DRAFT

Project Needs/Objectives/Concept

Requirements and constraints

Design Tradeoffs

Prototyping/Testing

Project Results

Poster Aesthetics

* NEED/OBJECTIVE OF THE SYSTEM, COMPONENT, OR PROCESS Score (1-4):
* PROJECT PLANNING Score (1-4): Exhibits a professional level of planning and time management
* TECHNICAL REQUIREMENTS, COSTRAINTS, AND TRADE-OFFS (1-4): Identified/ realistic technical, economic, safety and environmental constraints and performed trade-off analysis
* CONCEPT GENERATION AND SELECTION Score (1-4): Considered more than one technically feasible alternative solution, compared the alternative solutions, and recommended one of the solutions based on well-defined criteria
* PROTOTYPING, TESTING, AND VALIDATION Score (1-4): An iterative prototype and redesign process was utilized .Final prototype met all of the requirements and constraints. The design was adequately validated
* INNOVATION: PROJECT IDEA AND APPROACH Score (1-4): Developed a novel solution to a new problem that may be patentable and commercialized
  + POSTER AESTHETICS AND PRESENTATION Score (1-4): All team members contribute to discussion. Lessons learned are clearly articulated. Overall design is pleasing and harmonious. Creative poster design
  + PROFESSIONALISM: DISPLAYED GOOD ATTITUDE, BE ON TIME, LISTEN RESPECTFULLY, ETC. Score (1-4): Team shows excellent professionalism

Soham Naik

# Real-Time Object Detection Pipeline for Mobile Vision Systems

## Section Structure

* **Problem Statement**
* **Requirements Gathering & Project Scope**
* **Tech Stack**
* **Workflow Overview**
* **Technical Implementation**
* **Performance Analysis & Results**
* **Challenges, Solutions & Lessons Learned**
* **Future Improvements & Reflections**

## Abstract

This work focuses on the development and integration of a real-time object detection system optimized for mobile deployment. The system utilizes a lightweight, custom-trained YOLOv11n model for detecting an LED-based signal source in dynamic lighting conditions. The project involved building a complete vision processing pipeline—from data collection and preprocessing, through model training and optimization, to real-time deployment on a Kotlin-based Android application. The research included comparative analysis of detection models, extensive testing using OpenCV and PyTorch, and training acceleration using CUDA-enabled HPC infrastructure. Solutions were developed to mitigate noise, improve inference latency, and ensure reliable signal decoding using mathematical techniques. This work highlights the process of translating a research-grade model into a performant, embedded application suitable for real-world use.

## Problem Statement

Develop a mobile application capable of stabilizing and smoothing finger gestures captured in real-time using a smartphone’s rolling shutter camera. The system will reconstruct 3D finger movements using an LED-equipped glove and a YOLO-based object detection model. To ensure gesture clarity and user privacy, the application will leverage the shutter effect to filter out irrelevant background data and apply image processing techniques to reduce noise and hand tremors, resulting in a refined, interpretable gesture path.

## Requirements Gathering & Project Scope

Our client, Dr. Zhang, clearly outlined the scope and technical goals of the project. The objective was to demonstrate the feasibility of his research approach on a mobile edge device, ultimately paving the way for real-world use.

The project consists of the following key components:

1. **LED-Equipped Glove**:  
    A glove worn by the user containing multiple LEDs—one on each fingertip and one on the palm. Each LED is modulated with a unique PWM frequency, creating distinguishable patterns for each finger.
2. **Smartphone with Rolling Shutter Camera**:  
    The phone camera serves two purposes:
   * **Privacy Filtering**: The rolling shutter effect helps filter out background content, enhancing user privacy.
   * **Pattern Capture**: The shutter captures the unique PWM patterns emitted by each LED, allowing pattern-based detection and identification.
3. **YOLO-Based Object Detection**:  
    A custom-trained YOLO model detects and classifies each LED based on its unique visual pattern. This enables the system to track multiple LEDs (and therefore users) simultaneously.
4. **Multi-User Support**:  
    A hard requirement was to train the YOLO model to recognize different LED patterns, enabling tracking of multiple users or hands in parallel.
5. **3D Path Reconstruction**:  
    By capturing the position of each LED over time, the system reconstructs the 3D path of finger and hand movements.
6. **Gesture Smoothing & Stabilization**:  
    A critical feature is the application of smoothing techniques to reduce noise, mitigate hand tremors, and refine the reconstructed gesture paths.
7. **Edge Optimization**:  
    As real-time object detection on mobile hardware is computationally demanding, part of the project involved exploring optimizations to make the system performant and responsive on smartphones.

Collecting the requirements involved deep engagement with Dr. Zhang’s research, understanding both the theoretical underpinnings and the practical constraints of mobile development. The goal was not only to replicate the research setup, as demonstrated in Dr. Zhang’s work [insert link or title here], but to push it further—proving its viability on consumer-grade mobile hardware.

## Tech Stack

* **Android Studio (Kotlin)** – Developed the mobile application with real-time camera access and inference display.
* **LiteRT** – TensorFlow support library used for efficient real-time execution of TFLite models on mobile devices.
* **TensorFlow Lite** – Deployed trained YOLO models for on-device inference with minimal latency.
* **TensorFlow GPU & Support** – Enabled GPU acceleration during model testing and inference benchmarking.
* **PyTorch / TorchVision** – Core framework for model development, training, and architecture customization.
* **Ultralytics YOLO (YOLOv11n)** – Lightweight object detection model used for training and deployment, optimized for mobile environments.
* **OpenCV 2** – Used extensively for frame extraction, preprocessing, contour detection, and post-processing tasks.
* **NumPy** – Handled image matrix operations, data manipulation, and signal decoding logic.
* **Apache Commons Math** – Applied numerical methods and signal processing techniques for decoding LED signals.
* **CUDA** – Leveraged for GPU-accelerated training and performance testing of deep learning models.
* **VSCode** – Primary development environment for Python-based research, preprocessing scripts, and training pipelines.
* **FFmpeg** – Extracted video frames and converted formats during data collection and labeling stages.
* **HPC (GreatLakes)** – Used for high-performance distributed training of YOLOv11n models with large video datasets.

## Workflow Overview

1. **Data Collection & Labeling**
2. **Model Training**
3. **Model Testing**
4. **Model Conversion**
5. **Xamera Mobile Application Integration**
   1. **Frame Pre-Processing**
   2. **Model Inference**
   3. **Path Smoothing**

**1. Data Collection**

To ensure consistency between training data and inference data, all video recordings were collected directly from the mobile application. This was an intentional design choice—any preprocessing applied to the frames during inference must match exactly what was applied during training. Capturing and preprocessing video within the mobile app eliminates inconsistencies that might arise from using a separate data pipeline.

Once recorded, the video was transferred from the mobile device to a laptop. There, I used **FFmpeg** to extract individual frames from the video. These frames served as the raw input for the labeling process and subsequent model training.

### **2. Labeling**

YOLO requires a strict format for training data. Each image must be paired with a label file containing a tight bounding box around each object of interest—typically a single LED in our case. Poorly fitted boxes (too loose or too tight) can significantly degrade detection performance.

To handle this at scale (over 3,000 frames), I built a Python script using **OpenCV** and **NumPy**, leveraging our key advantage: a blacked-out background (thanks to the shutter effect) and bright LED patterns in the foreground.

**Automated Labeling Workflow:**

1. **Brightness Thresholding**:  
    All pixels below a brightness threshold were turned to pure black (0,0,0), isolating the LED light patterns.
2. **Bounding Box Generation**:  
    NumPy was used to locate all non-black pixels and calculate the tightest minimum-area bounding box around them.

**Format Conversion**:  
 The bounding box was converted into the YOLO-required format:  
  
 <class\_id> <x1> <y1> <x2> <y2>

1. Each label file (e.g., a.txt) corresponded directly to its image frame (e.g., a.png), ensuring proper alignment between images and labels.
2. **Quality Control**:  
    A version of each image with the bounding box drawn was generated for quick visual review. I manually skimmed through the dataset and removed any frames with incorrect or noisy annotations.

### **Model Training**

Our client, Dr. Zhang, provided us with full access to the necessary resources to complete this project—most notably, the **University of Michigan’s High Performance Computing (HPC)** cluster. This HPC system allowed us to leverage up to **four NVIDIA Tesla H100 GPUs**, enabling the training of complex models like YOLO in a matter of minutes.

#### **Dataset Preparation**

Before training a YOLO model, the dataset must be properly structured and split into two distinct sets: **training** and **validation**. The standard best practice is an **80/20 split**, where:

* **80%** of the images are used for training, helping the model learn the underlying features.
* **20%** are reserved for validation, used to evaluate the model’s performance on **previously unseen data**.

This separation is critical for ensuring that the model generalizes well and is not overfitting to the training data.

#### **HPC Environment Setup**

The HPC cluster runs **Python 3.12** and **CUDA 12.4**, ensuring compatibility with the latest versions of deep learning frameworks. A virtual environment was created with all necessary packages, including **PyTorch** and **Ultralytics YOLO**.

#### **Training Configuration**

Two key configuration files were created:

1. **.yaml Dataset Configuration File**:  
    This defines the paths to the training and validation datasets, the number of classes, and their labels.
2. **slurm Job Script**:  
    This script specifies the YOLO model configuration, training parameters (epochs, batch size, image size, etc.), and hardware requirements (e.g., number of GPUs, allocated memory). It handles the job submission to the SLURM scheduler.

Once the SLURM script is submitted, training is automatically queued and executed on the allocated GPUs. Thanks to the computational power of the H100s, training completes rapidly—even with large datasets.

### **Model Testing**

After training is complete, the model’s performance must be thoroughly evaluated before integration into the mobile app. This evaluation is conducted in two main phases: **quantitative assessment** using training metrics, and **qualitative testing** through real-world inference.

#### **Quantitative Evaluation (Training Metrics)**

The YOLO training process (using Ultralytics) automatically provides detailed logs and visualizations that track the model’s performance over each epoch. These include:

* **Mean Average Precision (mAP)**
* **Precision and Recall**
* **Loss curves (classification, localization, objectness)**
* **Training vs. validation accuracy over time**

These metrics help us assess whether the model is:

* **Overfitting** – performing well on training data but poorly on validation data (model has memorized the dataset).
* **Underfitting** – performing poorly on both sets (model hasn’t learned meaningful features).
* **Generalizing well** – showing solid performance across unseen validation data.

Graphs provided by Ultralytics make it easy to visualize learning trends, convergence, and potential drop-offs. Based on this analysis, we can adjust the number of epochs or fine-tune other hyperparameters to retrain the model for better performance.

#### **Qualitative Evaluation (Local Inference)**

Once ideal training metrics are achieved, the model is exported and tested locally using a Python development script. At this stage, videos and images collected from the mobile app are used as test inputs. Inference is run using both:

* **CPU-based inference** (for baseline comparison)
* **GPU-based inference via CUDA** (to assess real-time performance)

This phase answers two key questions:

1. **Is inference fast enough** given our hardware constraints (e.g., laptop or mobile device)?
2. **Are the detection results visually accurate** from a human perspective?

Only after passing both the metric-based and real-world testing stages is the model deemed ready for mobile integration.

## Technical Implementation

**Standard Android**

* **Kotlin:**We developed the app using Kotlin, which provided modern language features and robust support for Android. However, integrating low-level components—such as the Camera2 API for real-time processing—required more detailed, manual coding compared to higher-level abstractions.
* **XML:**User interfaces were built with XML, offering a consistent design framework. Integrating these layouts with dynamic camera settings and custom controls demanded precise coordination between XML and Kotlin code.

**Augmented Reality**

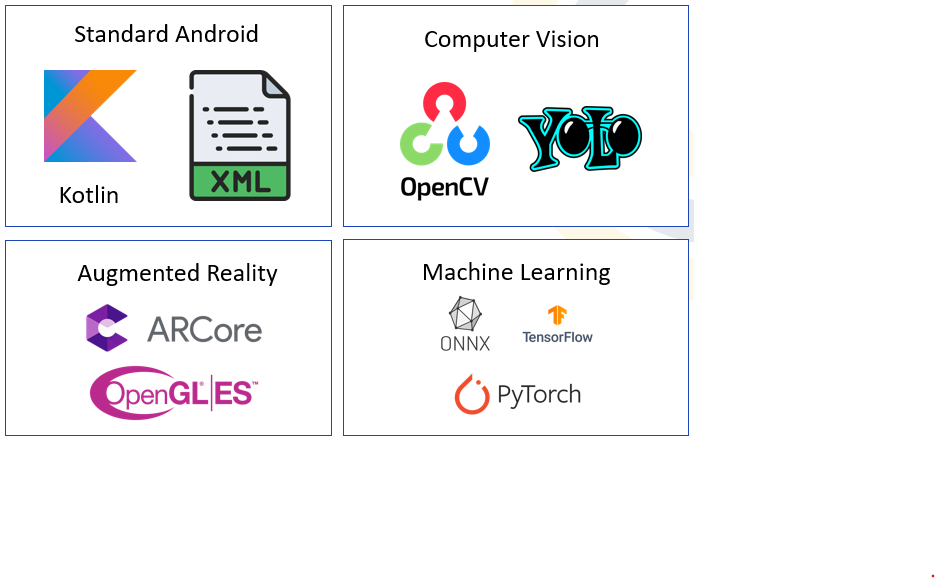
* **Google ARCore:**Google ARCore is utilized to provide an augmented reality experience within Xamera. In the final processing step, ARCore generates 3D paths or 3D letter boxes that simulate the drawn digit or letter, or displays the last drawn path if the user opts not to make a call or send an email. This functionality was crucial for validating our research by ensuring that the generated path is accurately smoothed and correctly inferred..
* **OpenGL ES:**For rendering immersive 3D paths and 3D boxes within the AR environment, we utilized OpenGL ES. This involved reverse-engineering existing samples and ensuring smooth, real-time integration with the main application.

**Computer Vision**

* **OpenCV:**OpenCV, a widely used computer vision library, is a core component of Xamera. After YOLO detects the LED light source, OpenCV is used to create lines around it through techniques like Kalman filtering, spline interpolation, and calculations using the Apache Commons Math library. The resulting smooth path is then processed by either Google ARCore for generating 3D paths or letter boxes, or by TensorFlow for digit or letter inference.
* **YOLO:**YOLO, short for "You Only Look Once," is a pre-trained machine learning model designed for rapid image detection. In Xamera, it is employed to detect LED lights that emit OOK signals unique to each user, enhancing privacy by reliably identifying the specific light regardless of ambient brightness. This detection then allows users to draw letters and digits.

**Machine Learning**

* **PyTorch:**Our letter and digit recognition models have been developed and trained using PyTorch, a Python library specifically designed for machine learning tasks. Both models were trained on the University of Michigan's Great Lakes Remote Computer Clusters and on local high-performance laptops. After training, the models were exported as “.pth” files, which were then converted to TensorFlow Lite format for improved performance.
* **ONNX:**To facilitate mobile deployment, we converted the PyTorch model to ONNX as an intermediate format. This step, however, introduced some degradation in performance.
* **TensorFlow Lite:**Finally, the ONNX model was converted to TensorFlow Lite to optimize it for inference on Android devices. Converting both the PyTorch-trained recognition models and the YOLO-based light detection model to TensorFlow Lite was necessary to avoid relying on PyTorch Mobile, which would have extended integration time and degraded performance.



## Performance Analysis & Results

#### **Path Smoothing**

While the initial YOLO model achieved 99% accuracy in detecting LED light sources, it does not exclusively detect the optimized LED beam. It often picks up unwanted ambient light, making it difficult to draw smooth lines with the LED gloves. As a result, users are required to perform airwriting in a dark environment for optimal performance. Future improvements will focus on refining the YOLO detection to exclusively recognize the LED beam designated for the user—which generates OOK signals with Arduino—thereby enhancing the smoothness of the drawn paths.

#### **Machine Learning**

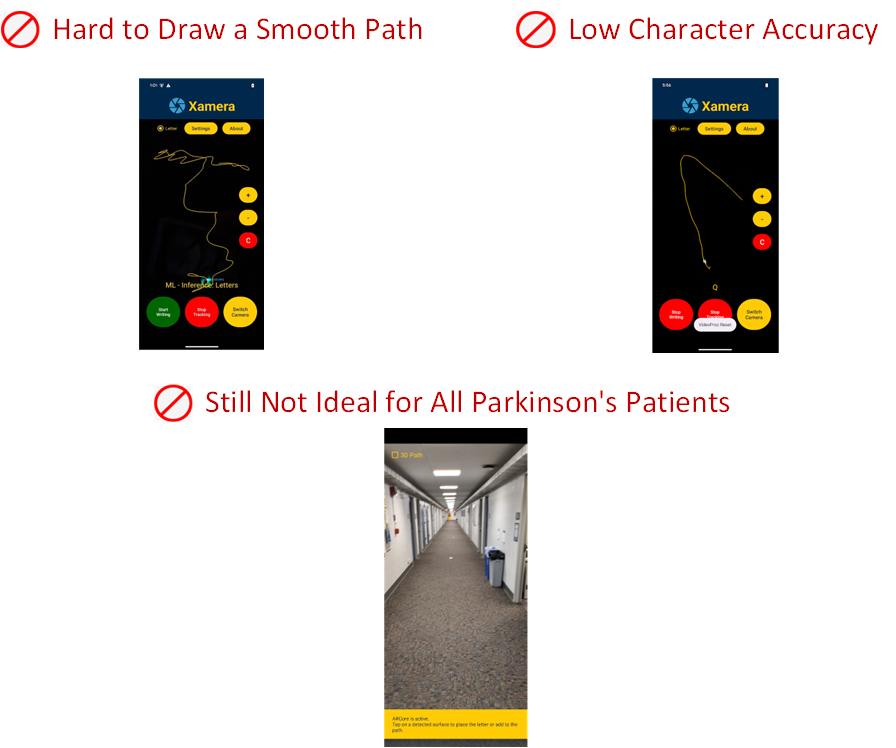
Our initial digit recognition model achieved 93% accuracy, while the letter recognition model reached 77% accuracy. Additionally, the initial YOLO model for detecting the LED light source demonstrated 99% accuracy. We plan to further design and train the YOLO LED light detection, digit recognition, and letter recognition models to boost their accuracy to a target of at least 90%.

#### **Augmented Reality**

The current AR implementation requires the user to tap on a surface to generate a 3D path or a 3D cube with a 2D letter. This manual step is a limitation, as it relies on user intervention instead of leveraging Google ARCore's potential for automatic surface detection. In future improvements, we aim to automate this process so that ARCore will automatically generate the object as soon as it detects a surface, thereby streamlining the user experience and enhancing overall usability.

#### **Accessibility**

In scenarios where two individuals with Parkinson’s are involved—one performing airwriting and the other holding the smartphone and AR device—accessibility becomes a critical concern. On some devices, the buttons may be too small, posing difficulties for users with motor impairments. To address these issues, we plan to optimize the button sizes and layouts specifically for Parkinson’s patients, ensuring a seamless and user-friendly experience for both the air writer and the device holder.



## Challenge, Solutions & Lessons Learned

**Lack of Experience with Non-Built-in Android Technologies**

* **Challenge:** I had no prior experience with technologies such as PyTorch, YOLO, TensorFlow Lite, OpenCV, or Google ARCore beyond built-in Android tools.
* **Solution:** I relied on code examples from Stack Overflow and other online resources, reverse-engineering solutions and adopting a divide-and-conquer approach to gradually learn and integrate these technologies.
* **Lesson Learned:** Self-directed learning and leveraging community resources are invaluable for overcoming unfamiliar technology challenges.

**No Background in Artificial Intelligence**

* **Challenge:** My AI experience was limited to creating custom ChatGPT models through prompt engineering, with little exposure to training deep learning models, CNNs, LSTM architectures, or concepts like epochs and inference.
* **Solution:** Collaborating closely with Soham and Zaynab allowed me to learn the fundamentals of deep learning and develop my own models.
* **Lesson Learned:** Hands-on collaboration and mentorship are critical in acquiring new technical skills in emerging fields like AI.

**CameraX's Rolling Shutter & Real-Time Processing Limitations**

* **Challenge:** CameraX’s higher-level abstraction lacked the low-level control required for adjusting rolling shutter frequency and real-time processing.
* **Solution:** The team switched to the Camera2 API, a transition that took 8 hours, to gain the necessary control over hardware.
* **Lesson Learned:** Sometimes, using a more low-level solution is essential to meet performance requirements, even if it requires additional development effort.

**Device Compatibility Issues with Google ARCore Support**

* **Challenge:** Motorola Moto G Pure devices did not support Google ARCore, limiting the ability to run advanced AR features.
* **Solution:** These devices were excluded from the project to ensure that only hardware with robust ARCore capabilities was used.
* **Lesson Learned:** Aligning hardware selection with software requirements is crucial to maintaining a consistent and high-performance user experience.

**OpenCV and Kotlin Integration**

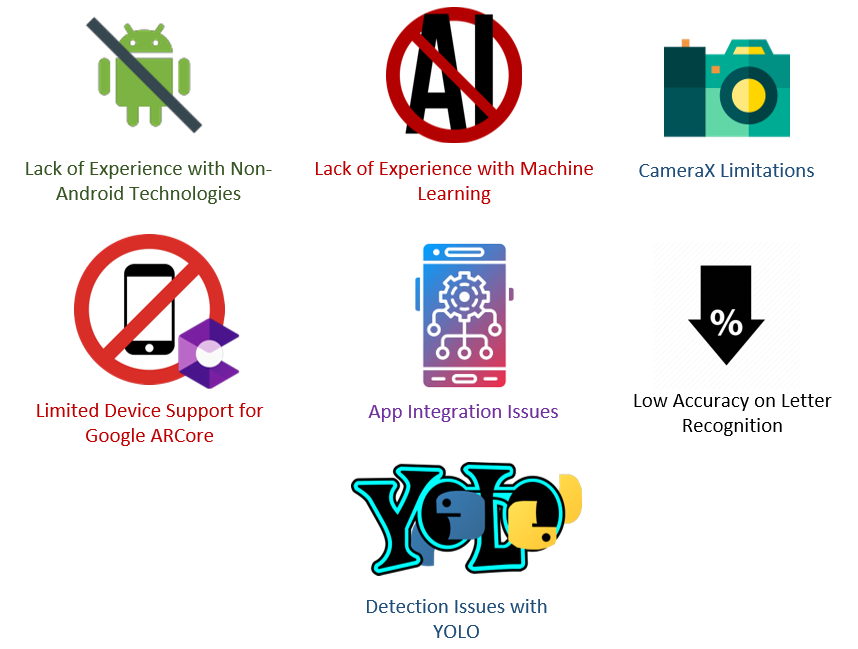
* **Challenge:** Converting standalone Python OpenCV code to Kotlin for seamless app integration was difficult, requiring a complete reimplementation of several core functionalities. Integrating the project with Gradle added further complexity in managing dependencies and build configurations.
* **Solution:** The process involved extensive reworking of algorithms, debugging compatibility issues, and optimizing performance in the Android environment.
* **Lesson Learned:** Transitioning code between programming languages and environments is complex and demands a structured, step-by-step approach and thorough testing.

**Low Letter Recognition Model Accuracy**

* **Challenge:** The letter recognition model, trained on the EMNIST dataset, achieved only around 77% accuracy—significantly lower than the digit recognition model's 93%—due to the increased number of characters (A–Z) and insufficient data, with further degradation from the model conversion process (PyTorch → ONNX → TensorFlow Lite).
* **Solution:** Although additional retraining and optimization were necessary, this highlighted the need for more extensive data collection for letters.
* **Lesson Learned:** Balanced and ample data is critical for robust model training, and model conversion processes must be carefully managed to preserve accuracy.

**Key Challenge: Accurate YOLO Detection Under Rolling Shutter Conditions**

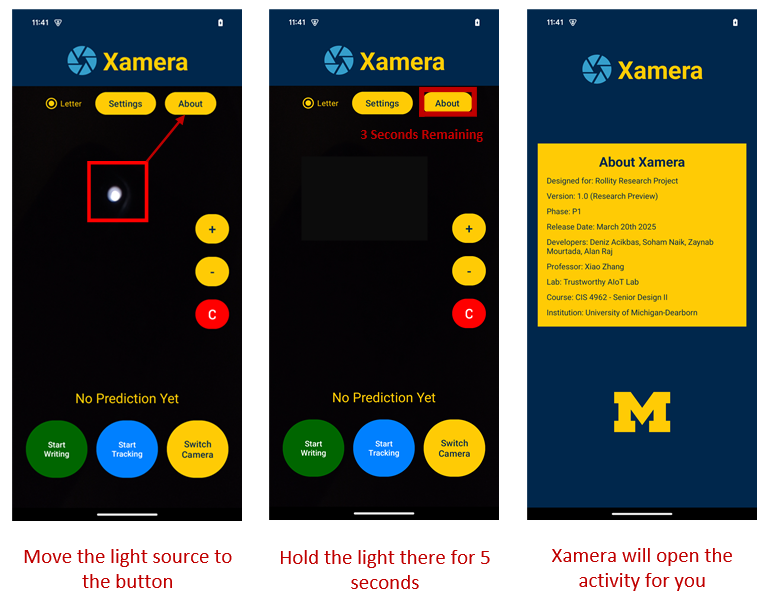
YOLO’s greatest challenge in our system was reliably detecting small, PWM-modulated LEDs captured under rolling shutter distortion. Since YOLO is optimized for spatial object detection, adapting it to identify temporal light patterns (like OOK signals) was non-trivial. The camera's rolling shutter caused flicker artifacts that varied with ambient lighting and motion, leading to occasional misidentifications and unstable gesture paths. Additionally, detecting small, fast-blinking LEDs in real-time pushed the limits of lightweight models like YOLOv11 Nano. Overcoming this required custom training on realistic LED footage and iterative fine-tuning to balance precision, speed, and robustness.



## Future Improvements & Reflections

**Hands-Off Experience**

We plan to implement a system that leverages a machine learning algorithm to detect an LED glove beam directed at a specific button area using precise button coordinates. When the beam is held steadily on the target for five continuous seconds, the algorithm will interpret this sustained focus as an intentional gesture and trigger the corresponding action. This hands-off approach minimizes physical contact—enhancing hygiene and reducing wear—while delivering a futuristic, reliable user interaction experience.



**Trajectory Generation and Optimization**

The video processing pipeline in our mobile app, Xamera, is managed by a dedicated class that handles the complete workflow—from receiving the camera input to real-time trace visualization. This pipeline consists of several interconnected stages:

**Camera Input and Pre-Processing:**

Frame-by-frame data is received via the Camera2 API.

A copy of each frame is passed to the pre-processing module, where thresholding and noise reduction isolate the objects of interest—the bright LED signals.

The resulting image is resized to a constant 416×416 resolution to ensure compatibility with the YOLO model for inference.

**Inference and Detection Processing:**

The pre-processed frame is converted into a tensor and fed into a custom-trained TFLite YOLOv11Nano model for object detection, utilizing the android device’s full hardware capabilities.

Post-inference, non-maximum suppression (NMS) is applied to retain only the most confident detection per class.

The final detection results—each comprising coordinates (x, y, width, height) and class labels—are scaled back to the original frame dimensions.

These detection points are stored in a class-specific FIFO queue that acts as a moving window, limiting memory usage by discarding the oldest point once the maximum queue size is reached.

**Trace Generation and Smoothing:**

The center of mass (COM) of each new detection bounding box is computed to obtain the (x, y) coordinates.

These coordinates are then passed through a moving average filter to smooth minor fluctuations, followed by a Kalman filter to further reduce noise and mitigate sudden jumps in the path.

A spline interpolation function is applied to the filtered points, generating an additional 50 interpolated points between consecutive detections. This step increases trace fidelity and produces a visually smooth, best-fit curve.

The spline-interpolated coordinates are stored in a separate class-specific FIFO queue, ensuring that only the most recent trace data is maintained.

**Frame Updates for Real-Time Visualization:**

Concurrently with the previous steps, the original input frames are sent to a dedicated frame update module.

This module overlays the most recent bounding boxes and a configurable number of points from the spline-generated trace onto the live video display, providing intuitive real-time feedback to the user.

**Optimization and Results**

This integrated pipeline combines noise reduction, precise detection, and high-fidelity trace generation into a real-time, user-focused proof of concept. Design choices were driven by both project requirements and user-centric considerations to ensure smooth, real‑time performance without sacrificing privacy or visual quality.

**Model Conversion (PyTorch to TFLite):** Converting the PyTorch model to TensorFlow Lite reduced inference latency from 140 ms (≈7 FPS) to 80 ms (≈12.5 FPS)—a 1.79× improvement. TFLite is optimized for Android and edge devices, making it more efficient on the CPU.

**Hardware Acceleration with NNAPI:** Although TFLite defaults to CPU inference, the Neural Networks API (NNAPI) enables deployment on GPUs or dedicated NPUs/TPUs. With NNAPI, latency further dropped from 80 ms to 30 ms (≈12.5 FPS to 33 FPS), a 2.64× improvement.

**Quantization:** Quantization reduces model size by converting weights and activations to lower-precision formats. Typically:

**CPUs:** Use FP32 (32-bit floating point), offering high precision.

**GPUs:** Are optimized for FP16 (16-bit floating point), which halves memory usage and improves throughput.

**TPUs/NPUs:** Often use INT8 (8-bit integer), providing even faster computation and lower power consumption.

In our benchmarks, quantizing from FP32 to FP16 and then to INT8 reduced latency further—from 30 ms to 22 ms and finally to 17 ms (approximately 33 FPS to 45 FPS to 59 FPS). Although these gains are hardware specific, quantization can substantially lower latency with only minimal accuracy loss. (Note: Quantization was benchmarked but not implemented in the final mobile app.)

**Concurrent Processing:**Running the processing (inference) and frame update modules on separate threads minimizes bottlenecks. Frame updates occur in under 10 ms, while processing averages 30 ms. This concurrency yields a real-time output of about 33 FPS, though the high refresh rate of frame updates can make the experience feel as smooth as 100 FPS. Without this threading, inference delays could cause noticeable lag.

By leveraging TFLite's optimizations and hardware acceleration through NNAPI, combined with quantization and multithreaded processing, the system achieves significant latency improvements while maintaining high accuracy and a smooth user experience.

**End-to-End Model Development and Integration**

This section outlines our complete workflow—from high-performance model training to final mobile app integration—ensuring both the YOLO-based object detector and the handwriting recognition models meet our performance, efficiency, and accuracy goals. The process encompasses model training in an HPC environment, iterative benchmarking and improvements, model conversion (with optional quantization), and seamless integration into the mobile pipeline.

**High-Performance Training Environment**

Models were initially developed and trained using PyTorch with CUDA on high-performance computing clusters. This environment allowed us to experiment with various architectures and process large datasets efficiently, forming the basis for both the YOLO detection model and the digit/letter recognition models.

**Iterative Benchmarking and Optimization**

We benchmarked models in our local Python development environment to evaluate inference latency, accuracy, and overall performance. By iteratively refining the models—such as expanding datasets, cleaning data, and extending training epochs—we achieved significant improvements in efficiency and precision.

**Model Conversion and Quantization**

Following optimization, models were converted from PyTorch to ONNX and then to TensorFlow Lite for deployment. We also explored quantization techniques (e.g., converting FP32 models to FP16 or INT8) to reduce model size and boost inference speed, while carefully monitoring any potential accuracy trade-offs.

**Mobile App Pipeline Integration**

The final TFLite models were integrated into the Xamera mobile app. Dedicated modules in the app pipeline handle real-time inference, ensuring that model predictions are seamlessly incorporated into the user interface without compromising responsiveness or visual quality.

Deniz Acikbas

****

**Deniz Acikbas**

*Lead Software Engineer and Software Integration Specialist*

+1 734 660 8867

[dacikbas@umich.edu](mailto:dacikbas@umich.edu)

<https://www.developer27.com/>

Hi, I'm Deniz. I led the progress on the Rollity (Xamera) Project, managing both front-end and back-end tasks. On the front end, I crafted the Maize and Blue theme, while on the back end, I developed an inference algorithm for machine learning models that recognize digits and letters. I integrated key technologies like Google ARCore, Camera 2 API, and OpenGL to enhance the project’s functionality. Additionally, I disciplined the team by assigning tasks and maintaining an organized workflow, and I directly reported progress on Xamera to our client, Dr. Xiao Zhang.

**Prior Major Projects**

* **Falcon |** Task scheduling app | Released in August 2024
* **Project Detroit |** A small budgeting app | Released in August 2023
* **Avalon |** Referral app for professors to refer students | Released in August 2022
* **Children`s Market |** A cashier app for children | Released in August 2021
* **X Word |** Java-based word processor for writing code | Released in December 2020

**Why Adaptive HCI?**

I have a strong interest in computer graphics and human-computer interaction, and I wanted to go deeper into AI to understand how it works. Dr. Zhang's Adaptive HCI project appeals to me because it not only explores these technologies but also incorporates machine learning, which is essential for the future of interactive systems. In addition, his proposed handwriting recognition model, which leverages image recognition techniques, aligns perfectly with my goal of delving further into image recognition within the field of machine learning.

**Why is the codename of Rollity is Xamera?**

Assigning codenames to month-long projects is a longstanding tradition in my software development approach. For instance, Microsoft used to assign names to their projects until the final release—Chicago for Windows 95, Whistler for Windows XP, and Longhorn for Windows Vista are just a few examples. Xamera leverages a smartphone camera with the Rolling Shutter effect to protect user privacy while also harnessing machine learning. This means it's not merely a camera app; it's a Xamera app. Here, the "X" symbolizes the future, and "amera" stands for the camera. In essence, Xamera represents a future camera that prioritizes privacy and incorporates artificial intelligence.

**Dr. Xiao Zhang was the best client**

Dr. Xiao Zhang was an exceptional client. Working with him on any project felt like collaborating with a close friend from high school. He offered us a publication opportunity with Rollity, organized workshops on machine learning, and provided the chance to join his Trustworthy-AI Lab and other research projects. He consistently offered positive feedback and would purchase our equipment the moment we asked. Notably, during the first two months, he even covered the project fees from his own income to facilitate our work. Although I did not take any of his classes, I consider him the best professor since my primary school teacher.

**Technologies Integrated**

1. **PyTorch**

Our letter and digit recognition models have been developed and trained using PyTorch, a Python library specifically designed for machine learning tasks. Both models were trained on the University of Michigan's Great Lakes Remote Computer Clusters and on local high-performance laptops. After training was completed, the models were exported as “.pth” files, which were then converted to TFLite format for better performance.

1. **YOLO**

YOLO is a pre-trained machine learning model designed for fast image detection, and its name stands for "You Only Look Once." In Xamera, YOLO is used to detect LED lights that emit OOK signals unique to each user. This approach enhances privacy by ensuring that the specific light is detected regardless of the room's brightness. Once the light is detected, the user can draw letters and digits.

1. **TensorFlow Lite**

The letter and digit recognition models were written in PyTorch, and the light detection model was implemented using YOLO. Both had to be converted to TFLite models for smooth integration. Otherwise, we would have had to rely on PyTorch's legacy feature, PyTorch Mobile, which would have significantly increased the integration time and caused performance issues.

1. **OpenCV**

OpenCV is a widely used computer vision library. In Xamera, once YOLO detects the LED light source, OpenCV creates lines around it using a Kalman filter, spline interpolation techniques, and the Apache Commons Math library to generate a smooth path. This path is then processed either by Google ARCore, which generates a 3D path or 3D Letter Box, or by TensorFlow for letter or digit inference.

1. **Google ARCore**

Google ARCore is a technology developed by Google to provide an augmented reality experience for Android users. The final step of the process involves creating 3D paths or 3D letter boxes to simulate the inference of the digit or letter drawn by the user, or to display the last drawn path if the user chooses not to send an email or make a phone call with the processed letter. This step was included to validate our research by confirming that the path is smoothed correctly and that the inference is accurate.

1. **Android Studio (Kotlin + Java + XML)**

Xamera is exclusively developed for Android, so it leverages the Android libraries and compilers available in Android Studio. Notably, these libraries enable functionalities such as initiating a phone call using the predicted phone number and composing an email based on the predicted letter. Moreover, Android Studio provides a variety of built-in features—from launching intents to creating splash screens—that enhance the overall application experience.

**Challenges**

1. **Lack of Experience with Non-Built-in Android Technologies**

Aside from Python, Android Studio, and other built-in SDK libraries, I did not have prior experience with PyTorch, YOLO, TensorFlow Lite, OpenCV, or Google ARCore. Most of my Android knowledge was acquired through CIS 436. Using code examples from Stack Overflow and other websites, I reverse-engineered solutions and adopted a divide-and-conquer approach to gradually learn the technology and ultimately complete the task.

1. **No Background in Artificial Intelligence**

Prior to Xamera, my experience with AI was mostly limited to creating custom ChatGPT models provided by OpenAI through prompt engineering. I had no knowledge of training deep learning models, convolutional neural networks, or LSTM architectures. Additionally, I was unfamiliar with key concepts such as epochs and inference. However, by working closely with Soham and Zaynab, I learned the fundamentals of deep learning and am now capable of creating my own deep learning models.

**Future Plans**

1. **Rollity Academic Journal Paper**

After releasing the Xamera Research Preview on March 20th, 2025, our next steps involve improving the accuracy of both the YOLO Object Detection Model and the Letter and Digit Recognition Models by redesigning their architectures and retraining them. Once the accuracy of these machine learning models has been enhanced, we will carry out the specific tasks assigned by Dr. Zhang for inclusion in his journal paper. After completing these tasks, we will begin drafting the journal paper on Overleaf and creating the accompanying figures. Our paper will be published at one of the Association for Computing Machinery's conferences, marking our first academic research publication.

1. **RoUtism Research Project**

Machine learning and computer vision features from Xamera will be integrated into RoUtism, introducing a novel approach to emotion recognition. RoUtism will be my first authored paper, co-sponsored by Dr. Xiao Zhang, and it is expected to be completed in the summer of 2025.

“  
Machine learning model development and training began by setting up the Great Lakes Computers with PyTorch CUDA. As a practice exercise, letter and digit recognition models were developed using convolutional neural networks and LSTM models in a PyTorch environment and trained on coordinate numbers. These models were then integrated with Xamera. Unfortunately, the initial models had a very low accuracy rate and their designs were not consistent with the research paper, so the machine learning development process was restarted.

The first machine learning model trained was the YOLO (You Look At Once) model, which detects the LED beam node of the glove optimized for 5 kHz using Arduino. A total of 2,640 images and 2,640 labels were used across three classes, with 80 percent allocated for training and 20 percent for validation. Once the accuracy rate had increased to 99 percent prior to the alpha demo, the model initially performed inference on a sample image. After the bounding box functionality was implemented, inference could be done on a real-time image. At that point, Soham replaced the Kalman filters and contour detection with the YOLO model and created a bounding box. Since PyTorch is not directly supported on Android, we converted the model to ONNX and then to TensorFlow Lite, although the app was initially using YOLO inference. The features that handle the YOLO model are located in the VideoProcessing.kt file.

The first integrated handwriting recognition model was the digit recognition model. The digit recognizer was first trained on MNIST and DIDA, then fine-tuned on our 600+ image Xamera dataset (with added noise), achieving over 99 percent training accuracy and near-perfect test results. The digit recognition model was first converted to ONNX to expedite the process of converting to TensorFlow Lite, and then converted to TensorFlow Lite. The problem with these conversions was that we had to create a separate Python environment to install the necessary libraries because local drive paths were not supported, which even required changing the Windows registry. After the converted TensorFlow Lite model was placed in the Assets folder, an inference algorithm was written that took the path from VideoProcessor.kt, converted the image into a 28×28 tensor for input generation, and assigned the inference prediction to a string in MainActivity.kt. After successfully integrating the digit recognition model, we released the first beta version of Xamera 1.0.

Later on, I trained the letter recognition model on my local PC using 520 images along with the EMNIST dataset. The overall accuracy was relatively lower, at 77.8 percent, compared to the digit recognition model. I repeated the same process as with the digit recognition model—converting to TensorFlow Lite and performing a 28×28 input conversion. Currently, Soham and Zaynab are working on increasing the accuracy rates of the YOLO object detection, digit, and letter recognition models to improve efficiency, while I handle the integration.

“

“  
At the beginning of the project, we created a simple camera app named Xamera using the CameraX API—a higher-level API—that allowed users to switch between the front and back cameras and capture video. In parallel with these changes, Soham Naik developed a standalone light detection and line drawing program using OpenCV in Python. After completing the Settings and About screens, I transformed Soham’s standalone light processing code into Kotlin. At that point, users could record a video, save it, and open it in Xamera, which then processed the video using embedded OpenCV-based video processing code.

After the transition from CameraX to the Camera2 API was completed, Xamera was able to process light in real time and create lines, which were further refined through the addition of spline interpolation, Kalman filters, and line trace techniques to draw smoother lines that help Parkinson's patients—all implemented using OpenCV. Video processing could be performed between when the user clicks "Start Tracking" and when they click "Stop Tracking". Throughout the development process, once the YOLO model was integrated, contour detection was replaced with machine learning model-based YOLO technology, which not only detects the light source under rolling shutter conditions but also displays an accuracy rate within a bounding box.

Several arguments have been made regarding whether to use plain OpenGL or Unity VR, but after developing several standalone programs with both and conducting external research, we decided to work on Google ARCore. Once we decided to use Google ARCore, we replaced our Motorola Moto G phones with Google Pixel 8 devices, since the Moto G lacked support for Google ARCore and its performance in machine learning model inference was relatively low.

A separate app called Xamera AR was developed, in which a 3D path and a 3D box with a 2D letter were generated using OpenGL ES. Xamera AR was written in Java with Google ARCore technology and reverse-engineered from a Google ARCore sample available on GitHub. This functionality was later transformed into a library and integrated with Xamera as an intent.

After Zaynab Mourtada completed the digit and letter recognition models and implemented their inference algorithms, the parameters generated by these ML inference algorithms were passed to the Xamera AR intent to generate a 3D path and a 3D letter box with a 2D letter, simulating the word or number drawn by the user. Later on, I added functionality that dials a phone number if it’s a digit and sends an email if it’s a word. Xamera also includes a rotation lock that prevents the app from rotating to landscape mode and a hibernation lock.

The theme of the app is exclusively Maize and Blue, representing the colors of the University of Michigan, as we are part of the university. The light blue icon with lines represents a camera app, conveying that this is more than just a camera—it’s the camera of the future. There are also zoom in, zoom out, and clear features in addition to front and back switching.

Xamera works by first allowing the user to set the rolling shutter frequency range (depending on their phone) and then select the type of airwriting—whether they will draw a letter or digit—using a radio button. After that, the user clicks "Start Writing" to begin the process, then clicks "Start Tracking" to draw a special character (which could be a letter or digit), and finally clicks "Stop Tracking" to receive a guess of what they have drawn. This process is repeated until the user writes a complete number or word. Once finished, the user clicks "Stop Writing." Xamera then asks whether to compose an email (or dial a phone number if a number was written). If neither option is chosen, Xamera launches its Augmented Reality environment, which provides an option to display the last letter or digit in 3D or the entire number or word as a letter box when the user taps on a surface.

“

Zaynab Mourtada

***Zaynab Mourtada***

**Supporting Research:**

I read the paper “Real-Time User-Independent Hand Gesture Recognition Using Deep Learning” because it’s very similar to our project. It helped me understand how gesture recognition works using deep learning, which I used as a reference when building our own digit and letter recognition system. I trained both models using CNNs, so when someone writes a letter or number in the air using the LED gloves, the app can detect it and convert it into text in real time. This directly supports the inference part of our professor’s research project, which focuses on recognizing and processing hand-written gestures through a camera.

**Technologies Used:**

For the digital handwriter recognition, I used Convolutional Neural Networks (CNNs) and trained both the digit and letter models using PyTorch. I trained the models on the Great Lakes Cluster because I needed GPU support, and then tested them on my personal computer. The digit recognizer was trained on the MNIST and DIDA datasets, and then fine-tuned on our own Xamera dataset, which has a little over 600 images. The letter recognizer was trained on the MNIST and EMNIST datasets, and also fine-tuned on the Xamera dataset, which consisted of 520 images in total.After training, the models were converted to TFLite for integration into the mobile app by Soham and Deniz.

**Results:**

The digit recognizer achieved over 99% accuracy during training on the Xamera dataset. It reached 98–99% accuracy on MNIST and DIDA during testing, and 100% accuracy on the Xamera test set—likely due to the smaller dataset size. After converting the model to TFLite, the accuracy dropped slightly.

The letter recognizer reached about 98% accuracy on EMNIST and 91% on Xamera during training. Testing accuracy was around 83% on EMNIST and 77% on Xamera. Accuracy also dropped a bit after conversion to TFLite.

**Lessons Learned:**

One of the main challenges with the digit recognizer was that it hit 100% accuracy during testing on my laptop, but that was only because the Xamera dataset was small—just over 600 images—and the model was being tested on a limited subset. A larger dataset, maybe around 1,000–2,000 images, would have helped give a more accurate picture of performance.

For the letter recognizer, I ran into issues with model complexity. At first, I tried using the same number of layers as the digit recognizer, but that didn’t work well because letters have more strokes and variation in handwriting. I ended up needing more layers and had to increase the learning rate slightly to better handle that complexity

Another thing I learned was that converting the models to TFLite and integrating them into the app naturally causes a drop in accuracy. I didn’t realize that at first, but I became aware of it after seeing the performance drop post-integration.

Alan Raj

**Alan Diviya Raj**

**Supporting Research:**

I initially joined this project because of my data science background and my interest in the machine learning aspect of gesture recognition. However, as the project evolved, I found myself delving into 3D modeling and visualization, which was a completely new area for me. To gain a deeper understanding, I studied research on motion tracking and filtering techniques, such as *“*3D hand tracking using Kalman filter in depth space*”*, which helped me grasp how filtering techniques can be applied to smooth noisy motion data. This knowledge helped guide my implementation of Kalman filtering and cubic spline interpolation for refining the air-written paths captured from the LED glove. By stabilizing the trajectory data, my work displays in real-time a more accurate representation of the user’s intended writing.

**Technologies Used:**

For the real-time 3D visualization of air-written letters, I utilized OpenGL to render the tracked path in a virtual space. The raw coordinate data, captured as a 4D sequence, was processed using Kalman filtering to reduce noise and hand tremors, ensuring a smoother representation of the writing motion. Additionally, I applied cubic spline interpolation to further refine the paths by smoothing out abrupt or “choppy” transitions between points. This processed path was then displayed in real-time, allowing users to see their air-written letters in a clear, continuous motion.

Although we ultimately did not use Unity in the final implementation, I spent a significant amount of time learning Unity and C# from scratch to explore other visualization options. This experience helped me better understand 3D rendering concepts and how different game engines handle real-time graphics, which influenced how I approached the OpenGL-based implementation.

A major challenge I faced was working with Kotlin, a language I had no prior experience with. Coming from a Python background, I had to adapt to Kotlin’s syntax and structure, which was quite different from what I was used to. Thankfully, with the help of my teammates, I was able to translate my logic into Kotlin and integrate my work into the existing codebase. This experience improved my ability to work across different programming languages and reinforced the importance of team collaboration in software development.

**Results:**

The combination of Kalman filtering and cubic spline interpolation significantly improved the accuracy and smoothness of the tracked paths. Initial tests without filtering showed noticeable noise and instability in the displayed trajectory, which would have made letter recognition more difficult. After applying Kalman filtering, the path stability improved, reducing fluctuations caused by hand tremors. The cubic spline interpolation further enhanced the smoothness by ensuring a natural curve between sampled points. These improvements not only provided a better user experience but also enhanced the performance of the letter recognition model by delivering cleaner input data.

**Lessons That I Learned:**

Learning 3D modeling and visualization was a major shift from my data science background. While I initially expected to focus more on machine learning, I ended up developing skills in graphics programming and real-time rendering. The time spent learning Unity was still valuable, as it helped me understand concepts like coordinate transformations, lighting, and rendering pipelines, which were useful when working with OpenGL.

The experience of working in Kotlin reinforced the importance of adaptability in software development. Initially, I found the transition from Python challenging, especially when dealing with Kotlin’s type system and Android-specific syntax. However, with guidance from my teammates, I was able to successfully implement my visualization components within the existing Kotlin-based application. This project broadened my technical skill set and gave me a deeper appreciation for the intersection of data science, visualization, and real-time AR applications.