting Fruitlet in Complex Orchard Environments». In: 2016 (2024). arXiv: arXivpreprintarXiv:2407.12040.

[40] Aayush Saxena et al. «Comprehensive Study on Performance Evaluation and Optimization of Model Compression: Bridging Traditional Deep Learning and Large Language Models». In: (2024), pp. 1–11. arXiv: 2407.15904. URL: http://arxiv.org/abs/2407.15904.

[41] Shreve. *ngrok*. Accessed: 2025-09-11. 2025. URL: https://ngrok.com/.

[42] Karen Simonyan and Andrew Zisserman. «Very Deep Convolutional Networks for Large-Scale Image Recognition». In: *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings* (Sept. 2014). URL: https://arxiv.org/abs/1409.1556v6.

[43] Karen Simonyan and Andrew Zisserman. «Very deep convolutional networks for large-scale image recognition». In: *International Conference on Learning Representations (ICLR)*. 2015.

[44] Andrew J.R. Simpson, Gerard Roma, and Mark D. Plumbley. «Deep karaoke: Extracting vocals from musical mixtures using a convolutional deep neural network». In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9237 (2015), pp. 429–436. ISSN: 16113349. DOI: 10.1007/978-3-319-22482-4_50. arXiv: 1504.04658.

[45] Nitish Srivastava et al. «Dropout: A simple way to prevent neural networks from overfitting». In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. ISSN: 15337928.

[46] Mingxing Tan and Quoc V. Le. «EfficientNet: Rethinking model scaling for convolutional neural networks». In: *36th International Conference on Machine Learning, ICML 2019* 2019-June (2019), pp. 10691–10700. arXiv: 1905.11946.

[47] Walmart Global Tech. *Blockchain in the food supply chain*. Accessed: 2025-09-12. 2017. URL: https://tech.walmart.com/content/walmart-global-tech/en_us/blog/post/blockchain-in-the-food-supply-chain.html.

[48] *Tensorflow 2*. URL: https://velog.io/@s_s/Tensorflow%EB%A5%BC-%ED%99%9C%EC%9A%A9%ED%95%9C-%EC%9D%B8%EA%B3%B5%EC%8B%A0%EA%B2%BD%EB%A7%9D-%EA%B5%AC%ED%98%842 (visited on 06/24/2025).

[49] Yunjie Tian, Qixiang Ye, and David Doermann. *YOLOv12: Attention-Centric Real-Time Object Detectors*. 2025. URL: https://github.com/sunsmarterjie/yolov12.

[50] Jane Torres. *How to Convert YOLOv8 to TFLite: A Step-by-Step Guide*. Accessed: 2025-09-21. 2024. URL: https://yolov8.org/how-do-you-convert-yolov8-to-tflite/.

[51] Ultralytics. *How to Use YOLO12 for Object Detection with the Ultralytics Package | Is YOLO12 Fast or Slow?* URL: https://docs.ultralytics.com/models/yolo12/ (visited on 03/26/2025).

[52] *Ultralytics YOLO11 - Ultralytics YOLO Docs*. URL: https://docs.ultralytics.com/models/yolo11/ (visited on 04/08/2025).

[53] Paul Viola and Michael Jones. «Rapid Object Detection using a Boosted Cascade of Simple Features». In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2001, pp. 511–518. DOI: 10.1109/CVPR.2001.990517.

[54] *YOLOv8 Architecture; Deep Dive into its Architecture - Yolov8*. URL: https://yolov8.org/yolov8-architecture/.

[55] Nurzarinah Zakaria and Yana Mazwin Mohmad Hassim. «Improved Image Classification Task Using Enhanced Visual Geometry Group of Convolution Neural Networks». In: *International Journal on Informatics Visualization* 7.4 (2023), pp. 2498–2505. ISSN: 25499904. DOI: 10.30630/joiv.7.4.1752.

DECLARAÇÃO

Declaro, sob compromisso de honra, que o trabalho apresentado neste relatório de estágio, com o título *"Implementation of AI-Powered Computer Vision Systems for Edge Devices Using YOLO and Face API"*, é original e foi realizado por João Rebelo dos Santos (2232646) sob orientação de Professora Doutora Marisa da Silva Maximiano (marisa.maximiano@ipleiria.pt).

Leiria, Setembro de 2025

João Rebelo dos Santos