PALETTE Project Final Report - ESE 3600

Ahmed Muharram, Kidus Seyoum

December 12, 2024

1 Introduction

Color blindness, affecting approximately **350 million** people worldwide, presents challenges in everyday activities that rely on accurate color identification. Traditional solutions like color-blind glasses are only applicable to a subset of cases (e.g., for individuals with protanopia). Our project, **PALETTE (Personalized Assistant for Labeling, Evaluating, and Translating Tints and Effects)**, addresses this problem by leveraging the power of **TinyML and keypoint detection** to provide real-time color identification for people with various forms of color vision deficiency.

With a focus on efficient **model development, data collection, and hardware integration** using an **Android application**, PALETTE serves as a model for future assistive devices and educational/artistic tools.

2 Motivation

Color blindness affects an estimated **8% of men and 0.5% of women globally**, which means millions of individuals face challenges distinguishing and recognizing colors in their day-to-day lives. This condition impacts various aspects of life, from personal tasks like choosing clothes and identifying safety signs to professional activities like selecting items in a store or interpreting visual information during painting.

Existing solutions, such as **EnChroma** glasses, only address a small portion of the colorblind population, particularly those with **protanopia** (red-green color blindness). Our solution aims to enhance independence by helping users confidently identify and interpret colors across different environments. PALETTE offers an intuitive interface, making it easier for colorblind individuals to interact with their surroundings.

This project seeks to **bridge the gap in accessibility** and empower individuals with color vision deficiency by providing a personalized and easy-to-use tool for everyday color identification. Whether for fashion, art, or other personal choices, our project **enhances the quality of life for its users** by providing reliable, detailed color information.

3 Dataset Details

Raw data: Our raw dataset consists of **6,181 4:3 images** before preprocessing. These images represent a variety of scenarios, including diverse pointing gestures directed at objects with different colors, sizes, and complexities, taken by both of us. We started with 1200 images and augmented the rest, using rescaling, rotation, shifting, zooming and horizontal flipping. Each image is labeled based on the normalized fingertip's location in the photo (between 0 and 1) to account for the fact that the raw dataset might have different sizes that still maintain the 4:3 ratio.

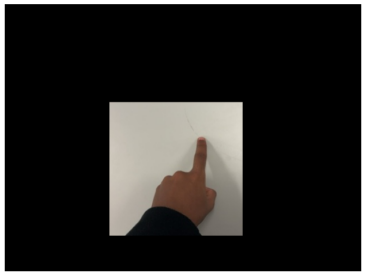
Preprocessed data: After preprocessing, the dataset was reduced to **3,889 images**. The preprocessing pipeline included the following steps:

Data Cleaning: Removing non-4:3 photos, mislabeled images, and to ensure the integrity of the dataset. This eliminated several images, but nothing notable.

Resizing: Standardizing image dimensions to fit 640x480 and change to PNG for uniformity. Normalization: Adjusting pixel values to improve model training efficiency.

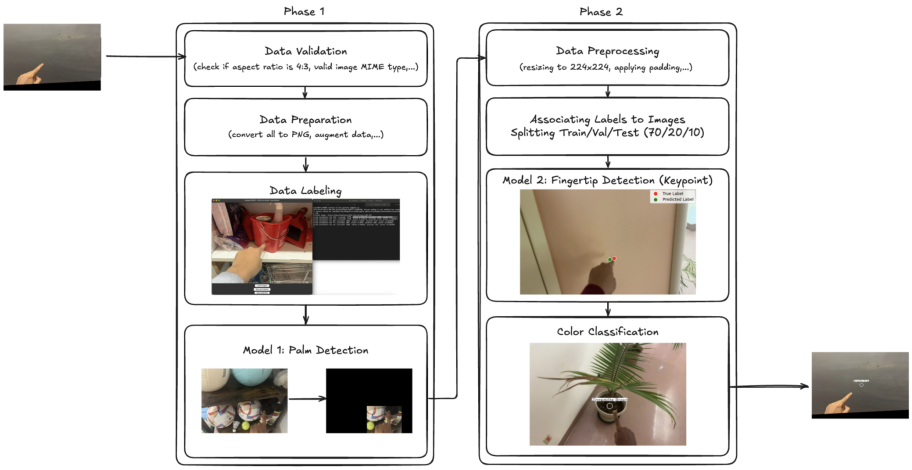
Data labeling: Let's be honest: there was no way to label 6,181 images by hand. To help this very tedious process, we wrote a Python script that, upon clicking on the fingertip in the image, appends to a JSON file the image name and the normalized (x, y) coordinates. This saved us a lot of time, but data labeling took a lot of time still. We will be providing all of the code, dataset, and labels on GitHub in spirit of open-source contribution to TinyML advancement! Please find the GitHub links in the last section.

Palm Detection: In one of the iterations, we were using a model provided in the MediaPipe library that we used for palm detection to help improve our accuracy. This model was too large for the XIAO, so we had to get rid of it. More will be discussed in the next section. We decided to continue use this model as a preprocessing step for the training data: if an image does not get detected as a hand by the palm detection model, it gets filtered out of the dataset. This dropped our size to 3,889 images: almost 3/5ths the size! T



Padding and resizing: We apply padding to make the image squared and resize it again.

Overall, here's what our flow looks like. Note that we have changing the resizing images to 112x112 instead of 224x224 to decrease the model size.

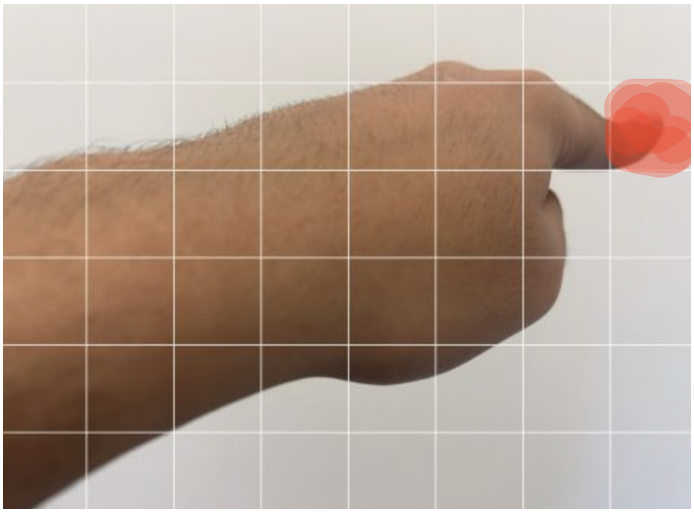


4 Model Choice/Design

To explain how we reached our final model design, let's go through the iterations we have made throughout the project:

Iteration 1: Grids

Our first idea was to divide the image into 8x6 grids and attempt to predict the grid location of the fingertip, then take the average of all the colors in that spot. Here's an example:



This failed because it was hard for the CNN we trained to understand what the grid is and that it has to predict integer grid coordinates. We could have pursued this further, but it was tedious to collect a large enough dataset for this method and to train a good enough model. Furthermore, the average color would have accounted for the fingertip color too, which would not be accurate.

Iteration 2: Transfer Learning

We attempted to use transfer learning by having MobileNetV2 trained on imagenet weights as our feature extractor. This yielded better results, but was making a lot of mistakes and the MSE was really high, so we did not pursue this further.

Iteration 3: MediaPipe + Edge Detection

We also tried to use MediaPipe's hand detection algorithms, CV2's gradient magnitude algorithm for edge detection, and many other algorithms. All of these either failed, were too simple to fit the project requirements, required LiDAR sensors, or were too large to fit on the XIAO.

Iteration 4: MediaPipe's Palm Detection Model + Custom CNN
This was almost perfect: we reached a very low MSE and we went through many stages of choosing a good loss function (MSE, MAE, RMSE, custom loss with regularization, and finally, this:

```
def loss_function(y_true, y_pred):
    # Compute the squared difference between true and predicted
    # coordinates
    square_diff = tf.math.squared_difference(y_true, y_pred)

    # Sum squared differences across coordinates (x and y)
    coordinate_loss = tf.reduce_sum(square_diff, axis=-1)

    # Compute the mean loss across the batch
    coordinate_loss = tf.reduce_mean(coordinate_loss)

    return coordinate_loss
```

This worked really well...Except, **the sum of the model sizes were 88MBs**! There was no way this would have fit on the XIAO. To remedy the situation, we looked into many solutions and ended up learning Kotlin in 2 days to create an Android app customized for PALETTE. Now, the challenge was to reduce the size. After going to office hours, attempting to use a U-Net architecture, and attempting to extract layers from the Palm Detection model, we ended up creating our own CNN with regularization, batch normalization, and many other features:

Current Iteration

Here's our current model:

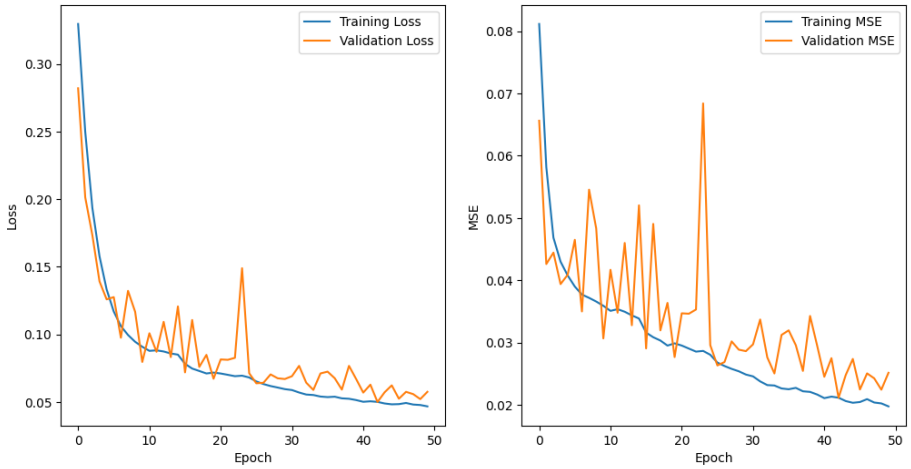
```
model = models.Sequential([
    layers.Conv2D(16, (3, 3), activation='leaky_relu',
                  kernel_regularizer=tf.keras.regularizers.L2(0.001),
                  input_shape=(112, 112, 3)),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),

    layers.Conv2D(32, (3, 3), activation='leaky_relu',
                  kernel_regularizer=tf.keras.regularizers.L2(0.001)),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),
    layers.Dropout(0.3),
```

```
    layers.Conv2D(64, (3, 3), activation='leaky_relu',
                  kernel_regularizer=tf.keras.regularizers.L2(0.001)),
    layers.BatchNormalization(),
    layers.MaxPooling2D((2, 2)),

    layers.GlobalMaxPooling2D(),

    layers.Dense(512, kernel_regularizer=tf.keras.regularizers.L2(0.001)),
    layers.BatchNormalization(),
    layers.Dropout(0.5),
    layers.LeakyReLU(),
    layers.Dense(2, activation='sigmoid')
])
```

We use 3 batches of layers (Conv2D with l2 regularization, batch norm., then max pooling), apply GlobalMaxPooling (which dramatically reduces the size), pass through a dense layer with regularization followed by more batch normalization and dropout for overfitting protection, and finally predict the x,y coordinates. This is no longer a classification problem like the first iteration, but instead we do regression. We use LeakyReLU because it proved better than ReLU.

5 Model Training and Evaluation Results

We trained our model for 50 epochs with the Adam optimizer (lr=1e-3, β_1=0.9, β_2=0.999, ϵ=1e-10, decay=0) and an LR scheduler on the validation loss using our custom loss function defined above. We ended up with a loss of 0.0469 (val: 0.0575) and MSE of 0.0198 (val: 0.0251).



(note that accuracy is not a valid metric for us in this question, since we are doing regression on keypoints)

We can notice a lot of spikes and noise, but the overall trend is going down. The model is performing worse than the large 88MB model, but given the huge drop in size (99.7%+ reduction!), it was definitely worth it.

6 Deployment/Hardware Details

Our project's deployment strategy centered around accessibility, practicality, and user convenience. Initially, we aimed to deploy our model on the XIAO ESP32-S3, a compact and powerful microcontroller known for its compatibility with TinyML applications and the one provided in class. However, we encountered several challenges during the integration process, prompting us to explore alternative deployment platforms. Below, we detail our approach and final deployment solution.

6.1 Initial Deployment on XIAO ESP32-S3

The XIAO ESP32-S3 was chosen due to its small form factor, integrated Wi-Fi/Bluetooth capabilities for final result display, and compatibility with TensorFlow Lite Micro. The deployment process included the following steps:

- Model Conversion: We converted our trained TensorFlow model to TensorFlow Lite format and further optimized it for TensorFlow Lite Micro.
- Model shirinking: To ensure the model's fit within the XIAO's memory constraints, we applied techniques like GlobalMaxPooling. This reduced the model size significantly while maintaining a good enough accuracy.
- Firmware Development: Using VSCode and PlatformIO, we developed firmware to load and run the model on the XIAO. This included preprocessing image inputs, running inference, and outputting results with MicroPrintf for debugging.
- Testing and Optimization: We iteratively tested the model on the hardware, identifying bottlenecks in memory usage and inference speed.

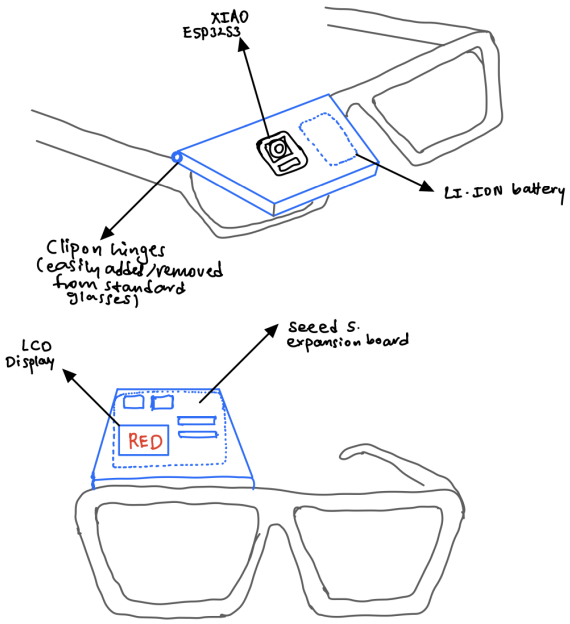
Unfortunately, despite significant effort and countless hours spent, the deployment on the XIAO ESP32-S3 faced several roadblocks:

- Performance Issues: Inference times were much higher than acceptable for real-time applications, primarily due to limited computational resources.
- Hardware Malfunctions: During testing, the XIAO suffered a hardware failure, rendering it inoperable.

6.2 Final Deployment on Android

Given the challenges with the XIAO, we pivoted to deploying the model on an Android platform. This approach offered several advantages:

- User-Friendly Interface: Developing a dedicated Android application allowed us to create an intuitive GUI for users to interact with the model, which works much better than having to wear glasses and look into an LCD screen as our initial proposed design looked.



- Rapid Iteration: Using Android Studio and Kotlin, we quickly iterated on features such as camera integration, real-time fingertip detection, and color identification like the Colab notebook does.

The Android app was designed with the following features:

- Camera Input: Captures a photo for fingertip detection.
- Color Identification: Identifies and displays the color at the fingertip's location in real time. Has a library of 5955 supported color names.
- Offline Functionality: Runs entirely offline to ensure privacy and reduce latency.

Deploying on Android significantly enhanced the project's accessibility and usability, making it a robust solution for end users while retaining the essence of TinyML principles.

7 Challenges and Future Work

7.1 Challenges

Throughout the development of PALETTE, we encountered several challenges between data collection, model development, and hardware integration:

- Data Collection and Labeling: Building a robust dataset was labor-intensive. Despite automation tools, manually labeling 6,181 images was time-consuming and prone to human error.
- Model Optimization: Balancing model accuracy with size constraints was challenging, especially for deployment on resource-constrained devices like the XIAO. Architecture redesign required significant experimentation, with models like U-Net taking hours of reiteration before eventually failing.
- Hardware Limitations: The XIAO's limited memory and processing power restricted our ability to deploy a fully functional model. Hardware malfunctions further added to the frustration and the delays.
- Transition to Android: Learning Kotlin and developing an Android app within a short timeframe required steep learning curves and very, very rapid prototyping.
- Performance Trade-offs: Reducing model size resulted in a slight loss of accuracy compared to the larger 88MB model, but it definitely was worth it.

7.2 Future Work

Despite the progress made, there are several areas for improvement and potential extensions for PALETTE:

- Improved Model Efficiency: Further optimize the model using techniques like pruning, using LiDAR + depth sensors, and custom advanced architectures to enable deployment on microcontrollers.
- Cross-Platform Deployment: Extend support to other platforms, such as iOS, to reach a broader audience.
- Enhanced Color Detection: Improve color detection accuracy by adding contextual information (e.g., lighting conditions, tint, contrast...) and advanced color correction algorithms.

- Accessibility Features: Incorporate voice-guided feedback using KWS and other techniques.
- Artistic Extensions: Develop features tailored for artists, such as suggesting color palettes or providing color theory insights.
- Visual Wake Word System: Leverage the lightweight model as a visual wake word detector in a cascade system, triggering more complex models like SAM for detailed analysis and processing.

8 Sources

8.1 Code

Colab for the data preprocessing and the models: Link

Android deployment GitHub link: Link

8.2 Datasets

Download link for the raw dataset: Download

Download link for the legacy palm extraction customized model: Download

Download link for the filtered hand data extracted from the previous model: Download

8.3 Labels

Download link for the labels: Download