COMPARING MACHINE LEARNING TECHNIQUES TO SEGMENTIZE AND CLASSIFY TONGUE REGIONS FOR TRADITIONAL AND COMPLEMENTARY MEDICINE (TCM) DIAGNOSIS

BY

BONG MIN XUAN

A REPORT

SUBMITTED TO

Universiti Tunku Abdul Rahman

in partial fulfillment of the requirements

for the degree of

BACHELOR OF INFORMATION SYSTEMS (HONOURS) INFORMATION SYSTEMS ENGINEERING

Faculty of Information and Communication Technology

(Kampar Campus)

JUNE 2025

COPYRIGHT STATEMENT

This Final Year Project report is submitted in partial fulfillment of the requirements for the degree of Bachelor of Information Systems (Honours) Information Systems Engineering at Universiti Tunku Abdul Rahman (UTAR). This Final Year Project report represents the work of the author, except where due acknowledgment has been made in the text. No part of this Final Year Project report may be reproduced, stored, or transmitted in any form or by any means, whether electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the author or UTAR, in accordance with UTAR's Intellectual Property Policy.

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, Dr. Lee Wai Kong, who has given me this invaluable opportunity to undertake this project, which focuses on applying machine learning techniques to segmentize tongue regions. Throughout the course of this project, Dr. Lee has provided unwavering guidance, insightful advice, and constant encouragement, helping me to navigate the challenges and complexities of the research. His patience, expertise, and commitment to teaching have profoundly deepened my understanding of machine learning and its real-world applications in medical imaging. This project not only marks an important milestone in my academic journey but also lays a strong foundation for my future endeavors in the field of artificial intelligence. I am truly grateful for the knowledge and professional development that I have gained under his supervision.

To a very special person in my life, Yvonne Wong, I extend my sincere thanks for your unconditional support, patience, and love. Yvonne's presence has been a constant source of strength, especially during the more challenging phases of this project. She has always believed in me even when I doubted myself, and your encouragement has given me the courage and determination to persevere. Her understanding, sacrifices, and continuous motivation have been an integral part of my success, and I could not have come this far without your unwavering companionship and positivity.

Lastly, I would like to express my deepest gratitude to my parents and family members for their endless support, love, and belief in my potential. Their sacrifices, encouragement, and constant reminders to strive for excellence have been the pillars that upheld me throughout my academic life. Every achievement I accomplish is a testament to the strength and inspiration that you have given me. Their faith in my abilities has fueled my motivation and has been the driving force behind my perseverance through every challenge faced along this journey. I am forever indebted to your boundless love and support.

A million thanks to all of them for being such an important part of my journey.

ABSTRACT

This project investigates the application of machine learning and deep learning techniques for automated tongue diagnosis in the context of Traditional Chinese Medicine (TCM). Tongue diagnosis, a long-established diagnostic method in TCM, is often limited by subjectivity and inconsistency. To address this, the study develops a systematic pipeline that integrates segmentation and classification models, enabling more objective, accurate, and reproducible analysis of tongue images. Three datasets—binary (stained vs. non-stained moss), four-class (color variations), and five-class (coating categories)—were utilized to evaluate performance under varying levels of complexity. Segmentation was performed using both classical methods (SVM) and a deep learning approach (DuckNet), with DuckNet providing superior accuracy and robustness. Classification was carried out through an evolutionary series of architectures, beginning with AdderNet and progressing through ResNet20, HybridNet, and an Improved HybridNet. Experimental results demonstrated that while AdderNet achieved the highest accuracy in complex multi-class scenarios, it suffered from excessive computational cost and scalability limitations. The Improved HybridNet consistently offered the best trade-off between performance and efficiency, delivering strong accuracy with reduced parameters, training time, and model size. Overall, the project highlights the potential of artificial intelligence to modernize tongue diagnosis by providing standardized, efficient, and clinically relevant computational tools. The findings establish a foundation for future integration of AI-driven diagnostic support systems into healthcare practice

Area of Study: Image Processing, Artificial Intelligence
Keywords: Image Processing, TCM, Deep Learning, Image Classification, Machine Learning

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1
Introduction

1.1 Problem Statement and Motivation

Traditional Chinese Medicine (TCM) relies heavily on tongue diagnosis to assess patients' internal health conditions. However, manual tongue inspection is inherently subjective, with diagnostic accuracy depending on the practitioner's experience and perception. This subjectivity often results in inconsistent outcomes and limits reproducibility. With the increasing availability of digital tongue image datasets, computational methods now present a significant opportunity to provide standardized, objective, and data-driven support for diagnosis. The main challenge lies in developing automated systems capable of accurately segmenting tongue regions and classifying subtle variations in color and coating while maintaining computational efficiency for real-world use. Therefore, this project seeks to design, evaluate, and optimize computational models that can support tongue diagnosis in a more reliable and standardized manner, bridging the gap between traditional practice and modern artificial intelligence.

1.2 Objectives

The objectives of this project are as follows:

1. To investigate and compare segmentation approaches using both traditional machine learning and deep learning methods for accurate tongue region isolation.
2. To design and implement an evolutionary series of classification models
3. To assess performance using multiple evaluation metrics, including accuracy, precision, recall, F1-score, Jaccard index, and computational efficiency indicators across different datasets.
4. To determine the most effective model that trade-off between accuracy and efficiency, with the aim of proposing a practical solution for diagnostic support applications.

1.3 Project Scope and Direction

This project focuses primarily on the classification of tongue images, with segmentation applied as a preprocessing step to ensure clinically relevant features are extracted. Three datasets are employed: a binary dataset (stained vs. non-stained moss), a four-class dataset (color variations), and a five-class dataset (coating categories). Classical methods such as Support Vector Machines (SVM) are included as baselines, while DuckNet represents the deep learning-based segmentation approach. For classification, the project implements an evolutionary sequence of convolutional neural network architectures, starting with AdderNet and culminating in an improved HybridNet. The scope is deliberately restricted to computational model development rather than hardware prototyping or direct clinical validation, in order to ensure reproducibility, controlled benchmarking, and feasibility within the academic timeframe. The overall direction emphasizes building reproducible pipelines, systematically benchmarking architectures, and analyzing the trade-offs between model accuracy and computational efficiency.

1.4 Contributions

This study makes several contributions toward advancing the role of artificial intelligence in modernizing tongue diagnosis and medical image analysis more broadly. First, it **bridges tradition and technology** by showing how data-driven approaches can reduce the subjectivity of traditional diagnostic practices, offering a more consistent and objective analysis of tongue images. In addition, the project provides an **evaluation of methodological strategies**, systematically comparing segmentation and classification approaches to reveal how different computational techniques address challenges such as unclear boundaries, lighting variation, and subtle visual differences in medical imagery. Another contribution is the focus on **promoting efficiency for real-world use**, where the study emphasizes model designs that balance diagnostic reliability with computational efficiency, making automated systems more feasible in practical contexts, including resource-constrained environments. Beyond technical results, the study offers **guidance for future development** by highlighting the importance of dataset quality, diversity, and preprocessing in building reliable diagnostic tools, providing insights that extend to broader medical AI applications. Finally, the work contributes by **laying a foundation for clinical integration**, presenting a structured evaluation framework that can

guide future efforts to incorporate automated tongue diagnosis into healthcare practice, thereby supporting more standardized, accessible, and efficient clinical decision-making.

1.5 Report Organization

This report is organized into six chapters. **Chapter 1 (Introduction)** outlines the project background, motivation, problem statement, objectives, scope, contributions, and report structure. **Chapter 2 (Literature Review)** surveys relevant technologies, datasets, segmentation methods, and classification models, highlighting the strengths and weaknesses of both traditional and deep learning approaches in medical image analysis. **Chapter 3 (System Methodology/Approach)** details the overall workflow, including dataset preparation, segmentation strategies, classification model design, evaluation metrics, and the implementation environment. **Chapter 4 (System Design)** presents the architectural details of each classification model—AdderNet, ResNet20, HybridNet, and the Improved HybridNet—emphasizing the evolutionary design choices made to balance efficiency and performance. **Chapter 5 (Results and Discussion)** reports experimental findings, comparing segmentation and classification outcomes across different datasets, analyzing computational efficiency, and identifying the most effective model. Finally, **Chapter 6 (Conclusion and Recommendation)** summarizes the project's contributions, key insights, and limitations, while offering recommendations for future work and potential improvements.

Chapter 2

Literature Review

2.1 Review of the Technologies

2.1.1 Hardware Platforms for Medical Image Analysis

The acceleration of tongue image processing, particularly for segmentation and classification tasks, has been significantly enhanced by the use of modern hardware. High-performance GPUs are essential for training complex deep learning models like DuckNet, which require substantial computational power for pixel-level prediction. For instance, studies utilizing architectures similar to U-Net for tongue segmentation have leveraged GPUs like the NVIDIA Tesla T4 and RTX series to reduce training times from days to hours. Furthermore, for practical deployment in clinical or mobile settings, there is a growing research focus on optimizing these models for lightweight edge devices such as the Jetson Nano and Google Coral. These platforms enable real-time analysis, showing promise for portable TCM diagnostic systems that could be deployed in clinics or for remote consultations [1].

2.1.2 Firmware / Operating System Environments

The development of AI-driven tongue diagnosis systems predominantly occurs in Linux-based environments (e.g., Ubuntu) due to superior compatibility with deep learning frameworks, GPU drivers, and development tools. However, the barrier to entry for such setups has been lowered by the advent of cloud-based platforms. Environments like Google Colab, which was used in this project, provide pre-configured, GPU-accelerated access to Jupyter notebooks, drastically simplifying experimentation and ensuring reproducibility without the need for local hardware configuration. While the choice of OS has minimal direct impact on model accuracy, it is crucial for development efficiency. The reproducibility and ease of collaboration offered by these cloud platforms have made them a popular choice in recent literature for prototyping medical image analysis systems, including those for TCM [2].

2.1.3 Datasets Used in Medical and Tongue Diagnosis

Datasets are foundational to building robust models. In the field of TCM, datasets like the TCM Tongue Image Dataset, SciTongue, and Baidu Tongue Coating Images are commonly used.

However, most are limited by small size, inconsistent labeling, or poor lighting conditions. Beyond tongue images, datasets such as ISIC 2020 (for skin lesions), LIDC-IDRI (lung CT scans), and DRIVE (retinal vessel segmentation) are often used to benchmark segmentation algorithms, providing insights into generalizable methods for medical imaging [3].

2.1.4 Programming Languages and Libraries

The overwhelming majority of literature utilizes Python as the primary programming language, largely due to its rich ecosystem of libraries for image processing and machine learning. Libraries such as TensorFlow, Keras, PyTorch, scikit-learn, and OpenCV enable rapid prototyping and development. According to a survey [4], over 90% of papers on deep learning in medical image classification from 2020 to 2022 used Python-based frameworks.

2.2 Review of Existing Systems and Applications

2.2.1 Tongue Diagnosis Systems in TCM

Tongue diagnosis is a cornerstone of Traditional Chinese Medicine (TCM), used to assess internal health by analyzing the tongue's color, shape, and coating; however, its traditional practice is highly subjective and prone to inconsistency. This limitation has driven the development of computer-aided diagnostic (CAD) systems, which have evolved from using hand-crafted features with classical machine learning models like SVMs to modern deep learning approaches that offer superior accuracy and robustness. Current research leverages convolutional neural networks (CNNs) such as U-Net for segmentation and various classifiers for diagnosis, yet a significant challenge remains in balancing high performance with computational efficiency for practical clinical deployment. This project addresses that gap by focusing on the development and evaluation of lightweight, efficient deep learning models specifically designed for deployable tongue image analysis, aiming to provide a reliable and accessible tool for modern TCM practice [5].

2.2.2 Traditional and Machine Learning-Based Segmentation

Early research in tongue image segmentation primarily relied on traditional image processing techniques such as thresholding, edge detection, and region growing, which attempted to isolate

the tongue region based on color intensity or texture. However, these methods often suffered from noise sensitivity, especially in images captured under non-uniform lighting or with complex backgrounds.

Subsequently, classical machine learning techniques were introduced to improve segmentation accuracy. For example, K-means clustering was widely adopted to partition tongue images into foreground and background based on pixel intensity clusters. Support Vector Machines (SVM) and Random Forest classifiers were later used for pixel-level classification using hand-crafted features such as color histograms, Gabor filters, or texture descriptors.

Although these methods offered incremental improvements, they were still dependent on manual feature extraction and lacked adaptability across diverse datasets. A survey [6] showed that classical ML methods, while more interpretable and computationally efficient to train, were outperformed by modern neural network-based approaches in terms of accuracy and robustness. Nevertheless, their efficiency makes them a valuable baseline for comparison, which is why methods like SVM are included in this study to benchmark the performance gains of deep learning models [7].

2.2.3 Deep Learning-Based Segmentation Models

The transition to deep learning has revolutionized medical image segmentation, with encoder-decoder architectures like U-Net becoming the gold standard. These models excel at precise pixel-level classification, which is critical for isolating the tongue region from complex and inconsistent backgrounds in clinical images. In tongue diagnosis, U-Net and its variants (e.g., DuckNet) are predominantly used due to their skip connections that preserve fine-grained spatial details necessary for accurate boundary delineation, a foundational step before any classification can occur.

2.2.3.1 Convolutional Neural Network (CNN – Base Architecture)

Convolutional Neural Networks form the foundational building block for most deep learning models in image analysis. Their ability to automatically learn hierarchical features—from edges and textures to complex patterns—makes them superior to hand-crafted feature methods. In tongue diagnosis, basic CNNs can perform initial classification tasks but are inherently limited. Their relatively shallow architecture struggles to capture the subtle and nuanced features critical for TCM, such as fine cracks or slight color variations in the coating, and they

are highly susceptible to overfitting on small, specialized medical datasets, serving primarily as a performance baseline [8].

2.2.3.2 VGG16

VGG16 addresses the depth limitation of basic CNNs through a uniform and deeper 16-layer architecture. This allows it to learn more complex feature representations, making it a strong candidate for classifying detailed tongue coating types. However, this gain in representational power comes at a significant cost: its massive number of parameters leads to high computational load and memory consumption, rendering it impractical for real-world deployment where efficiency is a priority, thus establishing a clear trade-off between accuracy and operational feasibility [9].

2.2.3.3 ResNet20

ResNet20 introduces a pivotal innovation with residual skip connections, which mitigate the vanishing gradient problem and enable the effective training of deeper networks. This architecture achieves a more favorable balance than VGG16, offering improved feature extraction capabilities for discerning tongue color and morphology without an excessive parameter count. Its stable and well-understood design makes it an ideal standardized benchmark or backbone model for controlled comparisons in research, allowing subsequent architectural modifications to be evaluated fairly without the confounding variable of training instability **[10]**.

2.2.3.4 MobileNetV2

MobileNetV2 represents a strategic shift towards efficiency, employing depthwise separable convolutions and inverted residual blocks to drastically reduce computational complexity and model size. This design is explicitly intended for mobile and embedded deployment, making it highly relevant for developing practical, real-time diagnostic tools. While its lightweight nature can sometimes come at a minor cost to accuracy on highly complex tasks, it provides a crucial foundation for designing models where speed and low power consumption are paramount, directly addressing the deployability goals of modern medical AI [11].

2.2.3.5 AdderNet

AdderNet pushes the efficiency frontier further by fundamentally rethinking the convolutional operation, replacing multiplications with additions to reduce computational energy expenditure. This presents a promising pathway toward ultra-low-power diagnostic systems. However, as a novel architecture, it faces challenges in training stability and a lack of hardware optimization, making it more experimental. Its exploration is valuable for probing the limits of efficiency but requires careful benchmarking against more established models to validate its effectiveness on specialized medical imagery like tongue features [12].

2.2.3.6 Summary of Classification Models

This progression of architectures reveals a clear trade-off in medical image analysis: representational capacity versus computational efficiency. While models like VGG16 and ResNet20 provide strong accuracy, their resource demands hinder practical application. This review justifies the evolutionary approach of this project, which begins with the experimental AdderNet and ResNet20 benchmark before strategically integrating the efficiency principles of MobileNetV2. The goal is not merely to select an existing model, but to engineer a new architecture that hybridizes the stability of residual learning with the extreme efficiency of inverted residuals and depthwise convolutions. This synthesis aims to achieve an optimal balance for accurate, deployable, and real-time tongue diagnosis, directly addressing the identified gap between robust performance and practical utility.

Chapter 3

System Methodology/Approach

3.1 Overview

This project implements a structured pipeline for automated tongue image analysis, consisting of distinct **data preparation**, **segmentation**, **classification** and **evaluation** stages. The methodology is designed to evaluate the effectiveness of different techniques at each step across three datasets.

The workflow begins with three tongue image datasets: a **binary (2-class) dataset** with pre-segmented images, and two multi-class datasets (**4-class and 5-class**) requiring manual preprocessing. For the multi-class datasets, a crucial **segmentation** step is applied to isolate the tongue region. This step utilizes a high-performance deep learning model (DuckNet), selected after a comparative analysis with traditional machine learning methods (**SVM and Random Forest**).

The outputs from this pipeline—the pre-segmented 2-class images and the newly segmented 4-class and 5-class images—are then used for the **classification** task. Classification is performed using a progressive sequence of CNN architectures, from an exploratory model (AdderNet) to a conventional baseline (ResNet20), and finally to efficiency-optimized designs (HybridNet and Improved HybridNet).

A comprehensive **evaluation** follows, where all models are rigorously assessed using standard performance metrics (e.g., accuracy, precision, recall) and efficiency indicators (e.g., model size, training time). This multi-faceted evaluation provides a complete understanding of each model's trade-offs, determining the most suitable architecture for accurate and practical tongue image analysis.

3.2 Dataset Preparation

This section details the acquisition and preparation of the three tongue image datasets used for classification. The process for the multi-class datasets involved a standardized pipeline of resizing, manual annotation, and targeted augmentation to ensure balance and quality before segmentation.

1. **Binary Dataset (2 Classes)**

Source: SciDB Tongue Image Database [13].
Classes: Stained Moss, Non-Stained Moss.
Preparation: This dataset was provided with pre-segmented tongue regions. Each class already contained more than 1,000 images, fulfilling the target dataset size without requiring augmentation. All images were resized to 224×224 pixels.
Size & Split: The final dataset consists of **2,000 images (1,000 per class)**, split into **1,600 for training and 400 for testing** (80:20 ratio).
Purpose: Serves as a benchmark for binary classification of tongue moss presence.

2. **Multi-Class Dataset (4 Classes)**

Source: Self-labeled and combined from multiple public sources, primarily Baidu AI Studio [14], supplemented with images from Kaggle [15], [16] and other repositories [17], [18].
Classes: Pale, Pale Red, Red, Bluish Purple.
Preparation: The initial collection had an uneven class distribution. The following pipeline was applied:

- **1. Resizing:** All images were first standardized to a resolution of 224×224 pixels.
- **2. Annotation:** Each resized image was then annotated using the CVAT tool (semi automation annotation) to obtain mask for further segmentation.
- **3. Targeted Augmentation:** To create a balanced dataset, classes with fewer than 500 samples were augmented using transformations (e.g., rotation, flipping) to reach the target of 500 images per class.

Size & Split: The final, balanced dataset consists of **2,000 images total (500 per class)**, split into **1,600 for training and 400 for testing**.
Purpose: Focuses on significant tongue color-based categorization. This dataset requires segmentation, as detailed in Section 3.3.

3. **Multi-Class Dataset (5 Classes)**

Source: Obtained directly from the project supervisor.

Classes: Mirror-Approximated, White-Greasy, Thin-White, Yellow-Greasy, Grey-Black.

Preparation: The initial class distribution was uneven. The preparation involved.

- **1. Resizing:** All images were first standardized to a resolution of 224×224 pixels.
- **2. Annotation:** Each resized image was annotated using the CVAT tool to obtain mask for further segmentation.
- **3. Targeted Augmentation:** The same augmentation strategy was applied to achieve a final balance of 500 images per class.

Size & Split: The final, balanced dataset consists of **2,500 images total (500 per class)**, split into **2,000 for training and 500 for testing**.

Purpose: Focuses on classifying variations in tongue coating. This dataset requires segmentation (Section 3.3).

Sample datasets

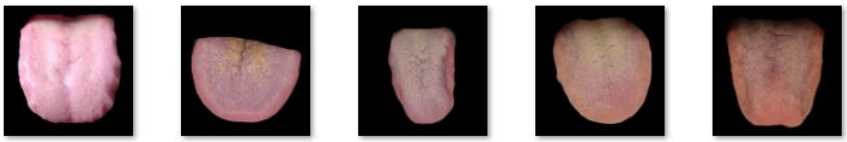
Class	Category	Sample Images
2	Stained Moss	
	Non-Stained Moss	
4	Pale	
	Pale Red	
	Red	
	Bluish Purple	
5	Mirror-Approximated	
	White-Greasy	
	Thin-White	
	Yellow-Greasy	
	Grey-Black	

Figure 3.2: Representative samples for categories across 2-, 4-, and 5-class datasets

Annotation using CVAT

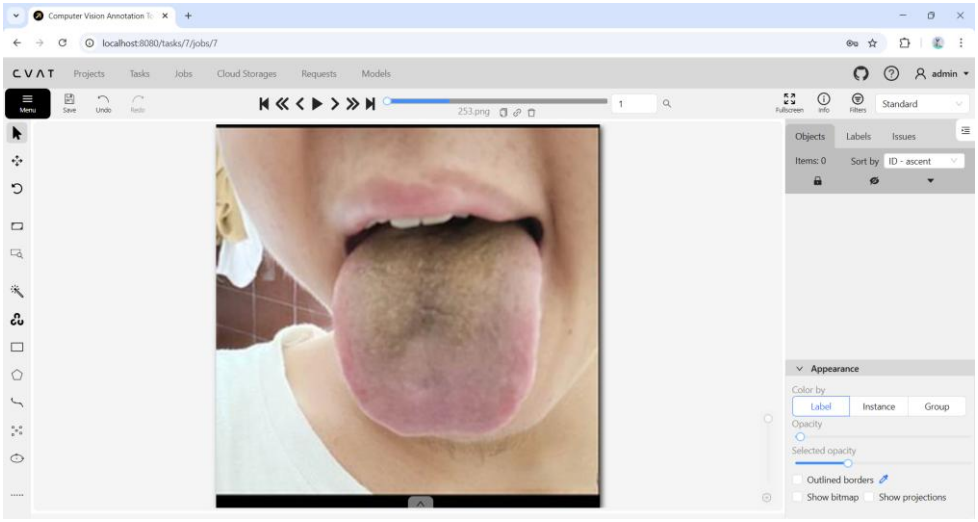


Figure 3.2.1: Tongue images from the 4 and 5 class datasets imported into the CVAT tool

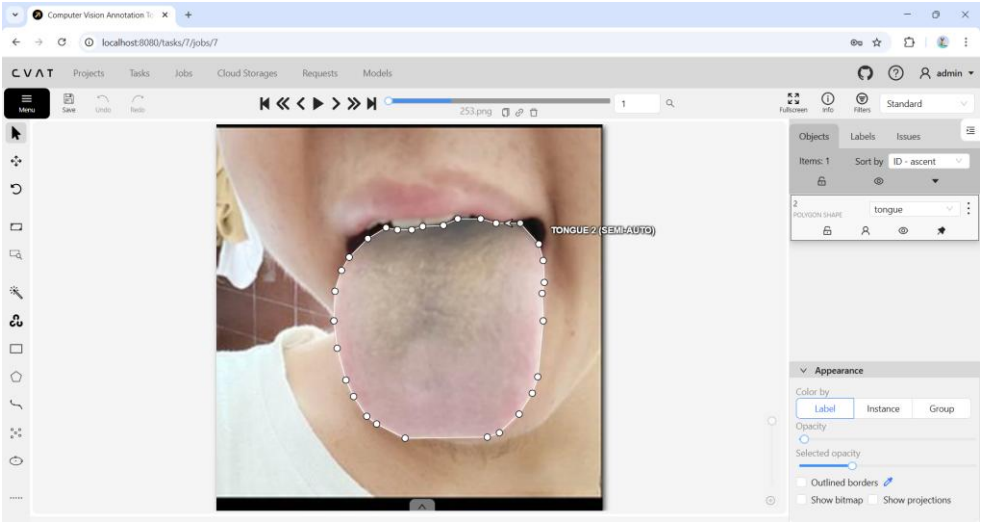


Figure 3.2.2: Semi auto annotation of the tongue region

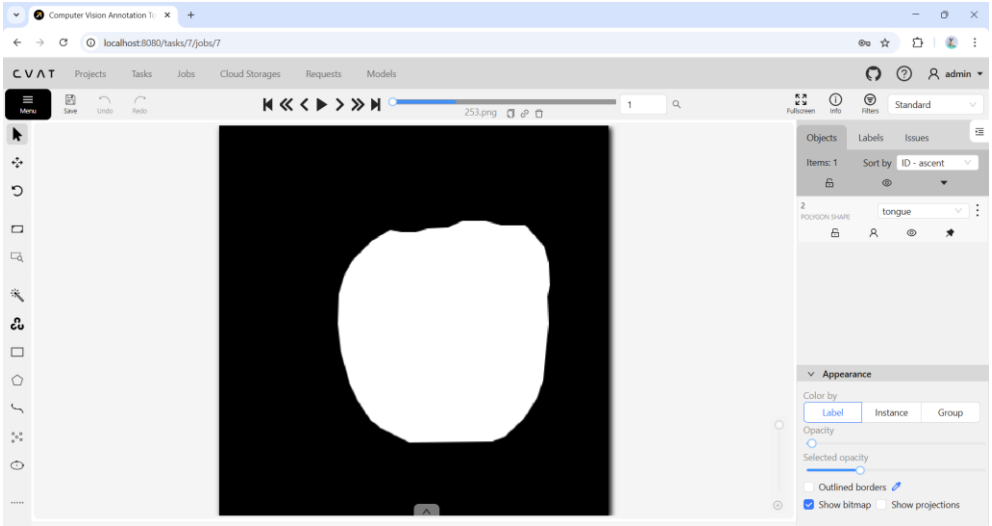


Figure 3.2.3: Ground truth mask generated for segmentation

3.3 Segmentation Methods

Segmentation is a critical preprocessing step that isolates the tongue region from the background, ensuring that classification models learn only from relevant features like coating, color, and texture. **This step was applied exclusively to the 4-class and 5-class datasets**, as the binary dataset was provided with pre-segmented images. Two segmentation paradigms were explored: classical machine learning methods from FYP1 and a deep learning-based approach.

3.3.1 Traditional Machine Learning Methods

The initial investigation involved two classical techniques: Support Vector Machine (SVM) and Random Forest (RF). Both methods classify pixels as tongue or background using manually engineered features, including texture (Local Binary Patterns), color (RGB channels), and spatial information (distance from center).

- **Random Forest (RF):** While achieving high recall (99.25%), this ensemble method was computationally intensive. It required a prohibitively long **training time of approximately 4 hours** and resulted in a large model size (227.5 MB) due to its deep decision trees.
- **Support Vector Machine (SVM):** Selected as a lightweight and efficient alternative, SVM delivered high precision with a drastically shorter **training time of about 1 hour** and a minimal model size (0.29 KB). However, it struggled with complex boundaries due to its reliance on a linear kernel.

These methods provided a strong baseline for comparative analysis against deep learning approaches.

3.3.2 DuckNet (Deep Learning Segmentation)

DuckNet is a fully convolutional neural network with an encoder-decoder structure, similar to U-Net, but enhanced for efficient multi-scale feature extraction using custom convolutional blocks. This model was implemented based on the architecture from https://github.com/RazvanDu/DUCK-Net.

- **Architecture:** It utilizes a series of strided convolutional layers for downsampling and skip connections with element-wise addition to merge features from the encoder and decoder paths.
- **Implementation:** A critical hyperparameter is the number of starting filters, which controls the model's capacity. Based on extensive experimentation in FYP1, the model was configured with **17 starting filters**.
- **Justification for 17 Filters:** As detailed in Table 4.5.1 of the FYP1 report, variants with 4, 8, 12, 17, and 34 filters were tested. The 17-filter configuration achieved an optimal balance, delivering high accuracy (99.60%) and a strong Dice score (0.9878) without the computational overhead of the larger 34-filter model, which showed only marginal improvement (0.9967 accuracy) at quadruple the parameter cost. This represents the point of diminishing returns for this specific task.
- **Strengths:** Automatically learns optimal pixel-level representations from data, capable of handling complex variations in tongue appearance. Achieves significantly higher accuracy than traditional methods.

3.3.3 Method Selection

For consistency and reliability, **the 17-filter DuckNet model was adopted as the primary segmentation method for the 4-class and 5-class datasets**. Although **SVM served as an efficient traditional baseline** with a fast training time, DuckNet was the default choice due to its superior robustness, accuracy, and generalization ability. The inclusion of both traditional methods demonstrates the clear performance-efficiency trade-off between classical and deep learning methodologies for this task.

3.4 Classification Models

Following segmentation, tongue images were classified using a sequence of progressively refined convolutional neural network architectures. The detailed system design, including specific modules and training hyperparameters for each model, is provided in **Chapter 4**. The model progression is as follows:

3.4.1 AdderNet

- **Origin:** Adopted from Huawei Noah's Ark Lab (https://github.com/huawei-noah/AdderNet) to explore **addition-based convolutions** as a hardware-efficient alternative.
- **Role:** Served as an **exploratory model** to test the feasibility of this novel approach.
- **Outcome: Computationally heavy training** due to unoptimized operators motivated a pivot to a standardized baseline.

3.4.2 ResNet20

- **Rationale:** A **conventional baseline** created by converting AdderNet back to standard convolutions, retaining the residual (3-3-3) structure.
- **Role:** Acts as a **controlled benchmark** to isolate the effect of subsequent architectural changes.

3.4.3 HybridNet

- **Motivation:** Integrates **MobileNetV2's efficiency principles** (inverted residuals, depthwise separable convolutions) into a residual framework.
- **Design Intent:** To create a **lightweight architecture** that maintains performance while improving computational efficiency.

3.4.4 Improved HybridNet

- **Motivation:** An enhanced version designed for superior **generalization and deployability**.

- **Refinements:** Incorporates **Squeeze-and-Excite (SE)** modules, **DropPath** regularization, and **activation checkpointing**.
- **Role:** The **final candidate architecture**, explicitly engineered for an optimal balance of accuracy, robustness, and efficiency.

3.5 Evaluation Metrics

To comprehensively assess system performance, both effectiveness and efficiency were evaluated. The evaluation strategy was designed to capture not only classification accuracy but also robustness across classes and practical deployability of models.

3.5.1 Classification Metrics

The following standard performance metrics were used for classification tasks:

- **Accuracy:** The ratio of correctly predicted samples to the total number of samples.
- **Precision:** The proportion of correctly predicted positive samples relative to all predicted positives, useful for measuring reliability in clinical contexts.
- **Recall (Sensitivity):** The proportion of correctly predicted positive samples relative to all actual positives, ensuring that clinically significant cases are not overlooked.
- **F1 Score:** The harmonic mean of Precision and Recall, balancing the trade-off between false positives and false negatives.

All metrics were calculated **per class** and **overall**, ensuring that performance differences between categories (e.g., pale vs. red tongues) were fully captured.

3.5.2 Segmentation Metrics

For segmentation tasks, an additional metric was included:

- **Jaccard Index (Intersection over Union):** Measures the overlap between predicted segmentation masks and ground-truth labels. This metric is particularly relevant for medical image segmentation, where precise region delineation is crucial.

3.5.3 Efficiency Metrics

In addition to predictive performance, the efficiency of each model was also evaluated, reflecting its practicality for real-world applications:

- **Training Time:** Total time required to train a model under a fixed configuration.
- **Parameter Count:** The total number of trainable parameters in the model, serving as an indicator of model complexity.
- **Model Size:** The storage footprint of the trained model, measured in megabytes.

These efficiency indicators provide a complementary view of model performance, balancing accuracy with deplorability considerations such as memory footprint and computational requirements.

3.6 Implementation Environment

All model training and evaluation were conducted using **Google Colab**, a cloud-based integrated development environment (IDE) that provides free GPU access. The following setup was used throughout the project:

- **Execution Environment:** Google Colab (Jupyter Notebook interface).
- **GPU:** NVIDIA Tesla T4 GPU, 16 GB GPU memory (allocated by Colab).
- **Code Management:**
    - All project code files (.py) were stored on **Google Drive**.
    - A Jupyter Notebook (.ipynb) was created for each model (AdderNet, ResNet20, HybridNet, Improved HybridNet).
    - Each notebook imported the corresponding .py scripts from Google Drive, enabling modular execution and reproducibility.
- **Software:**
    - Python 3.x with PyTorch and torchvision as the main deep learning frameworks.
    - Supporting libraries included NumPy, OpenCV, scikit-learn, Matplotlib, and seaborn for metrics visualization.

This environment provided sufficient computational resources to train deep learning models within practical time limits while maintaining a reproducible workflow that could be re-executed directly from cloud storage.

Chapter 4
System Design

4.1 Overview

The system design describes the detailed implementation of classification models for automated tongue image analysis. Each model was implemented in **PyTorch**, with a **shared training and evaluation pipeline** optimized for reproducibility. The architectures followed an evolutionary progression — **AdderNet → ResNet20 → HybridNet → Improved HybridNet** — where each stage introduced new design elements to balance classification accuracy and computational efficiency.

Shared Training & Evaluation Pipeline:

- **Training Script (**main.py**)**: A supervised training loop using **SGD with momentum**, a **cosine learning rate schedule**, and **CrossEntropyLoss**.
- **Evaluation Script (**test.py**)**: Computes metrics and generates confusion matrices.
- **Preprocessing**: All inputs were resized to **224×224** and normalized using dataset-specific mean and standard deviation values.
- **Loss Function**: All models were optimized using **categorical cross-entropy loss**.

4.2 AdderNet Design

AdderNet was chosen as the starting point to explore a novel operation: replacing convolution multiplications with addition operations, potentially reducing energy consumption.

- **Custom Operation – adder2d:**
  Implemented in *adder.py*, this operation replaces convolution's dot product with the absolute difference:

$$Y(i,j,k) = -\sum_{m=1}^{M}\sum_{n=1}^{N}\left|W_{m,n,k} - X_{i+m,j+n}\right|$$

where W are filter weights and X is the local input patch.

- **Architecture (resnet20.py with adder layers):**
  - Residual blocks constructed using `adder2d` instead of Conv2d.
  - Each block: Adder → BatchNorm → ReLU → Adder → BatchNorm → Residual Add → ReLU.
  - Stacked [3,3,3] blocks (20 layers).
  - Adaptive Average Pooling → Fully connected classifier.

Role in progression: Served as the **base model**, testing feasibility of addition-based convolutions. Limitation: high training time due to lack of GPU optimization.

4.3 ResNet20 Design

ResNet20 provided a conventional CNN baseline to isolate the effect of AdderNet's addition operations while retaining residual learning.

- **Architecture:**
  - Residual blocks with Conv2d layers.
  - Same [3,3,3] block structure as AdderNet.
  - Global Average Pooling → classifier head.
- **Rationale:**
  - Establishes a benchmark CNN.
  - Retains residual learning but removes AdderNet's complexity.

Training Loss Function: All CNNs were optimized using categorical cross-entropy loss:

$$\mathcal{L}_{CE} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(\hat{y}_{i,c})$$

where $y_{i,c}$ is the ground truth and $\hat{y}_{i,c}$ the predicted probability for class ccc.

Role in progression: ResNet20 acted as the **benchmark backbone**, providing a stable reference for further improvements.

4.4 HybridNet Design

HybridNet was introduced to integrate the **stability of ResNet** with the **efficiency of MobileNetV2**, aiming to reduce parameters without sacrificing performance.

- **Inverted Residual Block:**
  - Expansion → depthwise convolution → projection sequence.
  - Residual connection applied if stride=1 and channels match.
- **Hybrid Block:**
  - Wraps an InvertedResidual with ResNet-style shortcut projection when needed.
  - Adds outputs with ReLU activation.
- **HybridNet Architecture (*hybrid.py*):**
  - Stem: Conv3×3 → BN → ReLU.
  - Three stages of HybridBlocks (channels: 16 → 24 → 48 → 96).
  - Adaptive Average Pooling → Fully connected classifier.

Role in progression: To introduce **residual stability + depthwise efficiency** could produce a lighter network (~228K parameters) while maintaining strong performance.

4.5 Improved HybridNet Design

Improved HybridNet incorporated **modern refinements** to further enhance accuracy and efficiency.

- **Key Enhancements:**
  - DropPath: stochastic depth regularization.
  - Squeeze-and-Excite (SE): channel attention.
  - Activation Checkpointing: reduced memory cost.
  - Dynamic Downsampling: adaptive resolution reduction.

    - Reduced Expand Ratio: lowered from 6.0 → 3.0, reducing FLOPs.
    - Dropout & Weight Initialization: improve stability.
- **Architecture Flow:**
    - Stem (Conv3×3).
    - Stacked HybridBlocks with SE + DropPath.
    - Global Average Pooling → Dropout → Fully connected classifier.

Role in progression: The **final optimized model**, achieving the best trade-off: smallest size (~131K params, 0.57 MB), fastest training (~20–25 min), and highest efficiency.

4.6 Summary of Model Architectures and Their Roles in Progression

Model	Core Idea	Key Modules	Purpose in Progression
AdderNet	Replace Conv2d multiplications with L1-norm addition operations.	adder2d layers, standard residual blocks (BasicBlock).	To serve as a **foundational model** and explore the feasibility of a **novel, multiplication-light approach** for feature extraction on tongue imagery.
ResNet20	Establish a conventional, highly-optimized residual CNN baseline.	Standard Conv2d layers, BatchNorm, ReLU, and residual blocks (BasicBlock) with projection shortcuts.	To provide a **standardized benchmark** for fair comparison, ensuring performance differences are due to architecture, not training procedure.
HybridNet	Fuse the stability of ResNet with the efficiency of MobileNetV2.	**HybridBlock** (1x1 expansion conv → Depthwise conv → 1x1 projection conv + residual	To introduce **efficiency-focused design**, testing if lightweight, inverted residual blocks can

		connection), linear bottlenecks.	maintain accuracy with fewer parameters and FLOPs.
Improved HybridNet	Enhance HybridNet with modern attention, regularization, and memory optimization techniques.	**Squeeze-and-Excite (SE)** blocks for channel attention, **DropPath** for regularization, reduced expansion ratios, activation checkpointing.	To be the **final candidate architecture**, explicitly engineered for superior **robustness, generalization, and deployability** through targeted refinements.

Table 4.6: Summary of Model Architectures and Their Roles in Progression

Chapter 5
Results And Discussion

5.1 Segmentation Performance

Segmentation was applied to the multiclass datasets (4-class and 5-class) using two methods: a traditional SVM classifier and the deep learning-based DuckNet architecture. SVM was selected over Random Forest (RF) from prior work due to its significantly shorter training time (~1 hour vs. ~4 hours) for equivalent performance.

SVM

The SVM model achieved moderate segmentation results, as shown in Table 5.1.1. It is important to note that its performance metrics are lower than those reported in FYP1. This discrepancy is primarily due to the increased complexity of the FYP2 datasets. Unlike the FYP1 dataset, which was captured under controlled conditions, the images in the 4-class and 5-class datasets contain more challenging backgrounds, such as varying clothing colors and environments. These complex backgrounds make the segmentation task more difficult for a traditional, feature-based method like SVM.

Dataset	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Jaccard (%)	Training time
4 class	86.21	84.73	79.69	82.13	69.68	55h 49s
5 class	86.02	86.45	79.41	82.78	70.62	01h 11m 34s

Table 5.1.1: Segmentation performance of SVM on 4- and 5-class datasets

Total parameters: 45

Model size: 0.35 KB

DuckNet

In contrast, DuckNet demonstrated exceptional performance, consistently exceeding 98% accuracy and 95% Jaccard index across both datasets (Table 5.1.2). Its deep learning architecture enabled it to learn robust features capable of handling the complex backgrounds that challenged the SVM model.

Dataset	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Jaccard (%)	Training time
4 class	98.69	97.28	98.17	97.73	95.55	03h 54m 06s
5 class	98.53	98.85	97.77	98.31	96.67	03h 58m 10s

Table 5.1.2: Segmentation performance of DuckNet on 4- and 5-class datasets

Total parameters: 38,921,088
Model size: 148.47 MB

Discussion:

SVM is extremely lightweight (45 parameters; 0.35 KB) and CPU-friendly, but it underperforms on the more complex FYP2 imagery. DuckNet, though much larger (38.9 M parameters; 148.47 MB), delivers consistently superior segmentation (≥98% accuracy; ≥95% Jaccard) on both datasets. We therefore adopt DuckNet as the default pre-processing for all classification experiments. To quantify sensitivity to segmentation quality, we also report results using SVM-segmented data in subchapter 5.2–5.3.

Figure 5.1: Sample tongue images

Figure 5.2: Ground truth masks of tongue regions

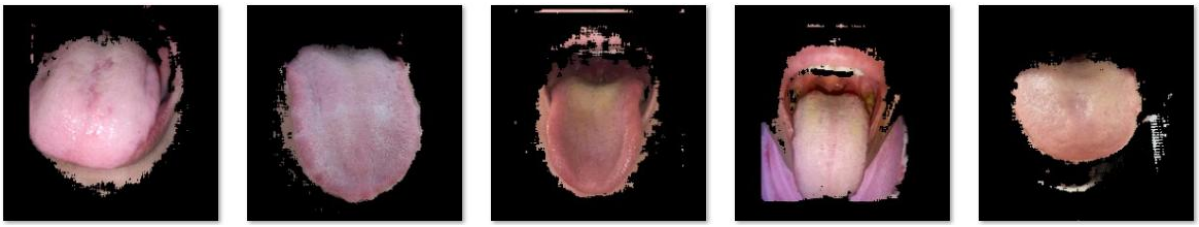


Figure 5.3: Tongue regions segmented using SVM

Figure 5.4: Tongue regions segmented using DuckNet

5.2 Classification Performance on SVM Segmented Datasets

5.2.1 Multi (4-Class) Dataset Results

5.2.1.1 ResNet20

Method	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Pale	87.00	84.47	87.00	80.65
Pale red	75.00	87.21	75.00	80.65
Red	89.00	91.75	89.00	90.36
Bluish purple	99.00	86.84	87.00	85.71
Overall	**87.50**	**87.57**	**87.50**	**87.31**

Table 5.2.1.1: Classification performance of ResNet20 on the 4-class dataset (SVM-segmented)

Training time: 20m 08s

Total parameters: 274,008

Model size: 1.09 MB

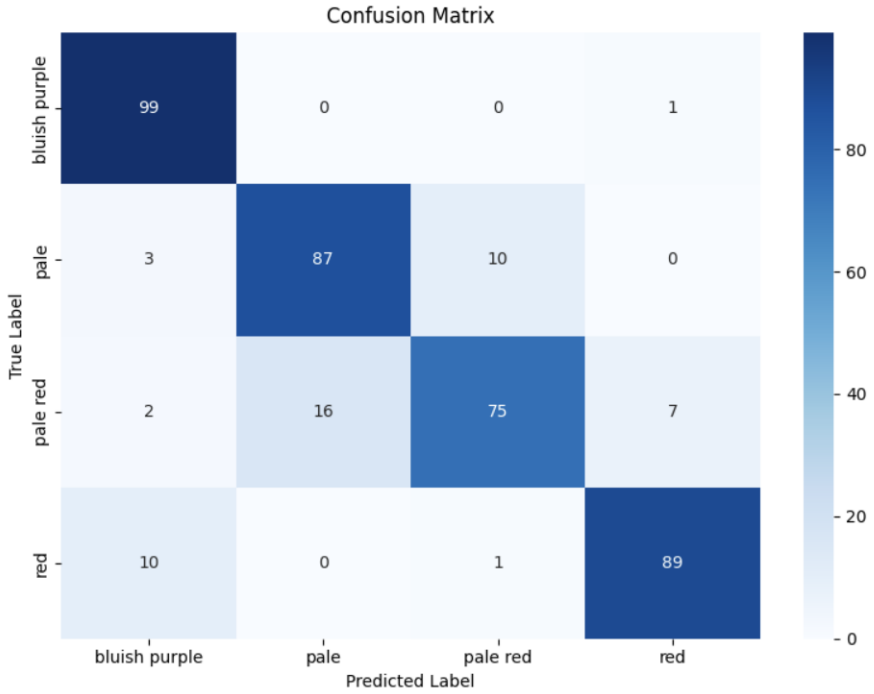


Figure 5.2.1.1: Confusion matrix of ResNet20 on the 4-class dataset (SVM-segmented)

5.2.1.2 HybridNet

Method	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Pale	93.00	76.86	93.00	84.16
Pale red	65.00	92.86	65.00	76.47
Red	91.00	91.92	91.00	91.46
Bluish purple	100.00	90.91	100.00	95.24
Overall	**87.25**	**90.91**	**100.00**	**95.24**

Table 5.2.1.2: Classification performance of HybridNet on the 4-class dataset (SVM-segmented)

Training time: 01h 24m 23s

Total parameters: 228,804

Model size: 0.95 MB

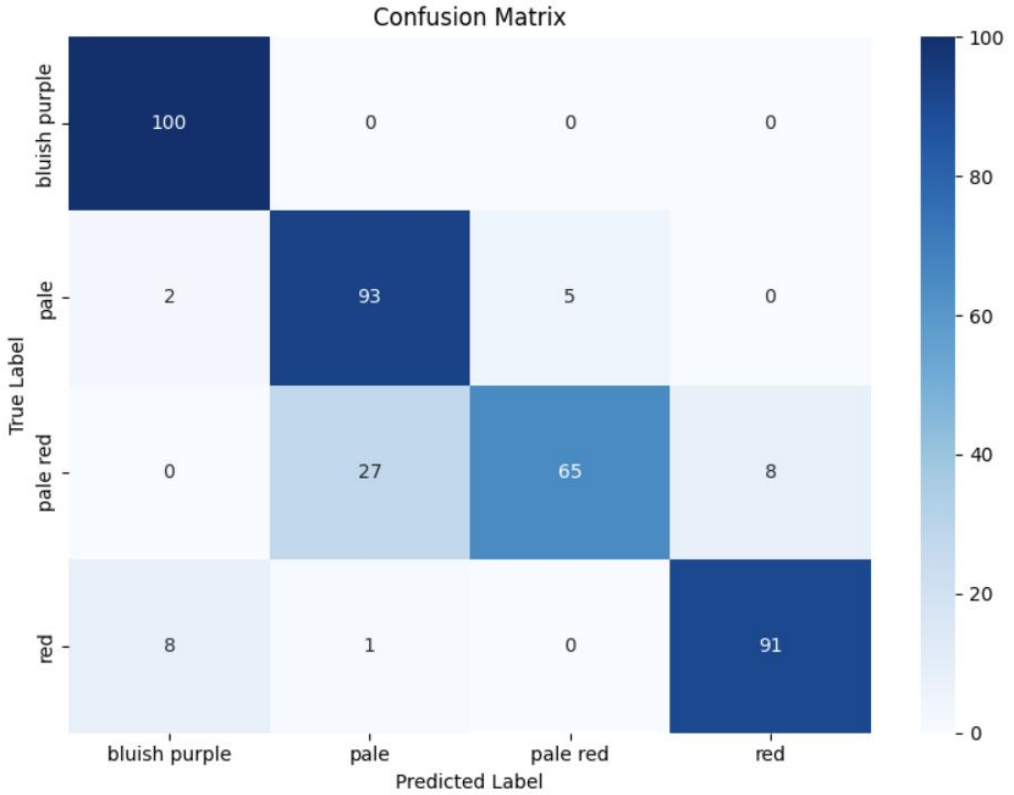


Figure 5.2.1.2: Confusion matrix of HybridNet on the 4-class dataset (SVM-segmented)

5.3 Impact of Segmentation using SVM or DuckNet on Classification

Note: To facilitate a direct comparison of segmentation methodologies, the classification results for the DuckNet-segmented dataset are presented here first alongside those from the SVM-segmented data.
(A full discussion of DuckNet's segmentation performance is reserved for the next chapter.)

ResNet20

Method	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Training time
SVM	87.50	87.57	87.50	87.31	20m 08s
DuckNet	86.25	86.16	86.25	85.92	22m 14s

Table 5.3.1: Comparison of ResNet20 performance on SVM vs. DuckNet-segmented 4-class data

HybridNet

Method	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Training time
SVM	87.25	90.91	100.00	95.24	01h 24m 23s
DuckNet	86.75	86.89	86.75	86.39	01h 23m 46s

Table 5.3.2: Comparison of HybridNet performance on SVM vs. DuckNet-segmented 4-class data

Key Finding: Segmentation Accuracy ≠ Classification Accuracy

Contrary to the principle that segmentation quality should influence classification performance, our results show a weak correlation between the two. Specifically, models trained on data from the weaker SVM segmentator achieved similar—and in some cases marginally higher—accuracy than those trained on DuckNet-segmented data.
This suggests that the classification networks are robust to the precise boundaries of the segmentation mask. The primary requirement is the successful isolation of the tongue region from the background. Once this is achieved, the model's capacity to learn discriminative features from the interior of the region appears to be the dominant factor in final performance. The minor performance differences observed (1-2%) fall within the expected variance of deep

learning training cycles (e.g., from weight initialization or data shuffling) and cannot be definitively attributed to the segmentation method.

Scope Note: This finding is based on a comparison of ResNet20 and HybridNet on the 4-class dataset.

5.4 Classification Performance on DuckNet-Segmented Datasets (Main Experiments)

5.4.1 Binary (2-Class) Dataset Results

5.4.1.1 AdderNet

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Stained moss	98.00	93.78	98.00	95.84
Non stained moss	95.30	97.91	93.50	95.65
Overall	**95.75**	**95.84**	**95.75**	**95.75**

Table 5.4.1.1: Classification performance of AdderNet on the 2-class dataset

Training time: 03h 51m 14s

Total parameters: 273.876

Model size: 1.09 MB

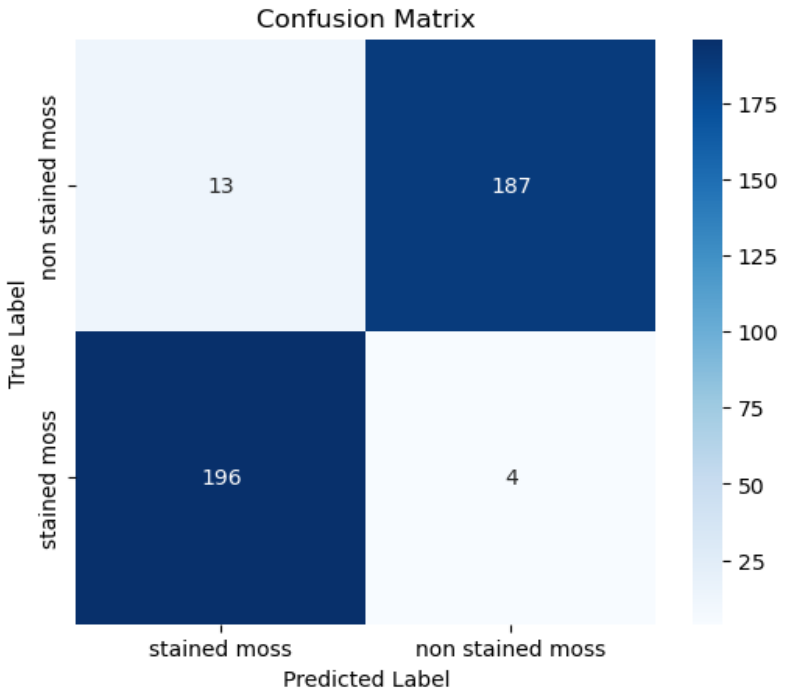


Figure 5.4.1.1: Confusion matrix of AdderNet on the 2-class dataset

5.4.1.2 ResNet20

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Stained moss	98.00	92.89	98.00	95.38
Non stained moss	92.50	97.88	92.50	95.12
Overall	**95.25**	**95.39**	**95.25**	**95.25**

Table 5.4.1.2: Classification performance of ResNet20 on the 2-class dataset

Training time: 24m 39s

Total parameters: 273,876

Model size: 1.09 MB

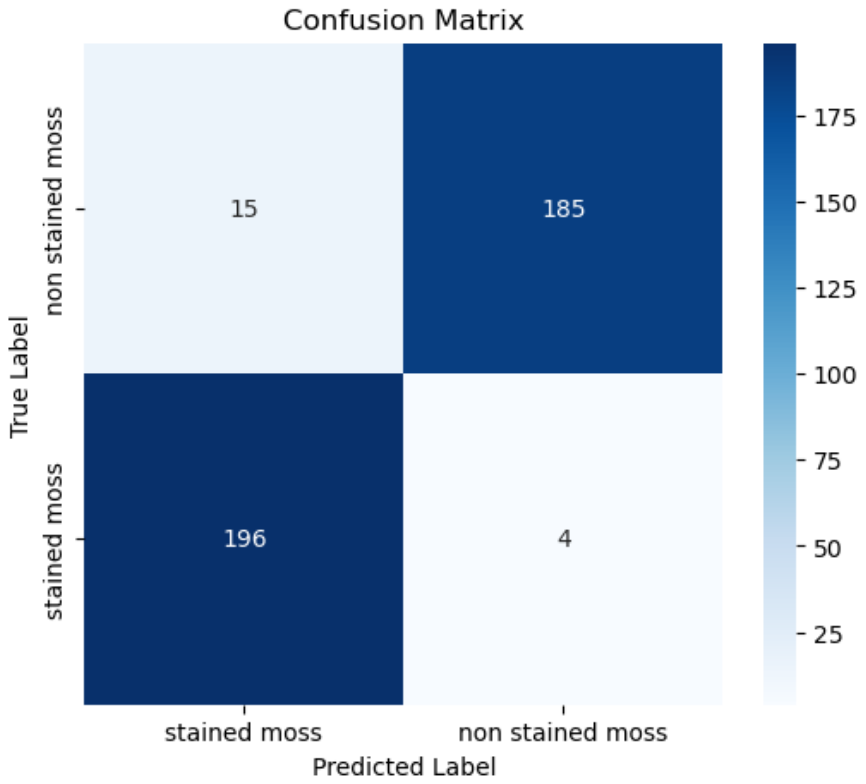


Figure 5.4.1.2: Confusion matrix of ResNet20 on the 2-class dataset

5.4.1.3 HybridNet

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Stained moss	99.50	93.87	99.50	96.60
Non stained moss	93.50	99.47	93.50	96.39
Overall	**96.50**	**96.67**	**96.50**	**96.50**

Table 5.4.1.3: Classification performance of HybridNet on the 2-class dataset

Training time: 01h 26m 07s

Total parameters: 228,610

Model size: 0.95 MB

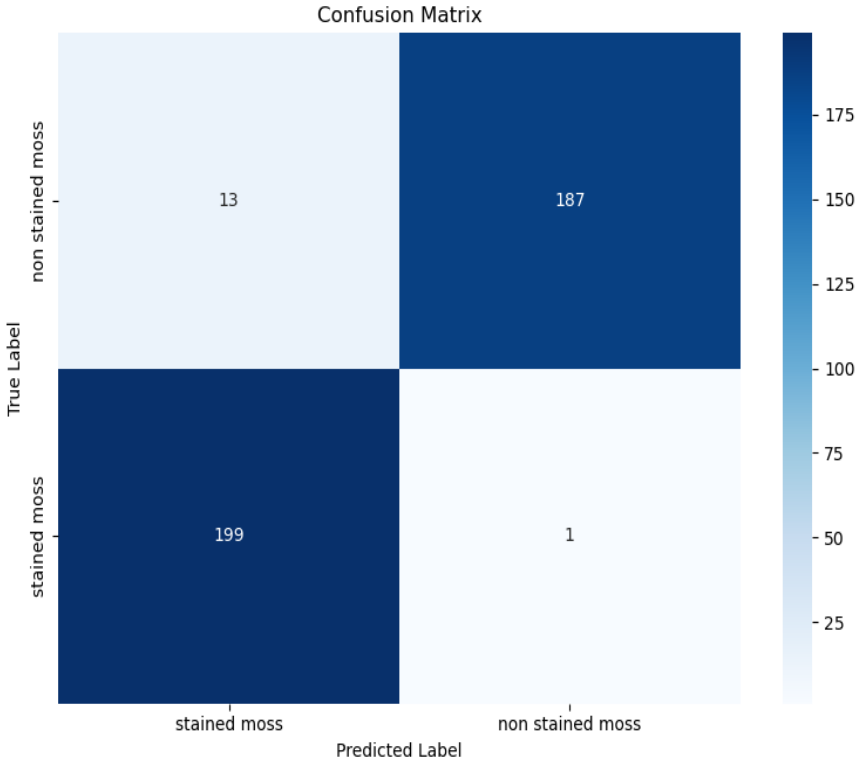


Figure 5.4.1.3: Confusion matrix of HybridNet on the 2-class dataset

5.4.1.4 Improved HybridNet

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Stained moss	99.50	95.22	99.50	97.31
Non stained moss	95.00	99.48	95.00	97.19
Overall	**97.25**	**97.35**	**97.25**	**97.25**

Table 5.4.1.4: Classification performance of Improved HybridNet on the 2-class dataset

Training time: 21m 02s

Total parameters: 130,922

Model size: 0.57 MB

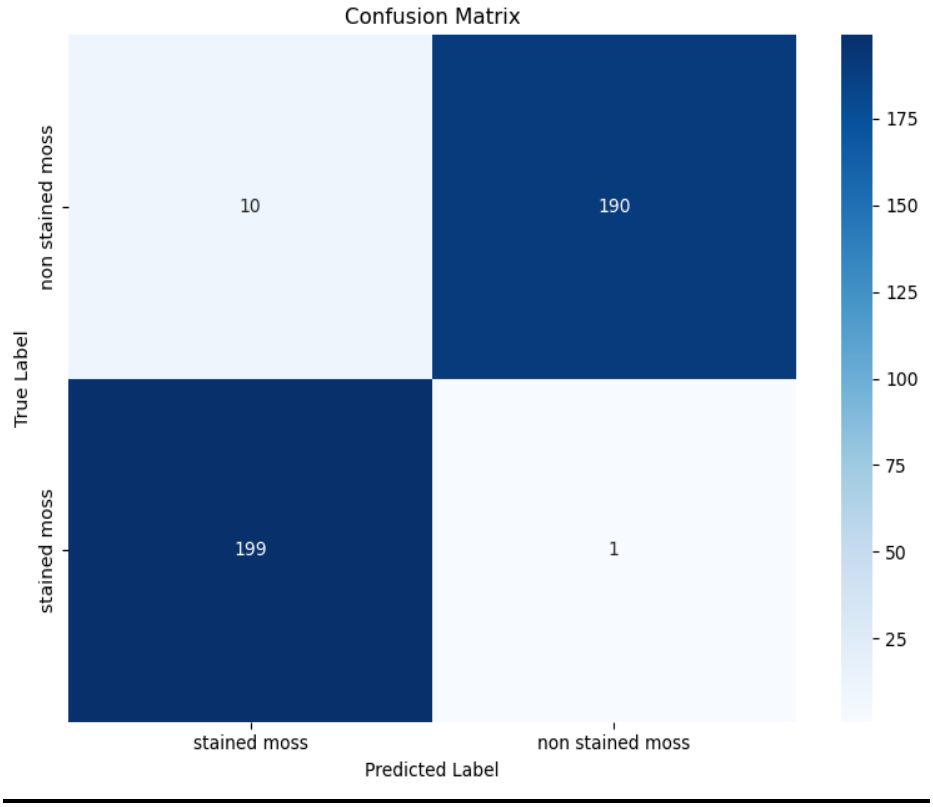


Figure 5.4.1.4: Confusion matrix of Improved HybridNet on the 2-class dataset

5.4.1.5 Model Performance on 2-Class Dataset (Stained moss vs. Non-stained moss)

Method	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Training time	Total parameters	Model size
AdderNet	95.75	95.84	95.75	95.75	03h 51m 14s	273.876	1.09 MB
ResNet20	95.25	95.39	95.25	95.25	24m 39s	273,876	1.09 MB
HybridNet	96.50	96.67	96.50	96.50	01h 26m 07s	228,610	0.95 MB
Improved HybridNet	**97.25**	**97.35**	**97.25**	**97.25**	**21m 02s**	**130,922**	**0.57 MB**

Table 5.4.1.5: Summary comparison of all models on the 2-class dataset

- **Observation:** All models performed strongly (>95% accuracy).
- **AdderNet and ResNet20:** Good performance, but relatively heavier in parameters.
- **HybridNet:** Slight improvement (~96.5%).
- **Improved HybridNet:** Best balance with 97.25% accuracy, reduced parameters (~131K vs. 230K), and smaller model size.
- **Discussion point:** Binary classification is inherently simpler, and all models can handle it well. The improved HybridNet is the most efficient choice, showing the benefit of architectural refinements.

5.4.2 Multi (4-Class) Dataset Results

5.4.2.1 AdderNet

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Pale	88.00	87.13	88.00	87.56
Pale red	86.00	87.76	86.00	86.87
Red	93.00	93.00	93.00	93.00
Bluish purple	97.00	96.04	97.00	96.52
Overall	**91.00**	**90.98**	**91.00**	**90.99**

Table 5.4.2.1: Classification performance of AdderNet on the 4-class dataset

Training time: 03h 51m 06s

Total parameters: 274,008

Model size: 1.09 MB

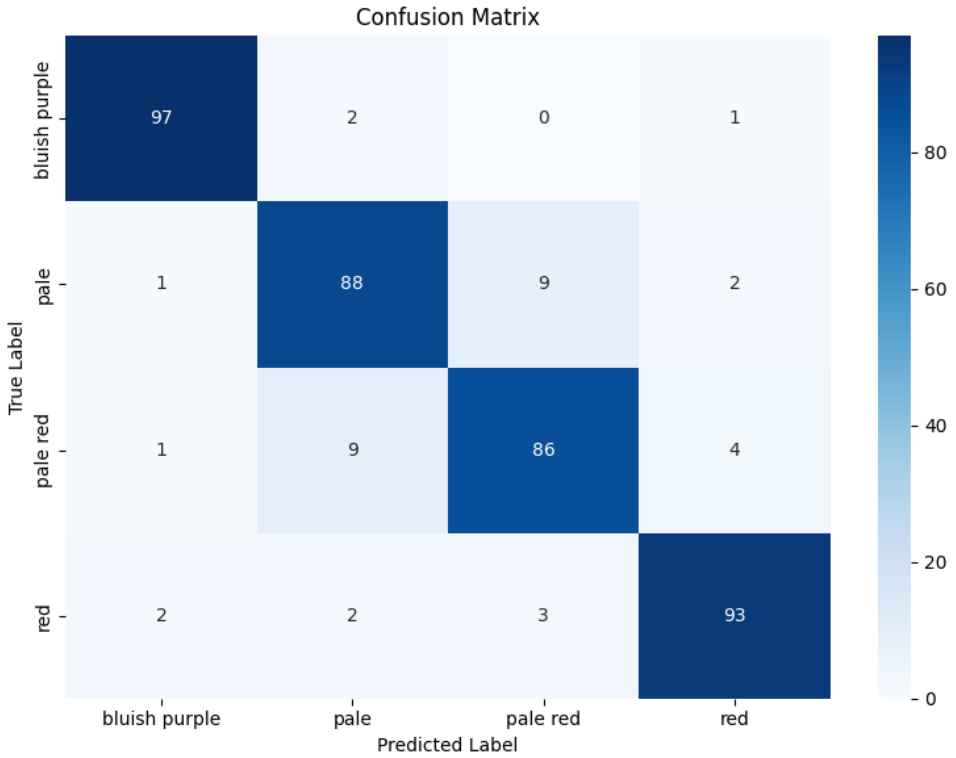


Figure 5.4.2.1: Confusion matrix of AdderNet on the 4-class dataset

5.4.2.2 ResNet20

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Pale	87.00	83.65	87.00	85.29
Pale red	68.00	83.95	68.00	75.14
Red	93.00	83.78	93.00	88.15
Bluish purple	97.00	93.27	97.00	95.10
Overall	**86.25**	**86.16**	**86.25**	**85.92**

Table 5.4.2.2: Classification performance of ResNet20 on the 4-class dataset

Training time: 22m 14s

Total parameters: 274,008

Model size: 1.09 MB

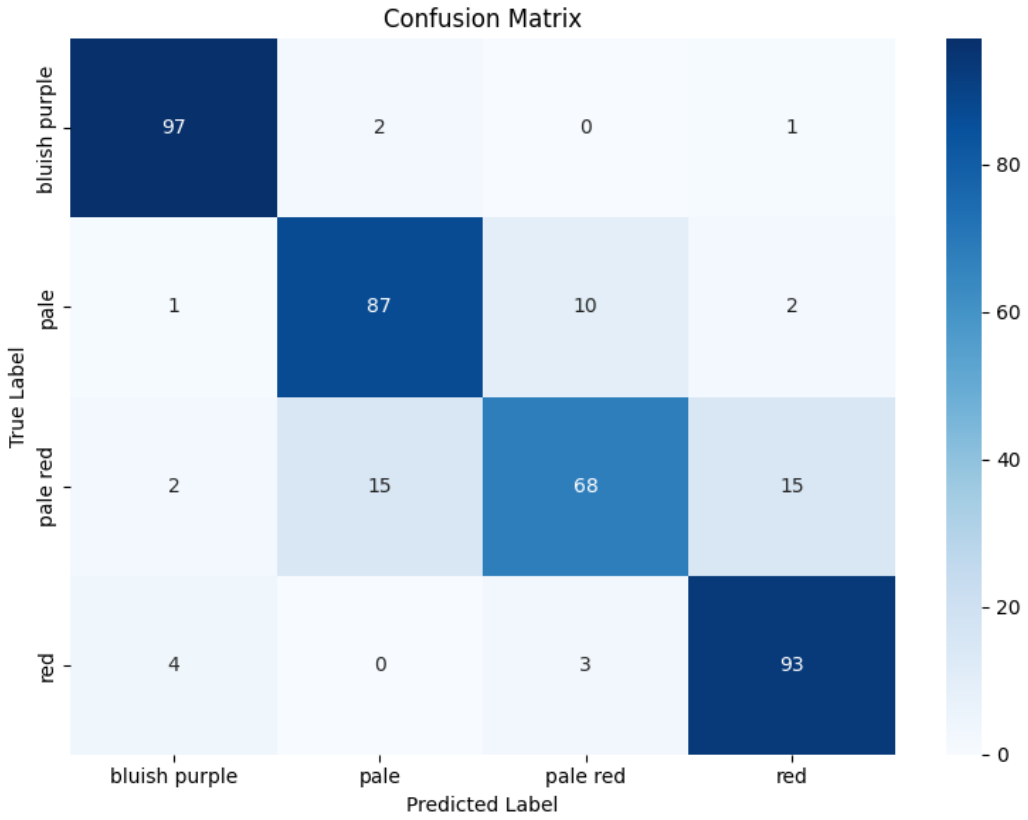


Figure 5.4.2.2: Confusion matrix of ResNet20 on the 4-class dataset

5.4.2.3 HybridNet

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Pale	89.00	83.18	89.00	85.99
Pale red	69.00	87.34	69.00	77.09
Red	89.00	90.82	89.00	89.90
Bluish purple	100.00	86.21	100.00	92.59
Overall	**86.75**	**86.89**	**86.75**	**86.39**

Table 5.4.2.3: Classification performance of HybridNet on the 4-class dataset

Training time: 01h 23m 46s

Total parameters: 228,804

Model size: 0.95 MB

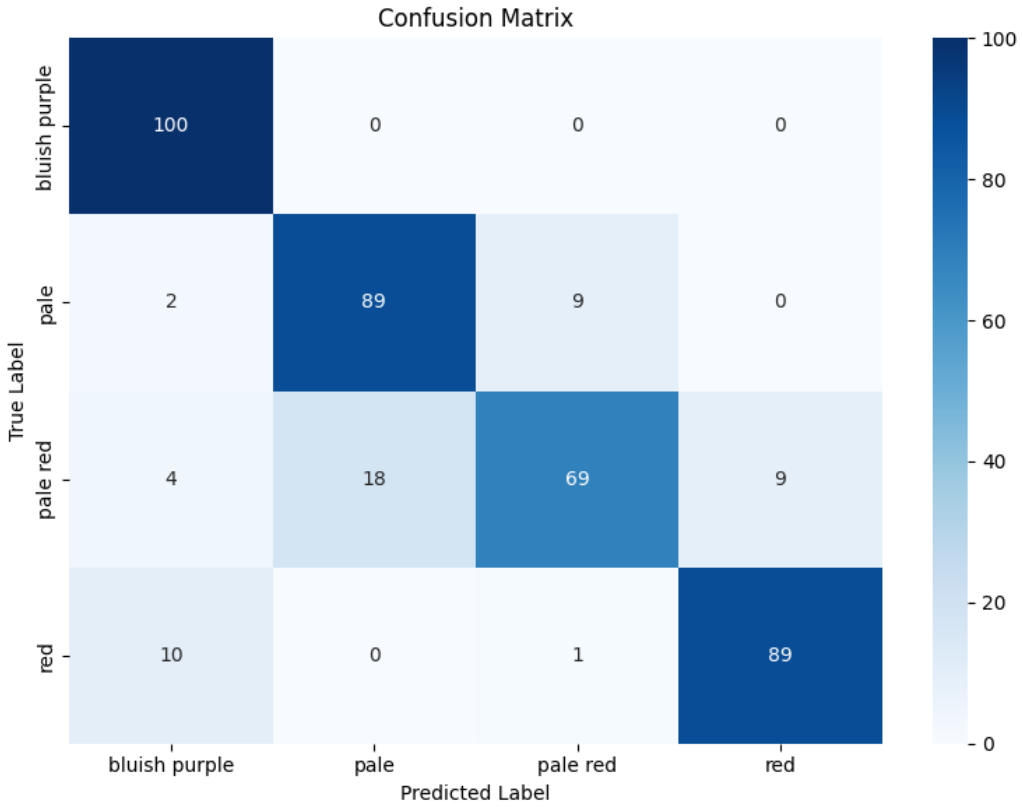


Figure 5.4.2.3: Confusion matrix of HybridNet on the 4-class dataset

5.4.2.4 Improved HybridNet

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Pale	84.00	84.85	84.00	84.42
Pale red	73.00	83.91	73.00	78.07
Red	96.00	89.72	96.00	92.75
Bluish purple	98.00	91.59	98.00	94.69
Overall	**87.75**	**87.52**	**87.75**	**87.48**

Table 5.4.2.4: Classification performance of Improved HybridNet on the 4-class dataset

Training time: 20m 09s

Total parameters: 131,116

Model size: 0.57 MB

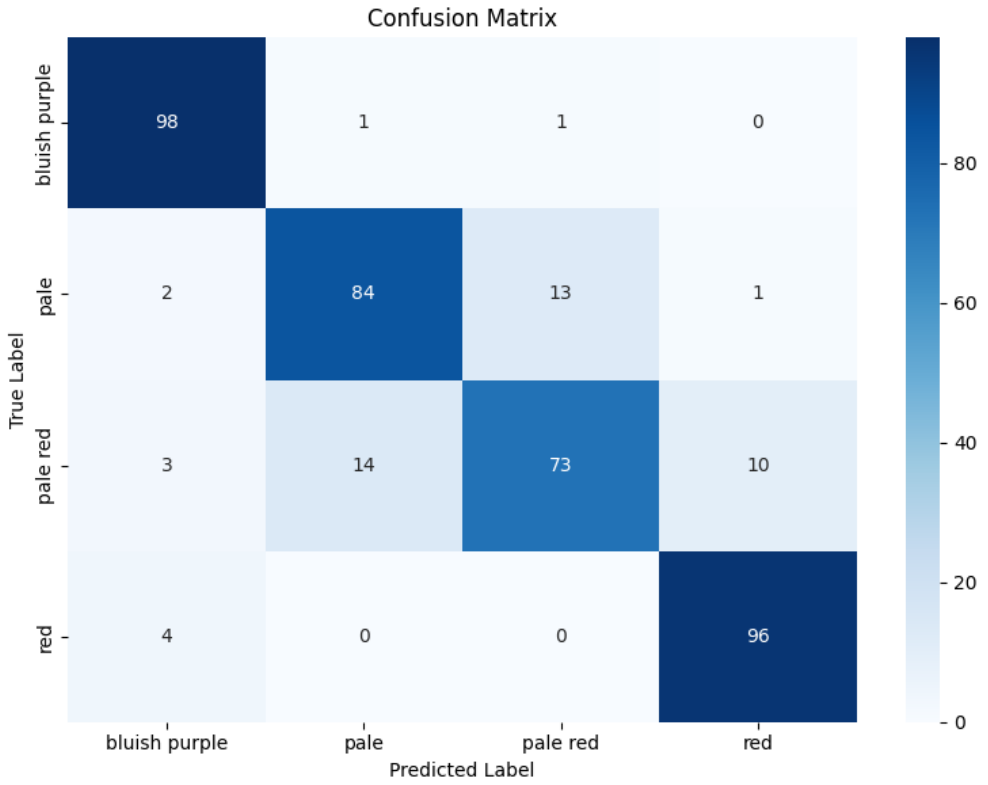


Figure 5.4.2.4: Confusion matrix of Improved HybridNet on the 4-class dataset

5.4.2.5 Model Performance on 4-Class Dataset (Pale, Pale Red, Red, Bluish Purple)

Method	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Training time	Total parameters	Model size
AdderNet	**91.00**	**90.98**	**91.00**	**90.99**	03h 51m 06s	274,008	1.09 MB
ResNet20	86.25	86.16	86.25	85.92	22m 14s	274,008	1.09 MB
HybridNet	86.75	86.89	86.75	86.39	01h 23m 46s	228,804	0.95 MB
Improved HybridNet	87.75	87.52	87.75	87.48	**20m 09s**	**131,116**	**0.57 MB**

Table 5.4.2.5: Summary comparison of all models on the 4-class dataset

- **Observation:** Accuracy drops (~86–91%) compared to binary classification due to greater complexity and class similarity.
- **AdderNet:** Highest accuracy (91%), but too large and inefficient.
- **ResNet20 and HybridNet:** Moderate performance (~86%), strong for some classes (Red, Bluish Purple), weak for Pale Red.
- **Improved HybridNet:** Balanced results, slightly better generalization, and much smaller model size (0.57 MB).
- **Discussion point:** The pale red class remains the hardest to classify, showing class overlap is a bigger limitation than model architecture. Efficiency vs. accuracy trade-off favors the improved HybridNet.

5.4.3 Multi (5-Class) Dataset Results

AdderNet was excluded from the 5-class experiments due to its excessive computational requirements, which caused training crashes. It was only feasible for the 2-class and, at most, the 4-class datasets.

5.4.3.1 ResNet20

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Mirror-Approximated	98.00	75.97	98.00	85.59
White-Greasy	63.00	86.30	63.00	72.83
Thin-White	69.00	82.14	69.00	75.00
Yellow-Greasy	99.00	86.84	99.00	95.52
Grey-Black	100.00	100.00	100.00	100.00
Overall	**85.80**	**86.25**	**85.80**	**85.19**

Table 5.4.3.1: Classification performance of ResNet20 on the 5-class dataset

Training time: 26m 45s

Total parameters: 274,074

Model size: 1.09 MB

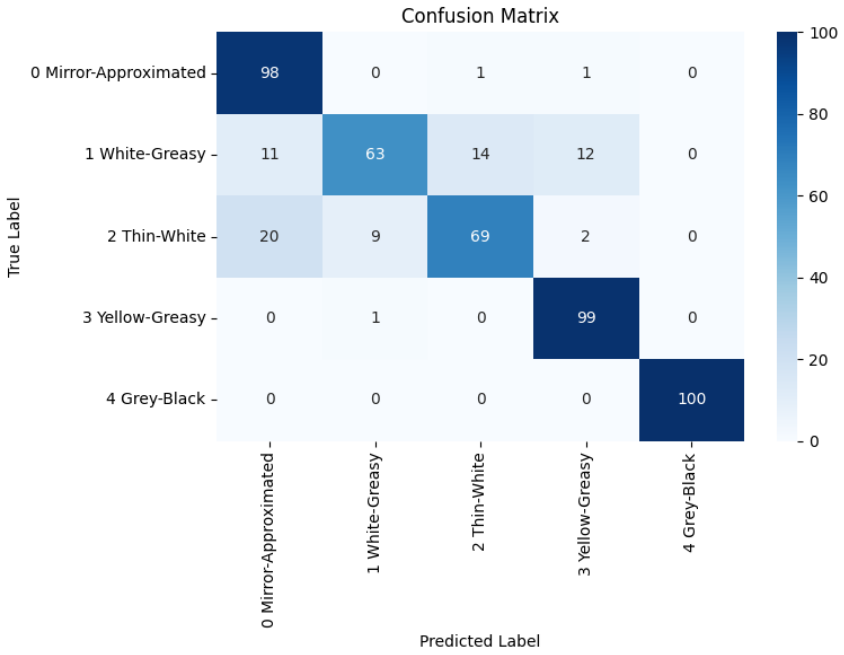


Figure 5.4.3.1: Confusion matrix of ResNet20 on the 5-class dataset

5.4.3.2 HybridNet

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Mirror-Approximated	98.00	73.13	98.00	83.76
White-Greasy	59.00	84.29	59.00	69.41
Thin-White	64.00	78.05	64.00	70.33
Yellow-Greasy	100.00	89.29	100.00	94.34
Grey-Black	100.00	98.04	100.00	99.01
Overall	**84.20**	**84.56**	**84.20**	**83.37**

Table 5.4.3.2: Classification performance of HybridNet on the 5-class dataset

Training time: 01h 53m 06s

Total parameters: 228,901

Model size: 0.95 MB

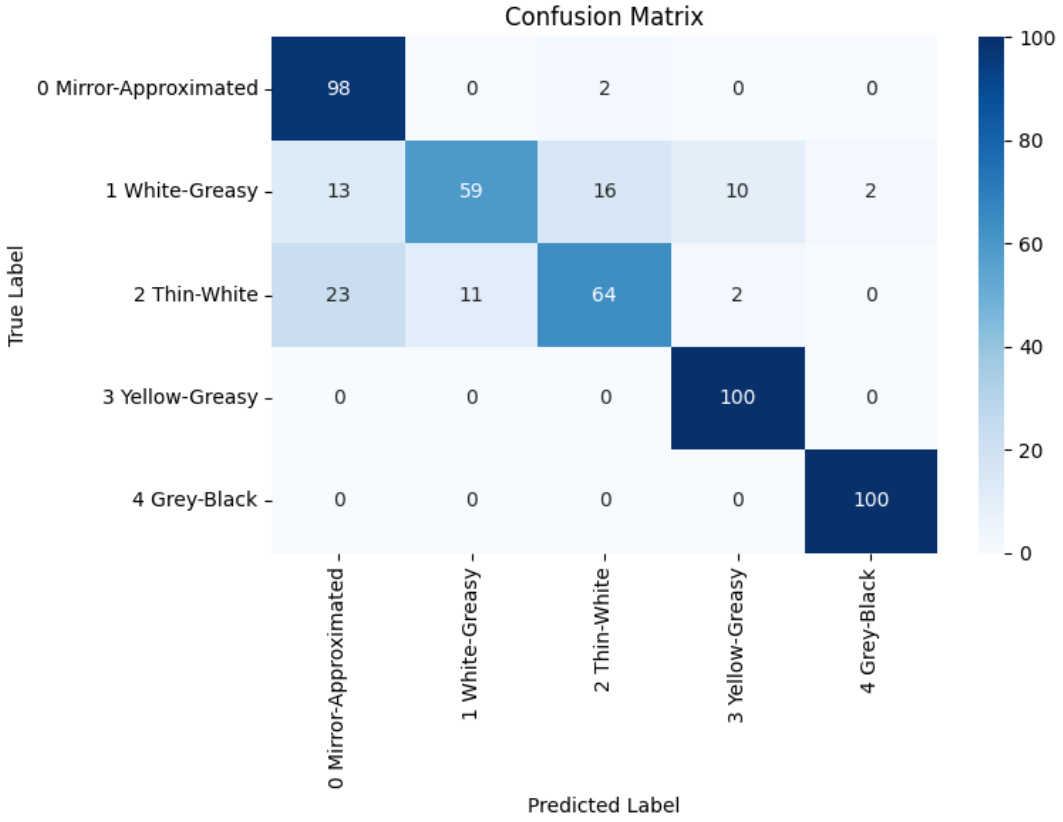


Figure 5.4.3.2: Confusion matrix of HybridNet on the 5-class dataset

5.4.3.3 Improved HybridNet

Class	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)
Mirror-Approximated	100.00	72.46	100.00	84.03
White-Greasy	59.00	88.06	59.00	70.66
Thin-White	69.00	84.15	69.00	75.82
Yellow-Greasy	100.00	90.91	100.00	95.24
Grey-Black	100.00	97.09	100.00	98.52
Overall	**85.60**	**86.53**	**85.60**	**84.86**

Table 5.4.3.3: Classification performance of Improved HybridNet on the 5-class dataset

Training time: 25m 07s

Total parameters: 131.213

Model size: 0.57 MB

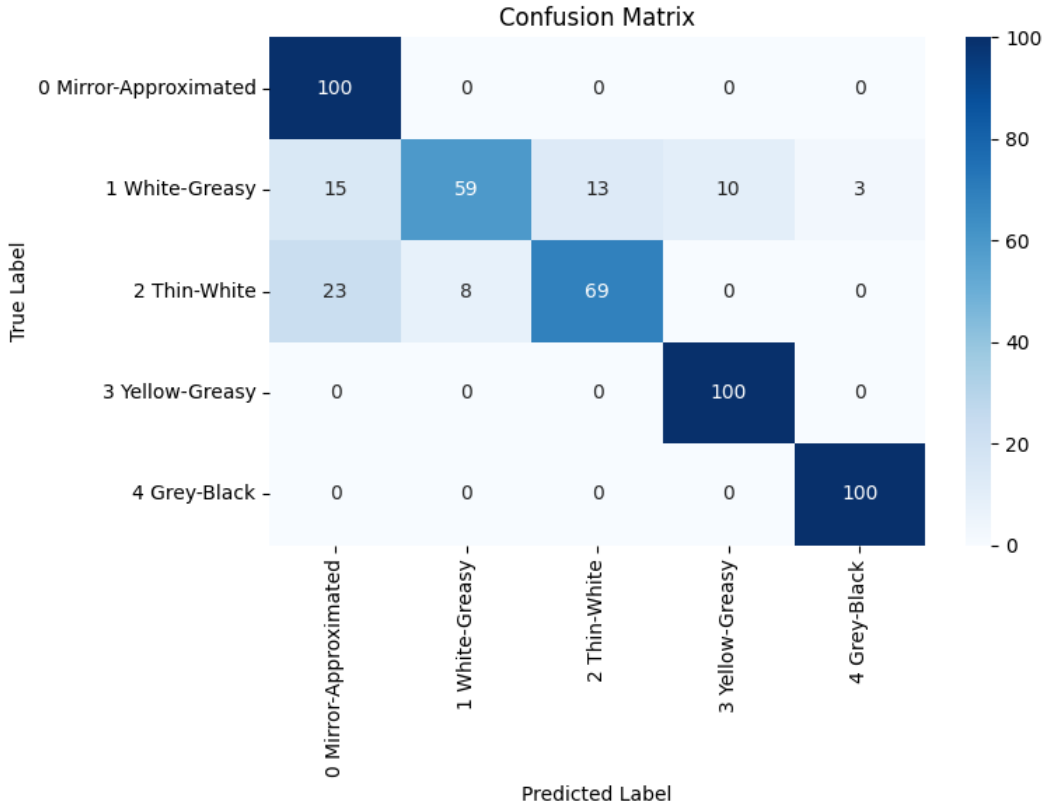


Figure 5.4.3.3: Confusion matrix of Improved HybridNet on the 5-class dataset

5.4.3.4 Model Performance on 5-Class Dataset (Mirror-Approximated, White-Greasy, Thin-White, Yellow-Greasy, Grey-Black)

Method	Accuracy (%)	Precision (%)	Recall (%)	F1 Score (%)	Training time	Total parameters	Model size
AdderNet	N/A	N/A	N/A	N/A	N/A	N/A	N/A
ResNet20	85.80	86.25	**85.80**	**85.19**	26m 45s	274,074	1.09 MB
HybridNet	84.20	84.56	84.20	83.37	01h 53m 06s	228,901	0.95 MB
Improved HybridNet	**85.60**	**86.53**	85.60	84.86	**25m 07s**	**131.213**	**0.57 MB**

Table 5.4.3.4: Summary comparison of all models on the 5-class dataset

- **Observation:** Accuracy lower (~84–86%), reflecting dataset difficulty.
- **AdderNet excluded** (crashes).
- **ResNet20:** Strong on Grey-Black and Yellow-Greasy (perfect accuracy), weaker on White-Greasy.
- **HybridNet:** Similar to ResNet20, slightly weaker overall.
- **Improved HybridNet:** Slightly higher overall (~85.6%), while being most efficient in size and parameters.
- **Discussion point:** White-Greasy remains a major challenge (low recall), highlighting dataset imbalance or visual ambiguity. Improved HybridNet again offers the best trade-off.

5.5 Overall Accuracy Summary and Trends

Overall Accuracy Summary Table

Dataset	AdderNet	ResNet20	HybridNet	Improved HybridNet
2-class	95.75%	95.25%	96.50%	**97.25%**
4-class	**91.00%**	86.25%	86.75%	87.75%
5-class	N/A	85.80%	84.20%	**85.60%**

Table 5.5.1: Overall accuracy summary of all models across 2-, 4-, and 5-class datasets

This summary table condenses the performance of all classification models across the three datasets into a single view. A few clear trends emerge:

- For the **2-class dataset**, all models performed very strongly, but the improved HybridNet achieved the highest accuracy (97.25%) while also being the most efficient in size and parameters. This can be explained by the relative simplicity of the binary classification task: the decision boundary between stained and non-stained moss is straightforward, so HybridNet's lightweight design is sufficient and even advantageous, avoiding overfitting and ensuring strong generalization.
- In the **4-class dataset**, AdderNet achieved the best overall accuracy (91%), but at the cost of much larger computational requirements. Here, the task is more complex because the model must discriminate subtle differences between tongue colors such as Pale, Pale Red, Red, and Bluish Purple. In this case, AdderNet's higher representational capacity allowed it to capture finer details, giving it an edge. However, the improved HybridNet, though slightly less accurate (87.75%), still provided a better trade-off by balancing efficiency with acceptable accuracy.
- The **5-class dataset** was the most challenging, with accuracies stabilizing around 84–86%. AdderNet could not be used here due to resource limitations, but both ResNet20 and HybridNet produced reasonable results, while the improved HybridNet achieved the highest accuracy (85.6%).

These results confirm two key insights: first, classification becomes progressively more difficult as the number of classes increases; and second, while AdderNet occasionally tops accuracy, the improved HybridNet consistently provides the best trade-off between performance and efficiency, making it the most suitable candidate for practical deployment.

Extra Class-Level Detailed Performance Analysis

To better understand *why* performance drops with more classes, the class-level metrics (accuracy, precision, recall, and F1) were examined in detail. Here, clear weaknesses appear in specific categories:

- **Pale (4-class):** All metrics were weaker compared to other classes, showing that this category is broadly difficult to distinguish.
- **Pale Red (4-class):** One of the most problematic classes. Accuracy, precision, recall, and F1 were all low, with recall and accuracy particularly poor, meaning the model both missed many true Pale Red cases and misclassified them frequently.
- **Thin-White (5-class):** Performed poorly across the board, especially in accuracy and recall, followed by F1, reflecting confusion with visually similar classes.
- **White-Greasy (5-class):** Another consistently weak class. All four metrics were low, with recall and accuracy standing out as particularly weak, indicating that the model struggled both to detect and to correctly classify this tongue condition.

In contrast, visually distinct classes — such as **Red and Bluish Purple (4-class)** and **Mirror-Approximated, Grey-Black and Yellow-Greasy (5-class)** — achieved very high or near-perfect scores across all metrics. This reinforces that the observed decline in dataset-level accuracy is driven primarily by the presence of **ambiguous, overlapping classes**, not by overall model limitations.

5.6 Computational Efficiency and Model Architecture Analysis

Model	Parameters	Model Size	Training Time (approx)	Best Dataset Performance
AdderNet	~274K	1.09 MB	~3h 51m	91% (4-class)
ResNet20	~274K	1.09 MB	~20–26 min	95.25% (2-class)
HybridNet	~228K	0.95 MB	~1h 23–93 min	96.50% (2-class)
Improved HybridNet	**~131K**	**0.57 MB**	**~20–25 min**	97.25% (2-class)

Table 5.6.1: Computational efficiency comparison of all models

AdderNet (ResNet20 backbone with adder2d)

AdderNet retains the **ResNet20 residual topology** (3-3-3 blocks) but replaces convolution multiplications with **L1 distance accumulation via adder2d layers**. The stem remains a standard Conv2d, so the **parameter count is almost identical to ResNet20 (~274K)**. However, because adder operations are **not cuDNN-optimized**, the model suffers from **much longer wall-clock training times (~3h 51m)** despite having the same size (1.09 MB). This extra representational capacity helps in **harder multi-class problems** (91% accuracy on 4-class, the highest among models), but it does not outperform leaner CNNs on simpler binary tasks. In short, AdderNet demonstrates that **capacity ≠ efficiency**: strong accuracy on complex datasets comes at a steep cost in runtime.

ResNet20 (vanilla convs + residual BasicBlocks)

ResNet20 uses the **standard Conv2d → BN → ReLU structure with identity/projection shortcuts**, making it **highly optimized by cuDNN**. With ~274K parameters and **model size of 1.09 MB**, it trains in **only 20–26 minutes** — the fastest of all models. Accuracy is **strong in binary classification (95.25%)**, showing that a straightforward convolutional backbone generalizes well for simple tasks. However, its performance dips in fine-grained multi-class settings compared to AdderNet, confirming that **efficiency in training time does not always translate into the best performance in more complex datasets**.

HybridNet (baseline, with MobileNetV2 inverted residuals)

HybridNet adopts the **MobileNetV2 design**: **expand (1×1) → depthwise (3×3) → pointwise (1×1)** with residuals when dimensions match. This use of **depthwise separable convolutions drastically reduces FLOPs and parameters** compared to standard ResNet blocks, cutting the parameter count to **~228K** and model size to **0.95 MB**. Although the parameter count is lower, **training speed was slower (~1.5h) compared to ResNet20** because **depthwise operations are not as well optimized in GPU libraries as standard convolutions**, resulting in longer runtimes despite the smaller model size. Even so, HybridNet achieves **higher accuracy (96.5% on 2-class)** than ResNet20, showing how **efficient feature reuse boosts performance** without increasing size. However, it underperforms on subtle multi-class datasets where greater capacity is required.

HybridNet (improved: SE, DropPath, checkpointing, gentler ratios)

The improved HybridNet incorporates **Squeeze-and-Excite (SE)** for channel attention, **DropPath** for regularization, and reduces the **expand ratio from 6.0 to 3.0**, making it **smaller and faster**. It further saves compute by applying **projection shortcuts only when shapes differ** and using **dynamic downsampling** at high resolutions. During training, **activation checkpointing lowers memory usage**, enabling faster and more efficient fitting. Altogether, these refinements shrink the model to **~131K parameters and 0.57 MB**, with a **training time of ~20–25 minutes**, while still delivering the **highest accuracy overall (97.25% on 2-class)**. Even though it trails AdderNet slightly on the 4-class dataset, it **wins decisively on the efficiency trade-off**, offering the best balance of size, speed, and accuracy.

5.6.1 Best Classification Model Selction

Although different models showed strengths in different areas, not all were equally practical for real-world use. **AdderNet** achieved the highest accuracy in the 4-class dataset (91%), but its excessive training time (~3h 51m), larger computational demand, and inability to scale to the 5-class dataset limited its practicality. **ResNet20** offered the fastest training speed (20–26 minutes) and stability, but its performance dropped notably on multi-class datasets (86% on 4-class and 85.8% on 5-class). The baseline **HybridNet** improved efficiency (~228K parameters, 0.95 MB) and achieved higher accuracy in the 2-class dataset (96.5%), yet it still underperformed in the more challenging multi-class settings compared to AdderNet.

Ultimately, the **Improved HybridNet** emerged as the most suitable overall model. It achieved the best binary classification accuracy (97.25%), competitive performance in the 4-class (87.75%) and 5-class (85.6%) datasets, and delivered this with the **smallest parameter count (131K), lowest storage footprint (0.57 MB), and fastest training time (~20–25 minutes)**. These results establish the Improved HybridNet as the most balanced and practical candidate for real-world TCM tongue diagnosis applications, combining efficiency, scalability, and strong performance.

Chapter 6
Conclusion And Recommendation

6.1 Conclusion

This project delivered a reproducible end-to-end pipeline for tongue image analysis—segmentation followed by classification—evaluated on multi-class datasets with increasing visual complexity. DuckNet produced consistently reliable masks on challenging backgrounds and was therefore adopted as the default pre-processing stage. Notably, classification accuracy was largely insensitive to the exact mask quality: models trained on SVM-segmented data performed similarly to those trained on DuckNet masks, indicating that once the tongue region is reasonably isolated, discriminative cues inside the region dominate downstream performance.

Within classification, ResNet20 served as a strong and compact baseline that trained quickly and provided a stable reference point for architecture comparisons. The (non-improved) HybridNet combined ResNet-style skip connections with MobileNetV2 inverted residuals, reducing parameters while maintaining accuracy close to ResNet20, albeit sometimes with longer training despite its smaller footprint. The Improved HybridNet—augmenting HybridNet with squeeze-and-excite, DropPath, reduced expand ratios, and activation checkpointing—offered the best accuracy-efficiency balance and fastest practical training in our setting. AdderNet validated the feasibility of addition-based convolutions but incurred high training cost due to limited GPU optimization and is not preferred for deployment. Overall, the recommended stack for practical use is DuckNet segmentation followed by Improved HybridNet classification, with ResNet20 retained as the reference baseline and the original HybridNet as a lightweight alternative when sticking to conventional inverted-residual designs.

6.2 Recommendation

To build on the outcomes of this project, future work should focus on improving data quality, refining the models, and preparing the pipeline for real-world deployment.

First, the dataset needs to be expanded and better balanced. Visually similar classes should receive particular attention, as they are more prone to misclassification. Increasing the number of samples across all categories and ensuring even class distribution will help reduce bias and improve generalization. Reporting class-wise metrics should remain standard practice, as it helps expose specific failure modes that may be hidden by overall accuracy figures.

In terms of segmentation, DuckNet should remain the default model, especially given its consistent performance on complex backgrounds. While the SVM-based method offers an extremely small footprint and is easy to train on CPUs, it struggled with more visually complicated scenes in this study. However, SVM can still be a valuable baseline if enhanced. Future iterations could improve it by incorporating deep features from CNN encoders like those used in HybridNet or DuckNet, or by shifting to a superpixel-level approach using richer color, texture, and deep features. Simple post-processing steps—such as morphological operations, conditional random fields (CRFs), or graph-cut techniques—could also help sharpen segmentation boundaries. Additional improvements may come from systematic tuning of kernel types and hyperparameters (e.g., RBF or χ^2 kernels with optimized C and γ values), or from using DuckNet-generated masks as pseudo-labels in a semi- or self-supervised learning setup.

On the classification side, the Improved HybridNet remains the recommended model due to its excellent balance of accuracy, efficiency, and training speed. It delivers reliable performance across complex multi-class tasks and is well-suited for real-world deployment. However, it's important to note that while Improved HybridNet performs strongly overall, it still falls short of the peak accuracy achieved by AdderNet. That said, AdderNet's high computational cost and limited GPU optimization make it impractical for most deployment scenarios. To close this gap without compromising on efficiency, future work could explore enhancements to Improved HybridNet—such as refined loss functions (e.g., class-balanced or focal loss), stronger normalization techniques, or targeted augmentation strategies for difficult classes. For resource-constrained environments, quantization or knowledge distillation could further compress the model, and exporting to ONNX or TensorRT would improve runtime

performance. Coupling it with a lightweight segmentation model can also enable real-time applications.

Finally, before any clinical or production deployment, the full system should be tested in real-world scenarios with practitioners. This step is essential to assess calibration, usability, and actual utility in a clinical workflow. Feedback from end users will be invaluable for making practical adjustments and closing the gap between technical performance and real-world impact.

REFERENCES

[1] José María Rodríguez Corral, Javier Civit-Masot, Francisco Luna-Perejón, Ignacio Díaz-Cano, Arturo Morgado-Estévez, Manuel Domínguez-Morales, "Energy efficiency in Edge TPU vs. embedded GPU for computer-aided medical imaging segmentation and classification," *Engineering Applications of Artificial Intelligence,* vol. 127, 2024.

[2] Angela Renton, Thuy T. Dao, Tom Johnstone, Oren Civier, "Neurodesk: an accessible, flexible and portable data analysis environment for reproducible neuroimaging," *Nature Methods,* vol. 21, no. 5, 2024.

[3] Xuebo Jin, Longfei Gao, Anshuo Tong, Zhengyang Chen, Jianlei Kong, Ning Sun, Huijun Ma, Qiang Wang, Yuting Bai, Tingli Su, "TCM-Tongue: A Standardized Tongue Image Dataset with Pathological Annotations for AI-Assisted TCM Diagnosis," 2025.

[4] Merjem Bećirović, Amina Kurtović, Nordin Smajlović, Medina Kapo, Amila Akagić, "Performance comparison of medical image classification systems using TensorFlow Keras, PyTorch, and JAX," 2025.

[5] Qi Liu, Yan Li, Peng Yang, Quanquan Liu, Chunbao Wang, Keji Chen, Zhengzhi Wu, "A survey of artificial intelligence in tongue image for disease diagnosis and syndrome differentiation".

[6] Ali Raad Hassoon, Ali Al-Naji, Ghaidaa A. Khalid, Javaan Chahl, "Tongue Disease Prediction Based on Machine Learning Algorithms," *Technologies,* vol. 12, no. 7, 2024.

[7] Qianzi Che, Yuanming Leng, Wei Yang, Xihao Cao, Zhongxia Wang, Lizheng Liu, Feibiao Xie, Ruilin Wang, "Tongue Image–Based Diagnosis of Acute Respiratory Tract Infection Using Machine Learning: Algorithm Development and Validation," *JMIR Med Inform,* vol. 13, 2025.

[8] Ahmad Waleed Salehi, Shakir Khan, Gaurav Gupta, Bayan Ibrahimm Alabduallah, Abrar Almjally, Hadeel Alsolai, Tamanna Siddiqui, Adel Mellit, "A Study of CNN and Transfer Learning in Medical Imaging: Advantages, Challenges, Future Scope," *MDPI,* vol. 15, no. 7, 2023.

[9] P. Gayathri, "Exploring the Potential of VGG-16 Architecture for Accurate Brain Tumor Detection Using Deep Learning," *Journal of Computers,* 2023.

[10] Meng-Yi Li, Ding-Ju Zhu, Wen Xu, Yu-Jie Lin, Kai-Leung Yung, Andrew W. H. Ip, "Application of U-Net with Global Convolution Network Module in Computer-Aided Tongue Diagnosis," *Journal of Healthcare Engineering,* 2021.

[11] Summiya Taskin, Ferdib-Al-Islam, "Transfer Learning-based Fine Tuned MobileNetV2 Model with Explainable Artificial Intelligence for Identifying Dental Diseases," *IEEE,* 2025.

[12] H. Chen, "AdderNet: Do We Really Need Multiplications in Deep Learning?," 2020.

[13] Zhong li qin, Xin Guojiang, Peng Qinghua, Liu Wanghua, Wu Yingjie, Sheng Dan, Zhu Lei, Sui Qiang, Liang Hao, "A dataset of stained tongue fur images of TCM," [Online]. Available: https://www.scidb.cn/en/detail?dataSetId=223214839b224f798a40120dcec4576a.

[14] "Tongue Image Dataset-neo," 24 3 2023. [Online]. Available: https://aistudio.baidu.com/datasetdetail/196398.

[15] Muhammad151, "tongue images," [Online]. Available: https://www.kaggle.com/datasets/muhammad151/tongue-images.

[16] Towfiq_Tomal, "Tongue__DIABETES," [Online]. Available: https://www.kaggle.com/datasets/towfiqtomal/tongue-diabetes.

[17] Muhammad Saddam Zikri Dalimunthe, Rossy Nurhasanah, "Type 2 Diabetes Mellitus Tongue Dataset," Mendeley, 7 10 2024. [Online]. Available: https://data.mendeley.com/datasets/hyb44jf936/2.

[18] "Intelligent-tongue-diagnosis-detection-dataset," 2025. [Online]. Available: https://github.com/btbuIntelliSense/Intelligent-tongue-diagnosis-detection-dataset?tab=readme-ov-file.

APPENDIX

SVM

svm.ipynb

```
from __future__ import annotations

# --- Colab / I/O ---
from google.colab import drive

# --- Std / third-party ---
import os
import cv2
import joblib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from tqdm import tqdm
from typing import List, Tuple, Dict

from skimage.feature import local_binary_pattern
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.svm import LinearSVC
from sklearn.metrics import (
    accuracy_score,
    precision_score,
    recall_score,
    f1_score,
    jaccard_score,
    confusion_matrix,
)


# ============================================================
# 1) Feature extraction helpers
# ============================================================

def _fixed_window(img: np.ndarray, x: int, y: int, window_size: int = 7) ->
np.ndarray:
    k = window_size
    pad = k // 2
    img_pad = cv2.copyMakeBorder(img, pad, pad, pad, pad, cv2.BORDER_REFLECT)
    return img_pad[y : y + k, x : x + k]
```

```
def extract_features(img: np.ndarray, x: int, y: int, window_size: int = 7):
    k = window_size
    window = _fixed_window(img, x, y, window_size=k)

    r, g, b = img[y, x]
    mean_rgb = window.mean(axis=(0, 1))
    std_rgb = window.std(axis=(0, 1))

    # Texture: uniform LBP histograms + local Sobel stats
    gray = cv2.cvtColor(window, cv2.COLOR_RGB2GRAY)
    lbp1 = local_binary_pattern(gray, P=8, R=1, method="uniform")
    lbp2 = local_binary_pattern(gray, P=16, R=3, method="uniform")
    hist1 = np.histogram(lbp1.ravel(), bins=10, range=(0, 10))[0]
    hist2 = np.histogram(lbp2.ravel(), bins=18, range=(0, 18))[0]

    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0, ksize=3)
    sobely = cv2.Sobel(gray, cv2.CV_64F, 0, 1, ksize=3)

    # Position
    h, w = img.shape[:2]
    cx, cy = w / 2.0, h / 2.0
    dist_to_center = np.sqrt((x - cx) ** 2 + (y - cy) ** 2)
    dist_to_center /= np.sqrt(cx ** 2 + cy ** 2)

    features = [
        r,
        g,
        b,
        mean_rgb[0],
        mean_rgb[1],
        mean_rgb[2],
        std_rgb[0],
        std_rgb[1],
        std_rgb[2],
        *hist1,
        *hist2,
        sobelx.mean(),
        sobely.mean(),
        sobelx.std(),
        sobely.std(),
        x / w,
        y / h,
        dist_to_center,
    ]

    feature_names = [
        "Red",
        "Green",
```

```
        "Blue",
        "Mean_R",
        "Mean_G",
        "Mean_B",
        "Std_R",
        "Std_G",
        "Std_B",
        *[f"LBP1_bin_{i}" for i in range(10)],
        *[f"LBP2_bin_{i}" for i in range(18)],
        "SobelX_mean",
        "SobelY_mean",
        "SobelX_std",
        "SobelY_std",
        "X_pos",
        "Y_pos",
        "Dist_to_center",
    ]
    return np.array(features, dtype=np.float32), feature_names


# ============================================================
# 2) Dense, vectorized full-image features
# ============================================================

def _local_mean_std(img_f32: np.ndarray, k: int):
    mean = cv2.blur(img_f32, (k, k), borderType=cv2.BORDER_REFLECT)
    sqr = cv2.blur(img_f32 * img_f32, (k, k), borderType=cv2.BORDER_REFLECT)
    var = np.maximum(sqr - mean * mean, 0.0)
    std = np.sqrt(var)
    return mean, std


def _lbp_hist_per_pixel(gray_u8: np.ndarray, P: int, R: int, k: int) ->
np.ndarray:
    lbp = local_binary_pattern(gray_u8, P=P, R=R,
method="uniform").astype(np.float32)
    H, W = gray_u8.shape
    B = int(P + 2)
    out = np.empty((H, W, B), dtype=np.float32)
    edges = np.arange(B + 1, dtype=np.float32)
    area = float(k * k)
    for b in range(B):
        low, high = edges[b], edges[b + 1]
        mask = ((lbp >= low) & (lbp < high)).astype(np.float32)
        out[..., b] = cv2.blur(mask, (k, k), borderType=cv2.BORDER_REFLECT) *
area
    return out
```

```
def _build_full_image_features(img_rgb: np.ndarray, window_size: int = 7) ->
np.ndarray:
    k = window_size
    H, W, _ = img_rgb.shape
    img_f32 = img_rgb.astype(np.float32)

    def to_hwc(x: np.ndarray) -> np.ndarray:
        x = np.asarray(x)
        # Ensure HxWxC no matter what comes in (1D/2D/3D)
        if x.ndim == 1:
            x = x.reshape(H, W, 1)  # rare 1D edge cases from cv ops
        elif x.ndim == 2:
            x = x[..., None]
        # if already 3D, keep as-is
        return x.astype(np.float32)

    # RGB at pixel
    R = to_hwc(img_f32[..., 0])
    G = to_hwc(img_f32[..., 1])
    B = to_hwc(img_f32[..., 2])

    # local mean/std RGB (HxWx3 each)
    mean_rgb, std_rgb = _local_mean_std(img_f32, k)
    mean_rgb = mean_rgb.astype(np.float32)
    std_rgb = std_rgb.astype(np.float32)

    # Gray + LBP
    gray_f32 = cv2.cvtColor(img_rgb, cv2.COLOR_RGB2GRAY).astype(np.float32)
    gray_u8 = np.clip(gray_f32, 0, 255).astype(np.uint8)
    lbp1_hist = _lbp_hist_per_pixel(gray_u8, P=8, R=1, k=k)
    lbp2_hist = _lbp_hist_per_pixel(gray_u8, P=16, R=3, k=k)

    # Sobel (float); then local mean/std in k×k
    sobelx = cv2.Sobel(gray_f32, cv2.CV_32F, 1, 0, ksize=3)
    sobely = cv2.Sobel(gray_f32, cv2.CV_32F, 0, 1, ksize=3)
    sx_mean, sx_std = _local_mean_std(sobelx[..., None], k)
    sy_mean, sy_std = _local_mean_std(sobely[..., None], k)
    # keep channels; robustly force HxWx1 shape
    sx_mean, sx_std = to_hwc(sx_mean), to_hwc(sx_std)
    sy_mean, sy_std = to_hwc(sy_mean), to_hwc(sy_std)

    # Position features
    xs = (np.arange(W, dtype=np.float32)[None, :] / W).repeat(H, axis=0)
    ys = (np.arange(H, dtype=np.float32)[:, None] / H).repeat(W, axis=1)
    cx, cy = W / 2.0, H / 2.0
    dist = np.sqrt((np.arange(W)[None, :] - cx) ** 2 + (np.arange(H)[:, None]
- cy) ** 2).astype(np.float32)
    dist /= np.sqrt(cx * cx + cy * cy)
```

```
    xs, ys, dist = to_hwc(xs), to_hwc(ys), to_hwc(dist)

    feats = np.concatenate(
        [R, G, B, mean_rgb, std_rgb, lbp1_hist, lbp2_hist, sx_mean, sy_mean,
sx_std, sy_std, xs, ys, dist],
        axis=2,
    ).astype(np.float32)
    return feats.reshape(-1, feats.shape[2])


# ============================================================
# 3) Balanced per-image pixel sampling (training/testing)
# ============================================================

def _sample_balanced_from_image(
    img: np.ndarray,
    mask_gray: np.ndarray,
    fg_target: int = 4000,
    bg_target: int = 4000,
    rng: np.random.Generator | None = None,
    window_size: int = 7,
) -> Tuple[np.ndarray, np.ndarray]:

    if rng is None:
        rng = np.random.default_rng(42)

    fg_y, fg_x = np.where(mask_gray != 0)
    bg_y, bg_x = np.where(mask_gray == 0)

    if len(fg_x) == 0 or len(bg_x) == 0:
        return np.empty((0, 44), dtype=np.float32), np.empty((0,),
dtype=np.uint8)

    fg_take = min(fg_target, len(fg_x))
    bg_take = min(bg_target, len(bg_x))
    fg_idx = rng.choice(len(fg_x), size=fg_take, replace=False)
    bg_idx = rng.choice(len(bg_x), size=bg_take, replace=False)

    xs = np.concatenate([fg_x[fg_idx], bg_x[bg_idx]])
    ys = np.concatenate([fg_y[fg_idx], bg_y[bg_idx]])
    labels = np.concatenate([np.ones(fg_take, dtype=np.uint8),
np.zeros(bg_take, dtype=np.uint8)])

    feats: List[np.ndarray] = []
    for x, y in zip(xs, ys):
        f, _ = extract_features(img, int(x), int(y), window_size=window_size)
        feats.append(f)
```

```
    return np.asarray(feats, dtype=np.float32), labels


def load_and_split_data(
    image_folder: str,
    mask_folder: str,
    n_images: int = 500,
    test_size: float = 0.2,
    fg_per_img: int = 4000,
    bg_per_img: int = 4000,
) -> Tuple[np.ndarray, np.ndarray, np.ndarray, np.ndarray, np.ndarray, List, List]:
    """Load images, split by index, and sample a balanced set of pixels per image."""
    all_indices = np.arange(1, n_images + 1)
    train_indices, test_indices = train_test_split(all_indices, test_size=test_size, random_state=42)
    rng = np.random.default_rng(123)

    print("Processing training images (balanced sampling)...")
    X_train, y_train, train_images = [], [], []
    for i in tqdm(train_indices):
        img = cv2.cvtColor(cv2.imread(f"{image_folder}/{i}.png"), cv2.COLOR_BGR2RGB)
        mask = cv2.imread(f"{mask_folder}/{i}.png", cv2.IMREAD_GRAYSCALE)
        train_images.append((i, img, mask))
        Xi, yi = _sample_balanced_from_image(img, mask, fg_target=fg_per_img, bg_target=bg_per_img, rng=rng)
        if Xi.size:
            X_train.append(Xi)
            y_train.append(yi)
    X_train = np.vstack(X_train)
    y_train = np.concatenate(y_train)

    print("\nProcessing test images (balanced sampling for the *sampled* report)...")
    X_test, y_test, test_images = [], [], []
    for i in tqdm(test_indices):
        img = cv2.cvtColor(cv2.imread(f"{image_folder}/{i}.png"), cv2.COLOR_BGR2RGB)
        mask = cv2.imread(f"{mask_folder}/{i}.png", cv2.IMREAD_GRAYSCALE)
        test_images.append((i, img, mask))
        Xi, yi = _sample_balanced_from_image(img, mask, fg_target=fg_per_img // 2, bg_target=bg_per_img // 2, rng=rng)
        if Xi.size:
            X_test.append(Xi)
            y_test.append(yi)
    X_test = np.vstack(X_test)
```

```
    y_test = np.concatenate(y_test)

    return (
        X_train,
        y_train,
        X_test,
        y_test,
        train_indices,
        test_indices,
        train_images,
        test_images,
    )


# ============================================================
# 4) Model + feature importance
# ============================================================

def build_svm_pipeline() -> Pipeline:
    return Pipeline(
        [
            ("scaler", StandardScaler()),
            (
                "svm",
                LinearSVC(
                    C=1.0,
                    class_weight=None,
                    random_state=42,
                    max_iter=10000,
                ),
            ),
        ]
    )


def analyze_feature_importance(model: Pipeline, feature_names: List[str],
result_folder: str) -> np.ndarray:
    weights = model.named_steps["svm"].coef_[0]
    abs_weights = np.abs(weights)
    normalized_weights = abs_weights / (abs_weights.sum() + 1e-12)

    df = pd.DataFrame({"Feature": feature_names, "Importance":
normalized_weights})
    print("\nSorted Feature Importance (|w| on standardized features):")
    print(df.sort_values("Importance",
ascending=False).to_markdown(tablefmt="grid", index=False, floatfmt=".6f"))

    plt.figure(figsize=(10, 0.30 * len(feature_names) + 2))
```

```
    sorted_idx = np.argsort(normalized_weights)
    plt.barh(range(len(sorted_idx)), normalized_weights[sorted_idx])
    plt.yticks(range(len(sorted_idx)), [feature_names[i] for i in sorted_idx])
    plt.xlabel("Feature Importance (|weight| on standardized features)")
    plt.title("Linear SVM Feature Importance")
    plt.tight_layout()
    importance_path = os.path.join(result_folder,
"svm_feature_importance.png")
    plt.savefig(importance_path, bbox_inches="tight", dpi=300)
    plt.close()
    print(f"Saved feature importance plot to: {importance_path}")
    return normalized_weights


# ============================================================
# 5) Dense visualization + cutouts
# ============================================================

def _maybe_postprocess(mask01: np.ndarray, enable: bool = False) ->
np.ndarray:
    if not enable:
        return mask01
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (3, 3))
    m = cv2.morphologyEx((mask01 * 255).astype(np.uint8), cv2.MORPH_OPEN,
kernel)
    m = (m > 127).astype(np.uint8)
    num, labels, stats, _ = cv2.connectedComponentsWithStats(m,
connectivity=8)
    if num <= 1:
        return m
    keep = 1 + np.argmax(stats[1:, cv2.CC_STAT_AREA])
    return (labels == keep).astype(np.uint8)


def visualize_sample_results(
    model: LinearSVC,
    scaler: StandardScaler,
    images: List[Tuple[int, np.ndarray, np.ndarray]],
    result_folder: str,
    split_tag: str = "test",
    window_size: int = 7,
    pred_batch: int = 200_000,
    cutout_root: str | None = None,
    postprocess: bool = False,
    decision_threshold: float = 0.0,
    bf_tolerance_px: int = 3,
) -> None:
    os.makedirs(result_folder, exist_ok=True)
```

```
    cutout_dir = None
    if cutout_root is not None:
        cutout_dir = os.path.join(cutout_root, split_tag)
        os.makedirs(cutout_dir, exist_ok=True)

    for img_idx, img, mask in images:
        H, W = img.shape[:2]

        # Dense features
        X_img = _build_full_image_features(img, window_size=window_size)

        # Scale+predict in chunks
        preds = np.empty((H * W,), dtype=np.int32)
        start = 0
        while start < X_img.shape[0]:
            end = min(start + pred_batch, X_img.shape[0])
            scores =
model.decision_function(scaler.transform(X_img[start:end]))
            preds[start:end] = (scores > decision_threshold).astype(np.int32)
            start = end

        pred_mask = preds.reshape(H, W).astype(np.uint8)
        pred_mask = _maybe_postprocess(pred_mask, enable=postprocess)

        # Metrics per-image
        mask_bin = (mask != 0)
        mask_flat = mask_bin.reshape(-1)
        pred_flat = (pred_mask == 1).reshape(-1)

        tp = (pred_mask == 1) & mask_bin
        fp = (pred_mask == 1) & (~mask_bin)
        fn = (pred_mask == 0) & mask_bin

        overlay = np.zeros_like(img)
        overlay[tp] = [0, 255, 0]
        overlay[fp] = [255, 0, 0]
        overlay[fn] = [0, 0, 255]

        cutout = np.zeros_like(img)
        cutout[pred_mask == 1] = img[pred_mask == 1]

        plt.figure(figsize=(20, 5))
        plt.subplot(141);
plt.imshow(img);              plt.title("Original");                plt.axis
("off")
        plt.subplot(142); plt.imshow(mask, cmap="gray"); plt.title("Ground
Truth");           plt.axis("off")
```

```
        plt.subplot(143); plt.imshow(cutout);            plt.title("Predicted
(cut-out)");       plt.axis("off")
        plt.subplot(144); plt.imshow(img); plt.imshow(overlay, alpha=0.5)
        plt.title("Overlay: Green=TP, Red=FP, Blue=FN"); plt.axis("off")
        plt.tight_layout()

        out_png = f"{result_folder}/{split_tag}_{img_idx}.png"
        plt.savefig(out_png, bbox_inches="tight", dpi=150)
        plt.close()

        if cutout_dir is not None:
            cutout_path = os.path.join(cutout_dir,
f"{split_tag}_{img_idx}.png")
            cv2.imwrite(cutout_path, cv2.cvtColor(cutout, cv2.COLOR_RGB2BGR))

        # --- Boundary-aware metrics ---
        cm = confusion_matrix(mask_flat, pred_flat, labels=[0,1])
        tn, fp, fn, tp = cm.ravel()
        specificity = tn / (tn + fp + 1e-9)
        balanced_acc = 0.5 * (specificity + recall_score(mask_flat,
pred_flat))

        bf1, assd, hd95 = compute_boundary_metrics(mask_bin.astype(np.uint8),
(pred_mask==1).astype(np.uint8), tolerance=bf_tolerance_px)

        metrics = {
            "Image Index": int(img_idx),
            "True Foreground Pixels": int(mask_bin.sum()),
            "Predicted Foreground Pixels": int((pred_mask == 1).sum()),
            "Dice (F1)": float(f1_score(mask_flat, pred_flat)),
            "Jaccard": float(jaccard_score(mask_flat, pred_flat)),
            "Precision": float(precision_score(mask_flat, pred_flat)),
            "Recall": float(recall_score(mask_flat, pred_flat)),
            "Specificity": float(specificity),
            "Balanced Accuracy": float(balanced_acc),
            "BoundaryF1@%dpx" % bf_tolerance_px: float(bf1) if not
np.isnan(bf1) else float("nan"),
            "ASSD_px": float(assd) if assd is not None else float("nan"),
            "HD95_px": float(hd95) if hd95 is not None else float("nan"),
            "Accuracy": float(accuracy_score(mask_flat, pred_flat)),
        }
        out_txt = f"{result_folder}/{split_tag}_{img_idx}.txt"
        with open(out_txt, "w") as f:
            for k, v in metrics.items():
                f.write(f"{k}: {v}\n")


# ========================================================
```

```python
# 6) Robust evaluation utilities
# ============================================================

# --- Boundary metrics helpers ---

def _binary_boundary(mask01: np.ndarray) -> np.ndarray:
    """1px boundary of a 0/1 mask using morphological gradient."""
    m = (mask01.astype(np.uint8) > 0).astype(np.uint8)
    kernel = cv2.getStructuringElement(cv2.MORPH_RECT, (3, 3))
    er = cv2.erode(m, kernel, iterations=1)
    bd = (m - er)
    bd[bd < 0] = 0
    return (bd > 0).astype(np.uint8)


def _distance_to_boundary(boundary01: np.ndarray) -> np.ndarray:
    """Euclidean distance (pixels) to nearest boundary pixel using OpenCV
DT."""
    inv = (boundary01 == 0).astype(np.uint8) * 255
    dt = cv2.distanceTransform(inv, cv2.DIST_L2, 3)
    return dt.astype(np.float32)


def compute_boundary_metrics(gt01: np.ndarray, pr01: np.ndarray, tolerance:
int = 3):
    """Return (BF1@tol, ASSD_px, HD95_px). NaN if boundary empty."""
    gt01 = (gt01 > 0).astype(np.uint8)
    pr01 = (pr01 > 0).astype(np.uint8)
    b_gt = _binary_boundary(gt01)
    b_pr = _binary_boundary(pr01)

    if b_gt.sum() == 0 and b_pr.sum() == 0:
        return float('nan'), 0.0, 0.0

    dt_gt = _distance_to_boundary(b_gt)
    dt_pr = _distance_to_boundary(b_pr)

    pred_d = dt_gt[b_pr == 1]
    true_d = dt_pr[b_gt == 1]

    # BF precision / recall within tolerance
    p = np.nan if pred_d.size == 0 else (pred_d <= tolerance).mean()
    r = np.nan if true_d.size == 0 else (true_d <= tolerance).mean()
    if np.isnan(p) or np.isnan(r) or (p + r) == 0:
        bf1 = np.nan
    else:
        bf1 = 2 * p * r / (p + r + 1e-9)
```

```
    # ASSD (symmetric) and HD95
    vals = []
    if pred_d.size > 0:
        vals.append(pred_d)
    if true_d.size > 0:
        vals.append(true_d)
    if len(vals) == 0:
        assd, hd95 = 0.0, 0.0
    else:
        vals = np.concatenate(vals)
        assd = float(np.mean(vals))
        hd95 = float(np.percentile(vals, 95))

    return float(bf1), assd, hd95

def _metrics_from_cm(cm: np.ndarray) -> Dict[str, float]:
    tn, fp, fn, tp = cm.ravel()
    eps = 1e-9
    precision = tp / (tp + fp + eps)
    recall = tp / (tp + fn + eps)
    dice = 2 * tp / (2 * tp + fp + fn + eps)
    jaccard = tp / (tp + fp + fn + eps)
    accuracy = (tp + tn) / (tp + tn + fp + fn + eps)
    return {
        "Accuracy": accuracy,
        "Dice Score": dice,
        "Precision": precision,
        "Recall": recall,
        "Jaccard": jaccard,
    }


def evaluate_sampled(y_true: np.ndarray, y_pred: np.ndarray) -> Dict[str,
float]:
    cm = confusion_matrix(y_true, y_pred, labels=[0, 1])
    print("Confusion Matrix (sampled):\n", cm)
    metrics = _metrics_from_cm(cm)
    for k, v in metrics.items():
        print(f"{k}: {v:.4f}")
    return metrics


def evaluate_dense_set(
    model: LinearSVC,
    scaler: StandardScaler,
    images: List[Tuple[int, np.ndarray, np.ndarray]],
    window_size: int = 7,
    pred_batch: int = 200_000,
```

```
    postprocess: bool = False,
    decision_threshold: float = 0.0,
    bf_tolerance_px: int = 3,
) -> Dict[str, float]:

    cm_total = np.zeros((2, 2), dtype=np.int64)
    bf_list, assd_list, hd95_list = [], [], []
    for _, img, mask in tqdm(images, desc="Dense eval"):
        H, W = img.shape[:2]
        X_img = _build_full_image_features(img, window_size=window_size)
        preds = np.empty((H * W,), dtype=np.int32)
        s = 0
        while s < X_img.shape[0]:
            e = min(s + pred_batch, X_img.shape[0])
            scores = model.decision_function(scaler.transform(X_img[s:e]))
            preds[s:e] = (scores > decision_threshold).astype(np.int32)
            s = e
        pred_mask = preds.reshape(H, W).astype(np.uint8)
        pred_mask = _maybe_postprocess(pred_mask, enable=postprocess)

        y_true = (mask != 0).reshape(-1)
        y_pred = (pred_mask == 1).reshape(-1)
        cm_total += confusion_matrix(y_true, y_pred, labels=[0, 1])

        bf1, assd, hd95 =
compute_boundary_metrics(y_true.reshape(img.shape[:2]).astype(np.uint8),
(pred_mask==1).astype(np.uint8), tolerance=bf_tolerance_px)
        bf_list.append(bf1)
        assd_list.append(assd)
        hd95_list.append(hd95)

    print("Global Confusion Matrix (dense over all pixels):", cm_total)
    metrics = _metrics_from_cm(cm_total)
    tn, fp, fn, tp = cm_total.ravel()
    specificity = tn / (tn + fp + 1e-9)
    bal_acc = 0.5 * (specificity + metrics["Recall"])
    metrics["Specificity"] = specificity
    metrics["Balanced Accuracy"] = bal_acc
    metrics["BoundaryF1@%dpx" % bf_tolerance_px] = float(np.nanmean(bf_list))
if len(bf_list) else float("nan")
    metrics["ASSD_px_mean"] = float(np.nanmean(assd_list)) if len(assd_list)
else float("nan")
    metrics["HD95_px_mean"] = float(np.nanmean(hd95_list)) if len(hd95_list)
else float("nan")

    print("Dense (global) metrics:")
    for k, v in metrics.items():
        print(f"{k}: {v:.4f}")
```

```
    return metrics


# ============================================================
# 7) Main
# ============================================================

def calibrate_threshold(
    model: LinearSVC,
    scaler: StandardScaler,
    images: List[Tuple[int, np.ndarray, np.ndarray]],
    thresholds: np.ndarray | List[float] = None,
    window_size: int = 7,
    pred_batch: int = 200_000,
    postprocess: bool = False,
    bf_tolerance_px: int = 3,
) -> float:

    if thresholds is None:
        thresholds = np.linspace(-0.5, 0.5, 21)
    subset = images[: min(16, len(images))]
    best_tau, best_score = 0.0, -1.0
    for tau in thresholds:
        dices = []
        for _, img, mask in subset:
            H, W = img.shape[:2]
            X_img = _build_full_image_features(img, window_size=window_size)
            preds = np.empty((H * W,), dtype=np.int32)
            s = 0
            while s < X_img.shape[0]:
                e = min(s + pred_batch, X_img.shape[0])
                scores = model.decision_function(scaler.transform(X_img[s:e]))
                preds[s:e] = (scores > tau).astype(np.int32)
                s = e
            pred_mask = preds.reshape(H, W).astype(np.uint8)
            pred_mask = _maybe_postprocess(pred_mask, enable=postprocess)

            y_true = (mask != 0).reshape(-1)
            y_pred = (pred_mask == 1).reshape(-1)
            dices.append(f1_score(y_true, y_pred))
        mean_dice = float(np.mean(dices)) if len(dices) else -1.0
        if mean_dice > best_score:
            best_score, best_tau = mean_dice, float(tau)
    print(f"Calibrated threshold (by mean Dice on subset): tau={best_tau:.3f}
(score={best_score:.4f})")
    return best_tau
```

```
def main():
    image_folder = "/content/drive/MyDrive/svm/data/images"
    mask_folder = "/content/drive/MyDrive/svm/data/masks"
    result_folder = "/content/drive/MyDrive/svm/svm_linear_results_fixed"
    cutout_root = "/content/drive/MyDrive/svm/predicted_cutouts"

    drive.mount("/content/drive", force_remount=False)
    os.makedirs(result_folder, exist_ok=True)
    os.makedirs(cutout_root, exist_ok=True)

    (
        X_train,
        y_train,
        X_test,
        y_test,
        train_indices,
        test_indices,
        train_images,
        test_images,
    ) = load_and_split_data(image_folder, mask_folder, n_images=500,
test_size=0.2, fg_per_img=4000, bg_per_img=4000)

    print(f"\nTraining sampled pixels: {len(X_train):,}")
    print(f"Test sampled pixels:     {len(X_test):,}")
    print(f"Train images: {len(train_indices)} | Test images:
{len(test_indices)}")

    # Train
    print("\nTraining Linear SVM model...")
    model = build_svm_pipeline()
    model.fit(X_train, y_train)

    # Feature importance (names from extractor)
    _, feature_names = extract_features(train_images[0][1], x=0, y=0)
    feature_importances = analyze_feature_importance(model, feature_names,
result_folder)

    # Complexity
    svm = model.named_steps["svm"]
    complexity = {
        "Total params": int(len(svm.coef_[0]) + 1),
        "Model size (approx)": f"{(svm.coef_.nbytes + svm.intercept_.nbytes) /
1024:.2f} KB",
        "Feature space": f"ℝ^{len(svm.coef_[0])}",
        "Regularization (C)": float(svm.C),
    }
    print("\nModel Complexity Metrics:")
    for k, v in complexity.items():
```

```
        print(f"{k}: {v}")

    # Quick sampled report (now balanced)
    print("\nTest Set Evaluation (BALANCED sampled pixels):")
    y_pred_sampled = model.predict(X_test)
    sampled_metrics = evaluate_sampled(y_test, y_pred_sampled)

    # Dense visualizations + per-image metrics
    print("Calibrating decision threshold on a small subset (mean Dice)...")
    DECISION_THRESHOLD = calibrate_threshold(
        model.named_steps["svm"],
        model.named_steps["scaler"],
        train_images,
        thresholds=np.linspace(-0.5, 0.5, 21),
        postprocess=False,
        bf_tolerance_px=3,
    )

    print("Generating train visualizations (dense)...")
    visualize_sample_results(
        model.named_steps["svm"],
        model.named_steps["scaler"],
        train_images,
        result_folder,
        split_tag="train",
        cutout_root=cutout_root,
        postprocess=False,
        decision_threshold=DECISION_THRESHOLD,
        bf_tolerance_px=3,
    )

    print("\nGenerating test visualizations (dense)...")
    visualize_sample_results(
        model.named_steps["svm"],
        model.named_steps["scaler"],
        test_images,
        result_folder,
        split_tag="test",
        cutout_root=cutout_root,
        postprocess=False,
        decision_threshold=DECISION_THRESHOLD,
        bf_tolerance_px=3,
    )

    # Dense, aggregated metrics across ALL pixels
    dense_metrics = evaluate_dense_set(
        model.named_steps["svm"],
        model.named_steps["scaler"],
```

```
        test_images,
        window_size=7,
        postprocess=False,
        decision_threshold=DECISION_THRESHOLD,
        bf_tolerance_px=3,
    )

    # Save everything
    bundle = {
        "model": model,
        "feature_importances": feature_importances,
        "sampled_metrics_balanced": sampled_metrics,
        "dense_metrics": dense_metrics,
        "train_indices": train_indices,
        "test_indices": test_indices,
        "model_complexity": complexity,
    }
    joblib.dump(bundle, os.path.join(result_folder,
"linear_svm_results_fixed.joblib"))

    with open(os.path.join(result_folder, "dense_metrics.txt"), "w") as f:
        for k, v in dense_metrics.items():
            f.write(f"{k}: {v}\n")

    print(f"\nResults saved to {result_folder}")
    print(f"Color-preserved predictions saved under: {cutout_root}/train and
{cutout_root}/test")


if __name__ == "__main__":
    main()
```

DuckNet (from https://github.com/RazvanDu/DUCK-Net)

Consist of:

- ModelNotebook.ipynb
- ImageLoader2D.py
- ConvBlock2D.py
- DUCK_Net.py
- DiceLoss.py

ModelNotebook.ipynb

```
from google.colab import drive
import sys
import tensorflow as tf
import numpy as np
import gc
import matplotlib.pyplot as plt
from keras.callbacks import CSVLogger
from datetime import datetime
from sklearn.model_selection import train_test_split
from sklearn.metrics import jaccard_score, precision_score, recall_score,
accuracy_score, f1_score
from PIL import Image
import os

# Mount Google Drive and set up paths
drive.mount('/content/drive')
sys.path.append('/content/drive/My Drive/duck')
sys.path.append('/content/drive/My Drive/duck/ModelArchitecture')
sys.path.append('/content/drive/My Drive/duck/ImageLoader')
sys.path.append('/content/drive/My Drive/duck/CustomLayers')

# Import project-specific modules
from ModelArchitecture.DiceLoss import dice_metric_loss
from ModelArchitecture import DUCK_Net
from ImageLoader import ImageLoader2D

print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))

# Model settings
img_size = 352
dataset_type = 'my_dataset'
learning_rate = 1e-4
seed_value = 58800
filters = 17
optimizer = tf.keras.optimizers.RMSprop(learning_rate=learning_rate)
ct = datetime.now()
```

```
model_type = "DuckNet"
progress_path =
f'ProgressFull/{dataset_type}_progress_csv_{model_type}_filters_{filters}_{ct}
.csv'
progressfull_path =
f'ProgressFull/{dataset_type}_progress_{model_type}_filters_{filters}_{ct}.txt
'
plot_path =
f'ProgressFull/{dataset_type}_progress_plot_{model_type}_filters_{filters}_{ct
}.png'
model_path =
f'ModelSaveTensorFlow/{dataset_type}/{model_type}_filters_{filters}_{ct}'

EPOCHS = 100
min_loss_for_saving = 0.2

# Load data
X, Y, filenames = ImageLoader2D.load_data(img_size, img_size, -1,
'my_dataset')
x_train, x_test, y_train, y_test, f_train, f_test = train_test_split(X, Y,
filenames, test_size=0.1, random_state=seed_value, shuffle=True)
x_train, x_valid, y_train, y_valid, f_train, f_valid =
train_test_split(x_train, y_train, f_train, test_size=0.111,
random_state=seed_value, shuffle=True)

# Create and compile model
model = DUCK_Net.create_model(img_height=img_size, img_width=img_size,
input_chanels=3, out_classes=1, starting_filters=filters)
model.compile(optimizer=optimizer, loss=dice_metric_loss)

# Training loop
step = 0
for epoch in range(0, EPOCHS):
    print(f'Training, epoch {epoch}')
    print('Learning Rate: ' + str(learning_rate))
    step += 1

    os.makedirs("ProgressFull", exist_ok=True)
    csv_logger = CSVLogger(progress_path, append=True, separator=';')
    model.fit(x=x_train, y=y_train, epochs=1, batch_size=4,
validation_data=(x_valid, y_valid), verbose=1, callbacks=[csv_logger])

    prediction_valid = model.predict(x_valid, verbose=0)
    loss_valid = dice_metric_loss(y_valid, prediction_valid).numpy()
    print("Loss Validation: " + str(loss_valid))

    prediction_test = model.predict(x_test, verbose=0)
```

```
    loss_test = dice_metric_loss(y_test, prediction_test).numpy()
    print("Loss Test: " + str(loss_test))

    with open(progressfull_path, 'a') as f:
        f.write('epoch: ' + str(epoch) + '\nval_loss: ' + str(loss_valid) +
'\ntest_loss: ' + str(loss_test) + '\n\n\n')

    if min_loss_for_saving > loss_valid:
        min_loss_for_saving = loss_valid
        print("Saved model with val_loss: ", loss_valid)
        model.save(model_path + '.h5')

    gc.collect()

# Reload best model
print("Loading the model")
model = tf.keras.models.load_model(model_path + '.h5',
custom_objects={'dice_metric_loss': dice_metric_loss})

# Predictions
prediction_train = model.predict(x_train, batch_size=4)
prediction_valid = model.predict(x_valid, batch_size=4)
prediction_test = model.predict(x_test, batch_size=4)

print("Predictions done")

# Metrics
flatten = lambda arr: np.ndarray.flatten(np.array(arr, dtype=bool))
bin_pred = lambda pred: np.ndarray.flatten(pred > 0.5)

dice_train = f1_score(flatten(y_train), bin_pred(prediction_train))
dice_test = f1_score(flatten(y_test), bin_pred(prediction_test))
dice_valid = f1_score(flatten(y_valid), bin_pred(prediction_valid))

miou_train = jaccard_score(flatten(y_train), bin_pred(prediction_train))
miou_test = jaccard_score(flatten(y_test), bin_pred(prediction_test))
miou_valid = jaccard_score(flatten(y_valid), bin_pred(prediction_valid))

precision_train = precision_score(flatten(y_train),
bin_pred(prediction_train))
precision_test = precision_score(flatten(y_test), bin_pred(prediction_test))
precision_valid = precision_score(flatten(y_valid),
bin_pred(prediction_valid))

recall_train = recall_score(flatten(y_train), bin_pred(prediction_train))
recall_test = recall_score(flatten(y_test), bin_pred(prediction_test))
recall_valid = recall_score(flatten(y_valid), bin_pred(prediction_valid))
```

```
accuracy_train = accuracy_score(flatten(y_train), bin_pred(prediction_train))
accuracy_test = accuracy_score(flatten(y_test), bin_pred(prediction_test))
accuracy_valid = accuracy_score(flatten(y_valid), bin_pred(prediction_valid))

final_file = f'results_{model_type}_{filters}_{dataset_type}.txt'
with open(final_file, 'a') as f:
    f.write(dataset_type + '\n\n')
    f.write('dice_train: ' + str(dice_train) + ' dice_valid: ' +
str(dice_valid) + ' dice_test: ' + str(dice_test) + '\n\n')
    f.write('miou_train: ' + str(miou_train) + ' miou_valid: ' +
str(miou_valid) + ' miou_test: ' + str(miou_test) + '\n\n')
    f.write('precision_train: ' + str(precision_train) + ' precision_valid: '
+ str(precision_valid) + ' precision_test: ' + str(precision_test) + '\n\n')
    f.write('recall_train: ' + str(recall_train) + ' recall_valid: ' +
str(recall_valid) + ' recall_test: ' + str(recall_test) + '\n\n')
    f.write('accuracy_train: ' + str(accuracy_train) + ' accuracy_valid: ' +
str(accuracy_valid) + ' accuracy_test: ' + str(accuracy_test) + '\n\n\n\n')

# Save segmented images with original filenames for all sets
save_path = "/content/drive/My Drive/duck/segmented_results/"
os.makedirs(save_path, exist_ok=True)

def save_segmented_images(images, filenames, model, save_dir):
    for i in range(len(images)):
        img = (images[i] * 255).astype(np.uint8)
        pred = model.predict(images[i].reshape(1, *images[i].shape))[0, :, :,
0]
        binary_mask = (pred > 0.5).astype(np.uint8)
        mask_3c = np.repeat(binary_mask[:, :, np.newaxis], 3, axis=-1)
        segmented = img * mask_3c

        out_file = os.path.join(save_dir, filenames[i])
        Image.fromarray(segmented).save(out_file)
        print(f"Saved: {out_file}")

save_segmented_images(x_train, f_train, model, save_path)
save_segmented_images(x_valid, f_valid, model, save_path)
save_segmented_images(x_test, f_test, model, save_path)

print(f'Dice Score - Train: {dice_train}, Valid: {dice_valid}, Test:
{dice_test}')
print(f'MIoU - Train: {miou_train}, Valid: {miou_valid}, Test: {miou_test}')
print(f'Precision - Train: {precision_train}, Valid: {precision_valid}, Test:
{precision_test}')
print(f'Recall - Train: {recall_train}, Valid: {recall_valid}, Test:
{recall_test}')
print(f'Accuracy - Train: {accuracy_train}, Valid: {accuracy_valid}, Test:
{accuracy_test}')
```

ImageLoader.py

```
import os
import glob
import numpy as np
from PIL import Image
from skimage.io import imread
from tqdm import tqdm

folder_path = "/content/drive/My Drive/duck/data/"  # Add the path to data
directory

def load_data(img_height, img_width, images_to_be_loaded, dataset):
    IMAGES_PATH = folder_path + 'images/'
    MASKS_PATH = folder_path + 'masks/'

    if dataset == 'my_dataset':
        train_ids = glob.glob(IMAGES_PATH + "*.png")

    if images_to_be_loaded == -1:
        images_to_be_loaded = len(train_ids)

    X_train = np.zeros((images_to_be_loaded, img_height, img_width, 3),
dtype=np.float32)
    Y_train = np.zeros((images_to_be_loaded, img_height, img_width),
dtype=np.uint8)
    filename_list = []

    print('Loading training images and masks (no resizing):',
images_to_be_loaded)
    for n, id_ in tqdm(enumerate(train_ids)):
        if n == images_to_be_loaded:
            break

        image_path = id_
        mask_path = image_path.replace("images", "masks")
        filename = os.path.basename(image_path)

        image = imread(image_path)
        mask_ = imread(mask_path)

        X_train[n] = image / 255.0

        mask = np.zeros((img_height, img_width), dtype=np.bool_)
        for i in range(img_height):
            for j in range(img_width):
                if mask_[i, j] >= 127:
                    mask[i, j] = 1
```

```
        Y_train[n] = mask
        filename_list.append(filename)

    Y_train = np.expand_dims(Y_train, axis=-1)

    return X_train, Y_train, filename_list
```

ConvBlock2D.py

```
from keras.layers import BatchNormalization, add
from keras.layers import Conv2D

kernel_initializer = 'he_uniform'


def conv_block_2D(x, filters, block_type, repeat=1, dilation_rate=1, size=3,
padding='same'):
    result = x

    for i in range(0, repeat):

        if block_type == 'separated':
            result = separated_conv2D_block(result, filters, size=size,
padding=padding)
        elif block_type == 'duckv2':
            result = duckv2_conv2D_block(result, filters, size=size)
        elif block_type == 'midscope':
            result = midscope_conv2D_block(result, filters)
        elif block_type == 'widescope':
            result = widescope_conv2D_block(result, filters)
        elif block_type == 'resnet':
            result = resnet_conv2D_block(result, filters, dilation_rate)
        elif block_type == 'conv':
            result = Conv2D(filters, (size, size),
                            activation='relu',
kernel_initializer=kernel_initializer, padding=padding)(result)
        elif block_type == 'double_convolution':
            result = double_convolution_with_batch_normalization(result,
filters, dilation_rate)

        else:
            return None

    return result


def duckv2_conv2D_block(x, filters, size):
    x = BatchNormalization(axis=-1)(x)
    x1 = widescope_conv2D_block(x, filters)

    x2 = midscope_conv2D_block(x, filters)

    x3 = conv_block_2D(x, filters, 'resnet', repeat=1)

    x4 = conv_block_2D(x, filters, 'resnet', repeat=2)
```

```
    x5 = conv_block_2D(x, filters, 'resnet', repeat=3)

    x6 = separated_conv2D_block(x, filters, size=6, padding='same')

    x = add([x1, x2, x3, x4, x5, x6])

    x = BatchNormalization(axis=-1)(x)

    return x


def separated_conv2D_block(x, filters, size=3, padding='same'):
    x = Conv2D(filters, (1, size), activation='relu',
kernel_initializer=kernel_initializer, padding=padding)(x)

    x = BatchNormalization(axis=-1)(x)

    x = Conv2D(filters, (size, 1), activation='relu',
kernel_initializer=kernel_initializer, padding=padding)(x)

    x = BatchNormalization(axis=-1)(x)

    return x


def midscope_conv2D_block(x, filters):
    x = Conv2D(filters, (3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same',
               dilation_rate=1)(x)

    x = BatchNormalization(axis=-1)(x)

    x = Conv2D(filters, (3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same',
               dilation_rate=2)(x)

    x = BatchNormalization(axis=-1)(x)

    return x


def widescope_conv2D_block(x, filters):
    x = Conv2D(filters, (3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same',
               dilation_rate=1)(x)

    x = BatchNormalization(axis=-1)(x)
```

```
    x = Conv2D(filters, (3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same',
               dilation_rate=2)(x)

    x = BatchNormalization(axis=-1)(x)

    x = Conv2D(filters, (3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same',
               dilation_rate=3)(x)

    x = BatchNormalization(axis=-1)(x)

    return x


def resnet_conv2D_block(x, filters, dilation_rate=1):
    x1 = Conv2D(filters, (1, 1), activation='relu',
kernel_initializer=kernel_initializer, padding='same',
                dilation_rate=dilation_rate)(x)

    x = Conv2D(filters, (3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same',
               dilation_rate=dilation_rate)(x)
    x = BatchNormalization(axis=-1)(x)
    x = Conv2D(filters, (3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same',
               dilation_rate=dilation_rate)(x)
    x = BatchNormalization(axis=-1)(x)
    x_final = add([x, x1])

    x_final = BatchNormalization(axis=-1)(x_final)

    return x_final


def double_convolution_with_batch_normalization(x, filters, dilation_rate=1):
    x = Conv2D(filters, (3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same',
               dilation_rate=dilation_rate)(x)
    x = BatchNormalization(axis=-1)(x)
    x = Conv2D(filters, (3, 3), activation='relu',
kernel_initializer=kernel_initializer, padding='same',
               dilation_rate=dilation_rate)(x)
    x = BatchNormalization(axis=-1)(x)

    return x
```

DUCK_Net.py

```
import tensorflow as tf
from keras.layers import Conv2D, UpSampling2D
from keras.layers import add
from keras.models import Model

from CustomLayers.ConvBlock2D import conv_block_2D

kernel_initializer = 'he_uniform'
interpolation = "nearest"


def create_model(img_height, img_width, input_chanels, out_classes,
starting_filters):
    input_layer = tf.keras.layers.Input((img_height, img_width,
input_chanels))

    print('Starting DUCK-Net')

    p1 = Conv2D(starting_filters * 2, 2, strides=2,
padding='same')(input_layer)
    p2 = Conv2D(starting_filters * 4, 2, strides=2, padding='same')(p1)
    p3 = Conv2D(starting_filters * 8, 2, strides=2, padding='same')(p2)
    p4 = Conv2D(starting_filters * 16, 2, strides=2, padding='same')(p3)
    p5 = Conv2D(starting_filters * 32, 2, strides=2, padding='same')(p4)

    t0 = conv_block_2D(input_layer, starting_filters, 'duckv2', repeat=1)

    l1i = Conv2D(starting_filters * 2, 2, strides=2, padding='same')(t0)
    s1 = add([l1i, p1])
    t1 = conv_block_2D(s1, starting_filters * 2, 'duckv2', repeat=1)

    l2i = Conv2D(starting_filters * 4, 2, strides=2, padding='same')(t1)
    s2 = add([l2i, p2])
    t2 = conv_block_2D(s2, starting_filters * 4, 'duckv2', repeat=1)

    l3i = Conv2D(starting_filters * 8, 2, strides=2, padding='same')(t2)
    s3 = add([l3i, p3])
    t3 = conv_block_2D(s3, starting_filters * 8, 'duckv2', repeat=1)

    l4i = Conv2D(starting_filters * 16, 2, strides=2, padding='same')(t3)
    s4 = add([l4i, p4])
    t4 = conv_block_2D(s4, starting_filters * 16, 'duckv2', repeat=1)

    l5i = Conv2D(starting_filters * 32, 2, strides=2, padding='same')(t4)
    s5 = add([l5i, p5])
    t51 = conv_block_2D(s5, starting_filters * 32, 'resnet', repeat=2)
```

```
t53 = conv_block_2D(t51, starting_filters * 16, 'resnet', repeat=2)

l5o = UpSampling2D((2, 2), interpolation=interpolation)(t53)
c4 = add([l5o, t4])
q4 = conv_block_2D(c4, starting_filters * 8, 'duckv2', repeat=1)

l4o = UpSampling2D((2, 2), interpolation=interpolation)(q4)
c3 = add([l4o, t3])
q3 = conv_block_2D(c3, starting_filters * 4, 'duckv2', repeat=1)

l3o = UpSampling2D((2, 2), interpolation=interpolation)(q3)
c2 = add([l3o, t2])
q6 = conv_block_2D(c2, starting_filters * 2, 'duckv2', repeat=1)

l2o = UpSampling2D((2, 2), interpolation=interpolation)(q6)
c1 = add([l2o, t1])
q1 = conv_block_2D(c1, starting_filters, 'duckv2', repeat=1)

l1o = UpSampling2D((2, 2), interpolation=interpolation)(q1)
c0 = add([l1o, t0])
z1 = conv_block_2D(c0, starting_filters, 'duckv2', repeat=1)

output = Conv2D(out_classes, (1, 1), activation='sigmoid')(z1)

model = Model(inputs=input_layer, outputs=output)

return model
```

DiceLoss.py

```
import tensorflow.keras.backend as K
import tensorflow as tf


def dice_metric_loss(ground_truth, predictions, smooth=1e-6):
    ground_truth = K.cast(ground_truth, tf.float32)
    predictions = K.cast(predictions, tf.float32)
    ground_truth = K.flatten(ground_truth)
    predictions = K.flatten(predictions)
    intersection = K.sum(predictions * ground_truth)
    union = K.sum(predictions) + K.sum(ground_truth)

    dice = (2. * intersection + smooth) / (union + smooth)

    return 1 - dice
```

AdderNet (from https://github.com/huawei-noah/AdderNet)

Consist of:

- AdderNet.ipynb
- adder.py
- resnet20.py
- main.py
- test.py

AdderNet.ipynb

```
from google.colab import drive
drive.mount('/content/drive')

import os
os.chdir('/content/drive/MyDrive/AdderNet')

!pip install torch torchvision

!python main.py --data /content/drive/MyDrive/AdderNet/root/cifar10-png --
output_dir /content/drive/MyDrive/AdderNet/output/

!python test.py --dataset cifar10 --data_dir
/content/drive/MyDrive/AdderNet/root/cifar10-png --model_dir
/content/drive/MyDrive/AdderNet/output/addernet.pth
```

adder.py

```
import torch
import torch.nn as nn
import numpy as np
from torch.autograd import Function
import math

def adder2d_function(X, W, stride=1, padding=0):
    n_filters, d_filter, h_filter, w_filter = W.size()
    n_x, d_x, h_x, w_x = X.size()

    h_out = (h_x - h_filter + 2 * padding) / stride + 1
    w_out = (w_x - w_filter + 2 * padding) / stride + 1

    h_out, w_out = int(h_out), int(w_out)
    X_col = torch.nn.functional.unfold(X.view(1, -1, h_x, w_x), h_filter,
dilation=1, padding=padding, stride=stride).view(n_x, -1, h_out*w_out)
    X_col = X_col.permute(1,2,0).contiguous().view(X_col.size(1),-1)
    W_col = W.view(n_filters, -1)

    out = adder.apply(W_col,X_col)

    out = out.view(n_filters, h_out, w_out, n_x)
    out = out.permute(3, 0, 1, 2).contiguous()

    return out

class adder(Function):
    @staticmethod
    def forward(ctx, W_col, X_col):
        ctx.save_for_backward(W_col,X_col)
        output = -(W_col.unsqueeze(2)-X_col.unsqueeze(0)).abs().sum(1)
        return output

    @staticmethod
    def backward(ctx,grad_output):
        W_col,X_col = ctx.saved_tensors
        grad_W_col = ((X_col.unsqueeze(0)-
W_col.unsqueeze(2))*grad_output.unsqueeze(1)).sum(2)
        grad_W_col = grad_W_col/grad_W_col.norm(p=2).clamp(min=1e-
12)*math.sqrt(W_col.size(1)*W_col.size(0))/5
        grad_X_col = (-(X_col.unsqueeze(0)-W_col.unsqueeze(2)).clamp(-
1,1)*grad_output.unsqueeze(1)).sum(0)

        return grad_W_col, grad_X_col

class adder2d(nn.Module):
```

```
    def __init__(self,input_channel,output_channel,kernel_size, stride=1,
padding=0, bias = False):
        super(adder2d, self).__init__()
        self.stride = stride
        self.padding = padding
        self.input_channel = input_channel
        self.output_channel = output_channel
        self.kernel_size = kernel_size
        self.adder =
torch.nn.Parameter(nn.init.normal_(torch.randn(output_channel,input_channel,ke
rnel_size,kernel_size)))
        self.bias = bias
        if bias:
            self.b =
torch.nn.Parameter(nn.init.uniform_(torch.zeros(output_channel)))

    def forward(self, x):
        output = adder2d_function(x,self.adder, self.stride, self.padding)
        if self.bias:
            output += self.b.unsqueeze(0).unsqueeze(2).unsqueeze(3)

        return output
```

resnet20.py

```
import adder
import torch.nn as nn


def conv3x3(in_planes, out_planes, stride=1):
    " 3x3 convolution with padding "
    return adder.adder2d(in_planes, out_planes, kernel_size=3, stride=stride,
padding=1, bias=False)


class BasicBlock(nn.Module):
    expansion=1

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride = stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out


class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=10):
        super(ResNet, self).__init__()
```

```
        self.inplanes = 16
        self.conv1 = nn.Conv2d(3, 16, kernel_size=7, stride=2, padding=3,
bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self._make_layer(block, 16, layers[0])
        self.layer2 = self._make_layer(block, 32, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 64, layers[2], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Conv2d(64 * block.expansion, num_classes, 1, bias=False)
        self.bn2 = nn.BatchNorm2d(num_classes)

        for m in self.modules():
            if isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                adder.adder2d(self.inplanes, planes * block.expansion,
kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion)
            )

        layers = []
        layers.append(block(inplanes = self.inplanes, planes = planes, stride
= stride, downsample = downsample))
        self.inplanes = planes * block.expansion
        for _ in range(1, blocks):
            layers.append(block(inplanes = self.inplanes, planes = planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)

        x = self.avgpool(x)
        x = self.fc(x)
        x = self.bn2(x)
```

```
        return x.view(x.size(0), -1)


def resnet20(num_classes=4, **kwargs): # num_class = 2 or 4 or 5
    return ResNet(BasicBlock, [3, 3, 3], num_classes=num_classes, **kwargs)
```

main.py

```
import os
from resnet20 import resnet20
import torch
from torch.autograd import Variable
from torchvision.datasets import ImageFolder
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
import argparse
import math

parser = argparse.ArgumentParser(description='train-addernet')
parser.add_argument('--data', type=str,
default='/content/drive/MyDrive/AdderNet/cifar10-png')
parser.add_argument('--output_dir', type=str, default='/cache/models/')
args = parser.parse_args()

os.makedirs(args.output_dir, exist_ok=True)

acc = 0
acc_best = 0

transform_train = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.2104, 0.1522, 0.1593), (0.2871, 0.2145, 0.2250)) #
mean & std for 2/4/5 classes training set
])

transform_test = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize((0.2104, 0.1522, 0.1593), (0.2871, 0.2145, 0.2250)) #
mean & std for 2/4/5 classes training set
])

data_train = ImageFolder(root=os.path.join(args.data, 'train'),
transform=transform_train)
data_test = ImageFolder(root=os.path.join(args.data, 'test'),
transform=transform_test)

data_train_loader = DataLoader(data_train, batch_size=16, shuffle=True,
num_workers=2)
data_test_loader = DataLoader(data_test, batch_size=16, num_workers=2)

net = resnet20(num_classes=4).cuda() # num_classes = 2 or 4 or 5
```

```
criterion = torch.nn.CrossEntropyLoss().cuda()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1, momentum=0.9,
weight_decay=5e-4)

def adjust_learning_rate(optimizer, epoch):
    lr = 0.05 * (1+math.cos(float(epoch)/50*math.pi)) # epoch = 50 or 100
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr

def train(epoch):
    adjust_learning_rate(optimizer, epoch)
    global cur_batch_win
    net.train()
    loss_list, batch_list = [], []
    for i, (images, labels) in enumerate(data_train_loader):
        images, labels = Variable(images).cuda(), Variable(labels).cuda()

        optimizer.zero_grad()

        output = net(images)

        loss = criterion(output, labels)

        loss_list.append(loss.data.item())
        batch_list.append(i+1)

        if i == 1:
            print('Train - Epoch %d, Batch: %d, Loss: %f' % (epoch, i,
loss.data.item()))

        loss.backward()
        optimizer.step()


def test():
    global acc, acc_best
    net.eval()
    total_correct = 0
    avg_loss = 0.0
    with torch.no_grad():
        for i, (images, labels) in enumerate(data_test_loader):
            images, labels = Variable(images).cuda(), Variable(labels).cuda()
            output = net(images)
            avg_loss += criterion(output, labels).sum()
            pred = output.data.max(1)[1]
            total_correct += pred.eq(labels.data.view_as(pred)).sum()

    avg_loss /= len(data_test)
```

```
    acc = float(total_correct) / len(data_test)
    if acc_best < acc:
        acc_best = acc
    print('Test Avg. Loss: %f, Accuracy: %f' % (avg_loss.data.item(), acc))


def train_and_test(epoch):
    train(epoch)
    test()


def main():
    epoch = 50 # epoch = 50 or 100
    for e in range(1, epoch):
        train_and_test(e)
    torch.save(net.state_dict(), args.output_dir + 'addernet.pth')


if __name__ == '__main__':
    main()
```

test.py

```
import os
import shutil
import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import argparse
from resnet20 import resnet20
import numpy as np
from sklearn.metrics import classification_report, confusion_matrix,
accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
import PIL

class ImageFolderWithPaths(datasets.ImageFolder):
    def __getitem__(self, index):
        original_tuple = super(ImageFolderWithPaths, self).__getitem__(index)
        path = self.imgs[index][0]
        return original_tuple + (path,)

def plot_confusion_matrix(cm, class_names, save_path):
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap="Blues",
xticklabels=class_names, yticklabels=class_names)
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.title('Confusion Matrix')
    plt.tight_layout()
    plt.savefig(save_path)
    plt.close()

def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

def get_model_size(model_path):
    size_bytes = os.path.getsize(model_path)
    return size_bytes / (1024 * 1024)

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument('--dataset', default='cifar10', type=str)
    parser.add_argument('--data_dir', default='./data', type=str)
    parser.add_argument('--model_dir', default='./model.pth', type=str)
    args = parser.parse_args()
```

```
    val_transform = transforms.Compose([
        transforms.Resize((224, 224)),
        transforms.ToTensor(),
        transforms.Normalize(mean=[0.2104, 0.1522, 0.1593], std=[0.2871,
0.2145, 0.2250]) # mean & std for 2/4/5 classes training set
    ])

    valdir = os.path.join(args.data_dir, 'test')
    val_dataset = ImageFolderWithPaths(valdir, transform=val_transform)
    val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)

    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model = resnet20()
    model.load_state_dict(torch.load(args.model_dir, map_location=device))
    model = model.to(device)
    model.eval()

    class_names = val_dataset.classes
    output_base = '/content/drive/MyDrive/AdderNet/results'
    os.makedirs(output_base, exist_ok=True)
    for class_name in class_names:
        os.makedirs(os.path.join(output_base, class_name), exist_ok=True)

    all_preds = []
    all_labels = []
    all_paths = []

    with torch.no_grad():
        for images, labels, paths in val_loader:
            images = images.to(device)
            outputs = model(images)
            _, preds = torch.max(outputs, 1)

            all_preds.extend(preds.cpu().numpy())
            all_labels.extend(labels.cpu().numpy())
            all_paths.extend(paths)

            for i in range(len(paths)):
                pred_class = class_names[preds[i]]
                dst_path = os.path.join(output_base, pred_class,
os.path.basename(paths[i]))
                shutil.copy(paths[i], dst_path)

    # === METRICS ===
    print("\n=== METRICS ===")
    acc = accuracy_score(all_labels, all_preds)
```

```
    report = classification_report(all_labels, all_preds,
target_names=class_names, output_dict=True)
    matrix = confusion_matrix(all_labels, all_preds)

    # Overall metrics
    print(f"Overall Accuracy: {acc:.4f}")
    print(f"Precision (macro avg): {report['macro avg']['precision']:.4f}")
    print(f"Recall (macro avg): {report['macro avg']['recall']:.4f}")
    print(f"F1 Score (macro avg): {report['macro avg']['f1-score']:.4f}")

    # === PER-CLASS METRICS ===
    print("\n=== PER-CLASS METRICS ===")
    total_per_class = np.zeros(len(class_names))
    correct_per_class = np.zeros(len(class_names))

    for i in range(len(all_labels)):
        total_per_class[all_labels[i]] += 1
        if all_labels[i] == all_preds[i]:
            correct_per_class[all_labels[i]] += 1

    for i, class_name in enumerate(class_names):
        precision = report[class_name]['precision']
        recall = report[class_name]['recall']
        f1 = report[class_name]['f1-score']
        acc_cls = correct_per_class[i] / total_per_class[i] if
total_per_class[i] > 0 else 0.0
        print(f"Class: {class_name}")
        print(f"  Accuracy:  {acc_cls:.4f}")
        print(f"  Precision: {precision:.4f}")
        print(f"  Recall:    {recall:.4f}")
        print(f"  F1 Score:  {f1:.4f}")

    # Save confusion matrix
    cm_path = os.path.join(output_base, 'confusion_matrix.png')
    plot_confusion_matrix(matrix, class_names, cm_path)
    print(f"\nConfusion matrix saved to: {cm_path}")

    # Model info
    print(f"\nTotal parameters: {count_parameters(model):,}")
    print(f"Model file size: {get_model_size(args.model_dir):.2f} MB")

if __name__ == '__main__':
    main()
```

ResNet20

Consist of:

- AdderNet.ipynb (use back the same .ipynb file from AdderNet model)
- resnet20.py
- main.py (use back the same main.py from AdderNet model)
- test.py (use back the same test.py from AdderNet model)

resnet20.py

```
import torch.nn as nn


def conv3x3(in_planes, out_planes, stride=1):
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,
padding=1, bias=False)


class BasicBlock(nn.Module):
    expansion=1

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(BasicBlock, self).__init__()
        self.conv1 = conv3x3(inplanes, planes, stride = stride)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = conv3x3(planes, planes)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out
```

```
class ResNet(nn.Module):

    def __init__(self, block, layers, num_classes=2):
        super(ResNet, self).__init__()
        self.inplanes = 16
        self.conv1 = nn.Conv2d(3, 16, kernel_size=7, stride=2, padding=3,
bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self._make_layer(block, 16, layers[0])
        self.layer2 = self._make_layer(block, 32, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 64, layers[2], stride=2)
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Conv2d(64 * block.expansion, num_classes, 1, bias=False)
        self.bn2 = nn.BatchNorm2d(num_classes)

        for m in self.modules():
            if isinstance(m, nn.BatchNorm2d):
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, block, planes, blocks, stride=1):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.inplanes, planes * block.expansion,
kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion)
            )

        layers = []
        layers.append(block(inplanes = self.inplanes, planes = planes, stride
= stride, downsample = downsample))
        self.inplanes = planes * block.expansion
        for _ in range(1, blocks):
            layers.append(block(inplanes = self.inplanes, planes = planes))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.layer1(x)
        x = self.layer2(x)
```

```
        x = self.layer3(x)

        x = self.avgpool(x)
        x = self.fc(x)
        x = self.bn2(x)

        return x.view(x.size(0), -1)


def resnet20(num_classes=4, **kwargs): # num_classes = 2 or 4 or 5
    return ResNet(BasicBlock, [3, 3, 3], num_classes=num_classes, **kwargs)
```

HybridNet

Consist of:

- AdderNet.ipynb (use back the same .ipynb file from AdderNet model)
- hybrid.py
- main.py (use back the same main.py from AdderNet model)
- test.py (use back the same test.py from AdderNet model)

hybrid.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F

# ------------------------
# MobileNetV2 Inverted Residual Block
# ------------------------
class InvertedResidual(nn.Module):
    def __init__(self, inp, oup, stride, expand_ratio):
        super(InvertedResidual, self).__init__()
        hidden_dim = int(inp * expand_ratio)
        self.use_res_connect = (stride == 1 and inp == oup)

        layers = []
        if expand_ratio != 1:
            # pointwise
            layers.append(nn.Conv2d(inp, hidden_dim, 1, 1, 0, bias=False))
            layers.append(nn.BatchNorm2d(hidden_dim))
            layers.append(nn.ReLU6(inplace=True))
        # depthwise
        layers.extend([
            nn.Conv2d(hidden_dim, hidden_dim, 3, stride, 1, groups=hidden_dim,
bias=False),
            nn.BatchNorm2d(hidden_dim),
            nn.ReLU6(inplace=True),
            # pointwise-linear
            nn.Conv2d(hidden_dim, oup, 1, 1, 0, bias=False),
            nn.BatchNorm2d(oup),
        ])
        self.conv = nn.Sequential(*layers)

    def forward(self, x):
        if self.use_res_connect:
            return x + self.conv(x)
        else:
            return self.conv(x)
```

```
# -------------------------
# Hybrid Block: ResNet shortcut + MobileNetV2 inverted residual
# -------------------------
class HybridBlock(nn.Module):
    def __init__(self, in_planes, out_planes, stride=1, expand_ratio=6):
        super(HybridBlock, self).__init__()
        self.inverted_residual = InvertedResidual(in_planes, out_planes,
stride, expand_ratio)
        self.shortcut = nn.Sequential()
        if stride != 1 or in_planes != out_planes:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_planes, out_planes, kernel_size=1, stride=stride,
bias=False),
                nn.BatchNorm2d(out_planes)
            )

    def forward(self, x):
        out = self.inverted_residual(x)
        out += self.shortcut(x)
        return F.relu(out)

# -------------------------
# HybridNet (ResNet20 + MobileNetV2 ideas)
# -------------------------
class HybridNet(nn.Module):
    def __init__(self, num_classes=4):
        super(HybridNet, self).__init__()
        self.stem = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1, bias=False),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True)
        )

        # Stages: (like ResNet-20, but each uses HybridBlock)
        self.layer1 = self._make_layer(16, 24, num_blocks=2, stride=1)  # like
MobileNet small expansion
        self.layer2 = self._make_layer(24, 48, num_blocks=2, stride=2)
        self.layer3 = self._make_layer(48, 96, num_blocks=2, stride=2)

        self.pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Linear(96, num_classes)

    def _make_layer(self, in_planes, out_planes, num_blocks, stride):
        layers = []
        layers.append(HybridBlock(in_planes, out_planes, stride=stride))
        for _ in range(1, num_blocks):
            layers.append(HybridBlock(out_planes, out_planes, stride=1))
        return nn.Sequential(*layers)
```

```
    def forward(self, x):
        out = self.stem(x)
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.pool(out)
        out = out.view(out.size(0), -1)
        return self.fc(out)
```

Improved HybridNet

Consist of:

- AdderNet.ipynb (use back the same .ipynb file from AdderNet model)
- hybrid.py
- main.py (use back the same main.py from AdderNet model)
- test.py (use back the same test.py from AdderNet model)

hybrid.py

```
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.checkpoint as cp  # why: cut activation memory

# -------------------------
# Helpers
# -------------------------
class DropPath(nn.Module):
    def __init__(self, drop_prob: float = 0.0):
        super().__init__()
        self.drop_prob = float(drop_prob)

    def forward(self, x):
        if not self.training or self.drop_prob == 0.0:
            return x
        keep = 1.0 - self.drop_prob
        shape = (x.shape[0],) + (1,) * (x.ndim - 1)
        return x * x.new_empty(shape).bernoulli_(keep).div_(keep)


class SqueezeExcite(nn.Module):
    def __init__(self, channels: int, se_ratio: float = 0.25):
        super().__init__()
        hidden = max(8, int(channels * se_ratio))
        self.pool = nn.AdaptiveAvgPool2d(1)
        self.fc = nn.Sequential(
            nn.Conv2d(channels, hidden, 1, bias=True),
            nn.ReLU(inplace=True),
            nn.Conv2d(hidden, channels, 1, bias=True),
            nn.Sigmoid(),
        )

    def forward(self, x):
        return x * self.fc(self.pool(x))
```

```
def _make_divisible(v: int, divisor: int = 8) -> int:
    return int(math.ceil(v / divisor) * divisor)

# -------------------------
# MobileNetV2 Inverted Residual Block
# -------------------------
class InvertedResidual(nn.Module):
    """
    expand -> depthwise -> pointwise-linear (+ optional SE).
    Has internal residual only if stride==1 and in==out.
    """
    def __init__(self, inp, oup, stride, expand_ratio, se_ratio: float =
0.25):
        super().__init__()
        assert stride in [1, 2]
        hidden_dim = int(inp * expand_ratio)
        self.use_res_connect = (stride == 1 and inp == oup)

        layers = []
        if expand_ratio != 1:
            layers += [
                nn.Conv2d(inp, hidden_dim, 1, 1, 0, bias=False),
                nn.BatchNorm2d(hidden_dim),
                nn.ReLU6(inplace=True),
            ]
        layers += [
            nn.Conv2d(hidden_dim, hidden_dim, 3, stride, 1, groups=hidden_dim,
bias=False),
            nn.BatchNorm2d(hidden_dim),
            nn.ReLU6(inplace=True),
            nn.Conv2d(hidden_dim, oup, 1, 1, 0, bias=False),
            nn.BatchNorm2d(oup),
        ]
        self.conv = nn.Sequential(*layers)
        self.se = SqueezeExcite(oup, se_ratio) if se_ratio and se_ratio > 0
else nn.Identity()

    def forward(self, x):
        y = self.conv(x)
        y = self.se(y)
        if self.use_res_connect:
            y = x + y
        return y

# -------------------------
# Hybrid Block
# -------------------------
class HybridBlock(nn.Module):
```

```
    """
    Projection shortcut only when shapes differ (avoid double identity).
    Activation checkpointing to save memory on forward.
    """
    def __init__(self, in_planes, out_planes, stride=1,
                 expand_ratio=3.0, se_ratio=0.25, drop_path=0.05,
use_checkpoint=True):
        super().__init__()
        self.irb = InvertedResidual(in_planes, out_planes, stride,
expand_ratio, se_ratio)
        self.use_proj = (stride != 1) or (in_planes != out_planes)
        self.proj = nn.Sequential(
            nn.Conv2d(in_planes, out_planes, 1, stride=stride, bias=False),
            nn.BatchNorm2d(out_planes),
        ) if self.use_proj else None
        self.drop_path = DropPath(drop_path) if drop_path > 0.0 else
nn.Identity()
        self.use_checkpoint = use_checkpoint

    def forward(self, x):
        if self.use_checkpoint and self.training:
            y = cp.checkpoint(self.irb, x)  # why: recompute backward, lower
peak mem
        else:
            y = self.irb(x)
        y = self.drop_path(y)
        if self.use_proj:
            y = y + self.proj(x)
        return F.relu(y, inplace=True)

# -------------------------
# HybridNet
# -------------------------
class HybridNet(nn.Module):
    """
    Backward-compatible: HybridNet(num_classes=2/4/5)
    Lighter defaults to avoid OOM; still higher quality via SE+DropPath.
    """
    def __init__(self, num_classes: int = 4,
                 channels=(24, 48, 96),      # keep widths modest
                 depths=(2, 2, 2),           # safe depth to avoid OOM
                 expand_ratio: float = 3.0,  # lower than 6 to cut activations
                 se_ratio: float = 0.25,
                 p_dropout: float = 0.10,
                 drop_path_rate: float = 0.05,
                 width_mult: float = 1.00,
                 dynamic_downsample: bool = True,  # auto-downsample very
large inputs
```

```
                 ds_threshold: int = 128,          # if max(H,W) >= threshold,
downsample by 2
                 use_checkpoint: bool = True):
        super().__init__()
        self.dynamic_downsample = dynamic_downsample
        self.ds_threshold = ds_threshold
        self.use_checkpoint = use_checkpoint

        c1, c2, c3 = [_make_divisible(int(c * width_mult), 8) for c in
channels]

        self.stem = nn.Sequential(
            nn.Conv2d(3, 16, 3, 1, 1, bias=False),
            nn.BatchNorm2d(16),
            nn.ReLU(inplace=True),
        )

        total_blocks = sum(depths)
        dp_rates = [drop_path_rate * i / max(1, total_blocks - 1) for i in
range(total_blocks)]
        dp_i = 0

        self.layer1, dp_i = self._make_layer(16, c1, depths[0], stride=1,
expand_ratio=expand_ratio,
                                             se_ratio=se_ratio,
dp_rates=dp_rates, dp_i=dp_i)
        self.layer2, dp_i = self._make_layer(c1, c2, depths[1], stride=2,
expand_ratio=expand_ratio,
                                             se_ratio=se_ratio,
dp_rates=dp_rates, dp_i=dp_i)
        self.layer3, dp_i = self._make_layer(c2, c3, depths[2], stride=2,
expand_ratio=expand_ratio,
                                             se_ratio=se_ratio,
dp_rates=dp_rates, dp_i=dp_i)

        self.pool = nn.AdaptiveAvgPool2d(1)
        self.drop = nn.Dropout(p_dropout) if p_dropout and p_dropout > 0 else
nn.Identity()
        self.fc = nn.Linear(c3, num_classes)

        self._init_weights()

    def _make_layer(self, in_planes, out_planes, num_blocks, stride,
expand_ratio, se_ratio, dp_rates, dp_i):
        layers = []
        layers.append(HybridBlock(in_planes, out_planes, stride=stride,
                                  expand_ratio=expand_ratio,
se_ratio=se_ratio,
```

```
                                    drop_path=dp_rates[dp_i] if dp_rates else
0.0,
                                    use_checkpoint=self.use_checkpoint))
            dp_i += 1
            for _ in range(1, num_blocks):
                layers.append(HybridBlock(out_planes, out_planes, stride=1,
                                          expand_ratio=expand_ratio,
se_ratio=se_ratio,
                                          drop_path=dp_rates[dp_i] if dp_rates
else 0.0,
                                          use_checkpoint=self.use_checkpoint))
                dp_i += 1
            return nn.Sequential(*layers), dp_i

        def _init_weights(self):
            for m in self.modules():
                if isinstance(m, nn.Conv2d):
                    nn.init.kaiming_normal_(m.weight, mode='fan_out',
nonlinearity='relu')
                elif isinstance(m, nn.Linear):
                    nn.init.normal_(m.weight, 0, 0.01); nn.init.zeros_(m.bias)
                elif isinstance(m, nn.BatchNorm2d):
                    nn.init.ones_(m.weight); nn.init.zeros_(m.bias)

        def forward(self, x):
            # optional dynamic downsample for big inputs (e.g., 224x224)
            if self.dynamic_downsample and max(x.shape[-2], x.shape[-1]) >=
self.ds_threshold:
                x = F.avg_pool2d(x, 2)  # why: halve H,W early → big mem drop
            x = self.stem(x)
            x = self.layer1(x)
            x = self.layer2(x)
            x = self.layer3(x)
            x = self.pool(x).flatten(1)
            x = self.drop(x)
            return self.fc(x)
```

POSTER

Comparing Machine Learning Techniques to Segmentize and Classify Tongue Regions for Traditional and Complementary Medicine (TCM) Diagnosis

By Bong Min Xuan • Supervisor: Dr. Lee Wai Kong
Bachelor of Information Systems (Honours) Informaton Systems Engineering

Introduction

- Traditional Chinese Medicine (TCM) relies on tongue diagnosis, but manual inspection is **subjective and inconsistent**.
- Aim: **Automate segmentation and classification** of tongue images using machine learning and deep learning to achieve **objective, reproducible diagnostics**.
- Challenges: complex backgrounds, subtle color/coating variations, and computational efficiency.

Objectives

- Compare **SVM vs. DuckNet** for segmentation.
- Design and evaluate classification models: **AdderNet → ResNet20 → HybridNet → Improved HybridNet**.
- Benchmark across **binary (2-class), 4-class, and 5-class datasets**.
- Identify the **most effective model** balancing accuracy, efficiency, and scalability.

Methodology

Datasets:

- **2**-class (Stained vs. Non-stained moss, 2000 images).
- 4-class (Pale, Pale Red, Red, Bluish Purple; 2000 images).
- 5-class (Mirror-Approximated, White-Greasy, Thin-White, Yellow-Greasy, Grey-Black; 2500 images).

Segmentation:

- **S**VM (traditional baseline, lightweight).
- DuckNet-17 (deep learning, high robustness; adopted as default).

Classification Models:

- AdderNet (experimental, addition-based ops).
- ResNet20 (baseline residual CNN).
- HybridNet (ResNet + MobileNetV2 efficiency).
- Improved HybridNet (adds SE, DropPath, checkpointing).

Metrics: Accuracy, Precision, Recall, F1-score, Jaccard (segmentation), Training time, Model size.

1

2

3

Results

Segmentation:

- sVM: ~86% accuracy, struggles with complex backgrounds.
- DuckNet: ≥98% accuracy, ≥95% Jaccard; chosen for all main experiments.

Classification:

- 2-class: Improved HybridNet best (97.25%, 0.57 MB).
- 4-class: AdderNet highest (91%) but inefficient; Improved HybridNet balanced (87.75%, 0.57 MB).
- 5-class: ResNet20 (85.8%), HybridNet (86.8%), Improved HybridNet (85.6%) — all competitive, but Improved HybridNet is most efficient.

Discussion

- **Segmentation quality ≠ direct impact on classification** → once tongue is isolated, classifiers dominate performance.
- **AdderNet**: promising but computationally heavy and unstable for >4 classes.
- **ResNet20**: stable benchmark, fast training, but limited scalability.
- **HybridNet**: fewer parameters, good efficiency, slightly weaker for complex tasks.
- **Improved HybridNet**: best trade-off (accuracy + efficiency), small model size, practical for deployment.

Conclusion

- Demonstrated a **reproducible pipeline** for segmentation + classification of tongue images.
- **DuckNet** is superior for segmentation, ensuring reliable preprocessing.
- **Improved HybridNet** consistently provided the **best balance** of accuracy, speed, and model size.
- Project shows the potential of AI to **standardize TCM tongue diagnosis** for future clinical applications.