Benchmark NSW

DIY Guidebook

Sep, 2024

UNSW

Massachusetts Institute of Technology

LCAU

NSW GOVERNMENT

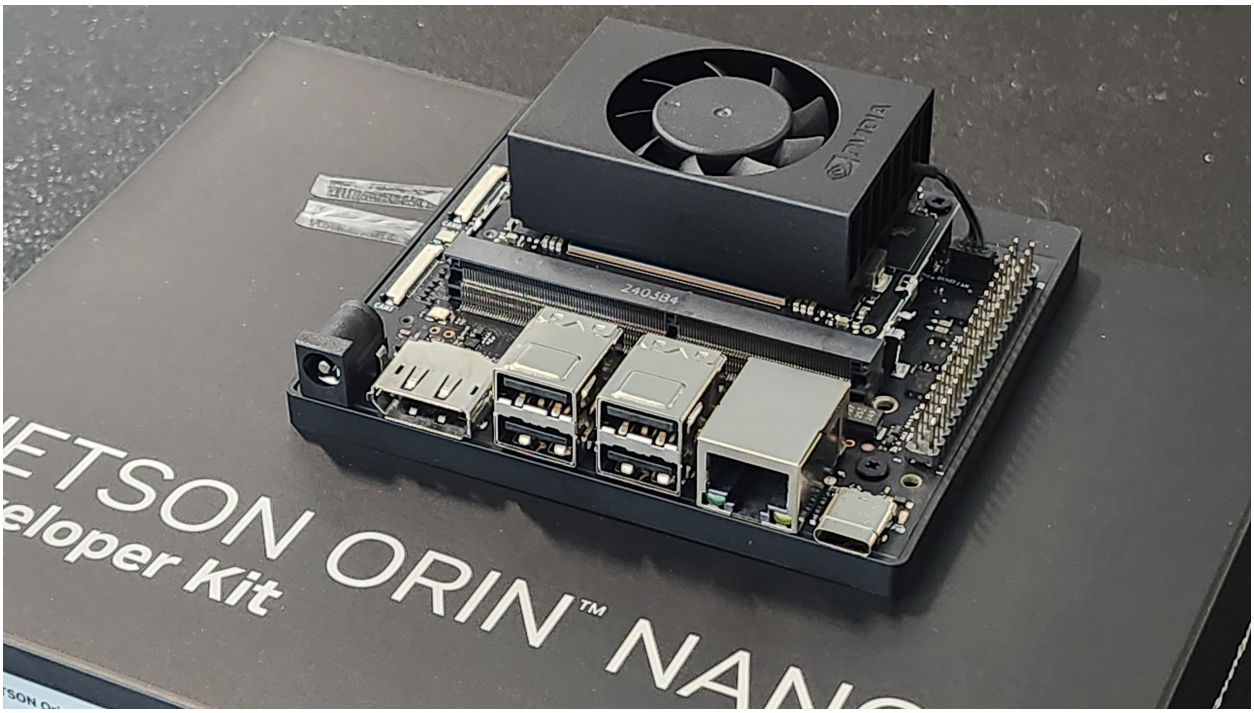
Contents

The Benchmark UNSW project, aimed at quantifying the activation of public spaces using Vision AI, employed the Nvidia Jetson. This AI computing device enabled real-time data processing, anonymization, and encryption of the data collected from various public spaces. By leveraging the Nvidia Jetson Orin Nano Developer Kit, we ensured robust performance and efficient handling of video streams, which was critical for accurate and secure analysis of public space utilisation.

Inspired by William Whyte's seminal studies on public space observation, which highlighted the importance of understanding how people interact with urban environments, our project takes a modern approach to this classic method of urban analysis. Whyte's work emphasised meticulous observation and recording of pedestrian behaviours, seating choices, and social interactions in public spaces. By applying these principles, we aimed to gather detailed insights into how public spaces are used, but with the enhanced capabilities of modern AI technology.

This guidebook explains how to set up the Nvidia Jetson environment, stream video from a GoPro, implement a basic object detection model, train a custom model, sample data, and outline the overall data process. We also discuss the challenges we faced and the solutions we found, such as using the Nvidia Jetson to anonymize and encrypt data in real-time. This comprehensive approach ensures that all aspects of data collection and analysis are thoroughly covered, providing a valuable resource for similar projects using Vision AI to study urban spaces.

1. NVIDIA Jetson



NVIDIA Jetson is a single board computer with state-of-the-industry processing power. That said, it is on the expensive side and is built for developers, so it can offer a higher monetary and technical barrier to entry than some alternatives. What makes Jetson stand out among similar single board computers like Raspberry Pi is that it's purpose-built with a GPU for powering AI models, such as the ones we employ in our computer vision code to assess pedestrian and bench activity on the site, meaning that we can perform inference detections in real time. In essence, we chose to use Jetson for this project because it is portable enough to be used as an edge device that is mounted outside nearby our camera sensor and is powerful enough to process and immediately discard raw video footage to ensure that our data is 'de-identified at source' (that is, that the streaming is never transferred anywhere after collection, but rather, used as a mere lens for AI detections of activity).

The following directions are meant to lessen the technical barrier to entry for similar projects that might like to set up Jetson to better understand activity in public space.

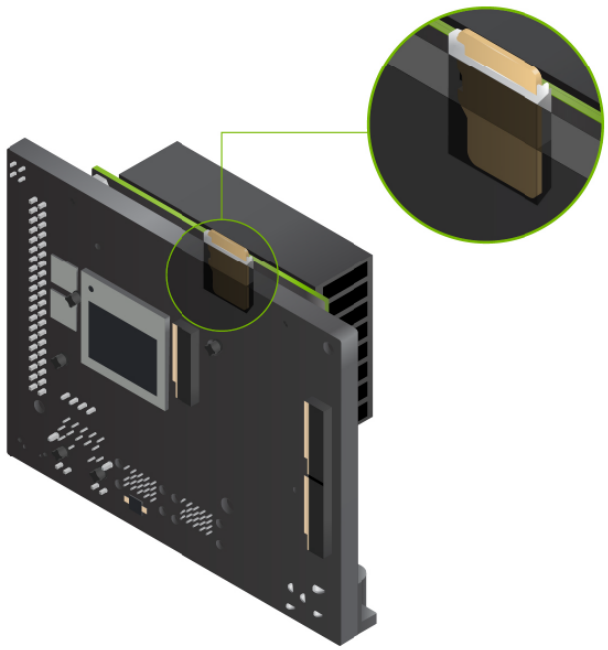
1.1 Hardware setup

Required Items

- NVIDIA Jetson Orin Nano device
- Solid State Drive (this one, for example).
- Micro SD Card
- A computer with a micro SD card slot or an external SD card reader with write capability (this one, for example).
- A computer monitor
- A DisplayPort to HDMI (or DisplayPort to USB-C, if your monitor supports it) cable
- USB keyboard
- USB mouse
- Small screwdriver

Set-up guide

1. Physically install a solid state drive (SSD) into the underside of the Jetson device, using a small screwdriver. [I'll include an annotated picture of it installed on our device as well to better explain how it works.] Running Jetson's operating system, JetPack, from a SSD substantially improves the device's performance.
2. Follow the NVIDIA Jetson AI Lab's instructions to flash the correct version of JetPack onto the SD card and update the firmware if needed.
    - Note: Do not use the "I'm feeling lucky" directions. This will likely only waste time, as most Jetson devices purchased do in fact need firmware updates.
3. Once you have the firmware updated and the JetPack system image ready to go, do the following to boot Jetson (adapted from the directions offered by NVIDIA):
    - Insert the microSD card (with system image already written to it) into the slot on the underside of the Jetson Orin Nano module.



- Power on your computer monitor and connect it through Jetson's DisplayPort using the DisplayPort to HDMI cable.
- Connect the USB keyboard and mouse.
- Connect the provided power supply. The Jetson Orin Nano Developer Kit will power on and boot automatically.
- A green LED next to the USB-C connector will light as soon as the developer kit powers on.
  - When you boot the first time, the Jetson Orin Nano Developer Kit will take you through some initial setup, including selecting system language, keyboard layout, and time zone, and creating a username, password, and computer name.
  - You can hold off on setting up wireless connections for now, as we'll set that up later.
- Next, you will be prompted to log in. If you see a screen that looks something like this, you're all set:

1.2 Connecting Jetson to the Internet

Required Items

- Mobile hotspot device
- Ethernet cable

Setup guide

1. Attach the hotspot to the Jetson with the Ethernet cord (it is perhaps easiest to purchase a hotspot that comes with an Ethernet cord), using Jetson's Ethernet port.
2. Boot up Jetson and open the Wifi connection settings (you can find this in the upper right side of the screen).
3. Select the name of your hotspot as the 'Network' from the list of options. Put in the password provided in the hotspot box to connect for the first time.
4. Ensure that you see the Connected message next to the Network name.

1.3 Ensuring remote access to Jetson

To set up remote access to the Jetson device while it is deployed in the field, we will follow a process heavily based on the Medium article How to access a Raspberry Pi anywhere with reverse ssh and Google Cloud Platform by Jason Jurotich. (Raspberry Pi devices have similar components as the NVIDIA Jetson).

1. First, set up a compute instance (a virtual machine) in the Google Cloud Platform. We found that we needed a slightly larger machine type, `g1-small`, than the one recommended in Jurotich's article, because we were accessing our machine quite often and its CPU was over-utilised when we had the `f1-micro` machine. Find our machine configuration below:
2. Create a Google Cloud Firewall Rule to allow traffic through ports 64001-64005 on the default network used by your new instance.

Storage

Boot disk

Name ↑	Image	Interface type	Size (GB)	Device name	Type	Architecture	Encryption
ssh-proxy-server	ubuntu-2204-jammy-v20240628	SCSI	30	ssh-proxy-server	Balanced persistent disk	x86/64	Google-managed

Figure 1: machine configuration 2

3. Navigate to the 'IP Addresses' menu in Google Cloud. Reserve a static external IP address for your new virtual machine from there. This will incur a small cost but ensures the long-term integrity of the reverse proxy server setup process, which is especially important if you don't have consistent (or any) physical access to Jetson after initial configuration.
4. Over in Jetson's terminal, create SSH keys by running the following (accept all defaults):

   `ssh-keygen -t rsa && cat ~/.ssh/id_rsa.pub`

   - Copy the resulting key.
5. Go back to Google Cloud and click on your new virtual machine to see its instance details. Click Edit and scroll down to the SSH Keys section. There, add a new entry and paste in the key you just copied from Jetson.
6. Now, in the Jetson terminal, SSH into the Google Cloud Platform virtual machine to accept the SSH keys from there.

   - Simply type: `ssh {email}@{STATIC_EXTERNAL_IP}`
   - Then, type yes after entering.
   - Finally, type exit to exit the virtual machine.
7. In the Jetson terminal, now type `vi tunnel.sh` to create a new file. Once in this file, paste the following, with requisite changes to your username and IP (type `:wq` to save and exit the file after you have pasted):

```
remote_user={your username}
remote_host="{your virtual machine's static IP}"

# Function to get the IP address of a network interface
get_ip() {
    local interface=$1
    ip addr show $interface | grep 'inet ' | awk '{print $2}' | cut -d/ -f1
}

# Get IP addresses for the ethernet connection, eth0
interface1_ip=$(get_ip eth0)

# Function to set up reverse SSH tunnel
setup_tunnel() {
    local remote_port=$1
```

```
    local selected_ip=$2
    local local_port=$3
    ssh -fN -R ${remote_port}:${selected_ip}:${local_port} ${remote_user}@${remote_host}
    echo "Reverse SSH tunnel established: Port ${remote_port} -> Port ${local_port}
    on ${selected_ip}"
}

# Set up first reverse SSH tunnel
setup_tunnel 64001 ${interface1_ip} 22

# Additional tunnels or commands can be added as needed
# setup_tunnel <remote_port> <interface ip> <local_port>
```

8. Finally, we'll use Ubuntu's `cron` scheduling function to ensure that the tunnels stay open and that we have consistent access from our virtual machine to our Jetson.
    - Type `crontab -e`
    - Paste the following at the end of the file: `/1* * * * ~/tunnel.sh > tunnel.log 2>&1`
    - Type `:wq` to save and exit.
9. Type `sudo reboot` to reboot Jetson so that the cron changes take effect.
10. Go to the Google Cloud Platform virtual machine instance and click on the SSH button to ssh into the machine (it will open a new browser window for 'browser SSH').
    - Once a terminal opens, type `ssh -p 64001 {username}@localhost`, where the `{username}` value represents the user that you want to log into Jetson as.
    - The system will prompt you for a password, and once you enter it, you're in!

2. Camera (GoPro)

The GoPro HERO12 Black is a popular waterproof camera designed for outdoor activities, and it offers the GoPro SDK, allowing users to control the camera through code. This section outlines the steps to integrate the GoPro into our project. However, it's important to note that while GoPro cameras are waterproof, they are not weatherproof and can overheat in direct sunlight at temperatures above 30 degrees Celsius. Our project was conducted under mild conditions, below 30 degrees Celsius, but if you plan to use the camera year-round, including in summer, it is recommended to design a sunshade for the GoPro or choose another camera module that can be directly connected to the NVIDIA Jetson, ensuring all devices are appropriately waterproofed. We used the GoPro HERO12 Black for real-time streaming, utilising the GoPro API for capturing and streaming data, and this section explains the initial connection tests with the GoPro, including example codes. A key improvement in this version of the benchmark project is that we do not save any video footage; instead, we use the GoPro solely as a lens for streaming data, enhancing privacy and efficiency compared to previous versions.

The following directions aim to reduce the technical barriers for similar projects that want to use GoPro cameras to capture and stream real-time data in outdoor settings.

2.1 Hardware setup

Required Items

- GoPro HERO12 Black
- Laptop with Bluetooth and Wi-Fi Network Card (If your laptop has only one Wi-Fi network card and you are using Wi-Fi to connect to the internet, you will lose internet access once the GoPro is connected. We recommend using an Ethernet connection to maintain internet connectivity.)
- 100W PD Charger (pull out battery and connect 100W PD charger to GoPro directly)
- SD Card

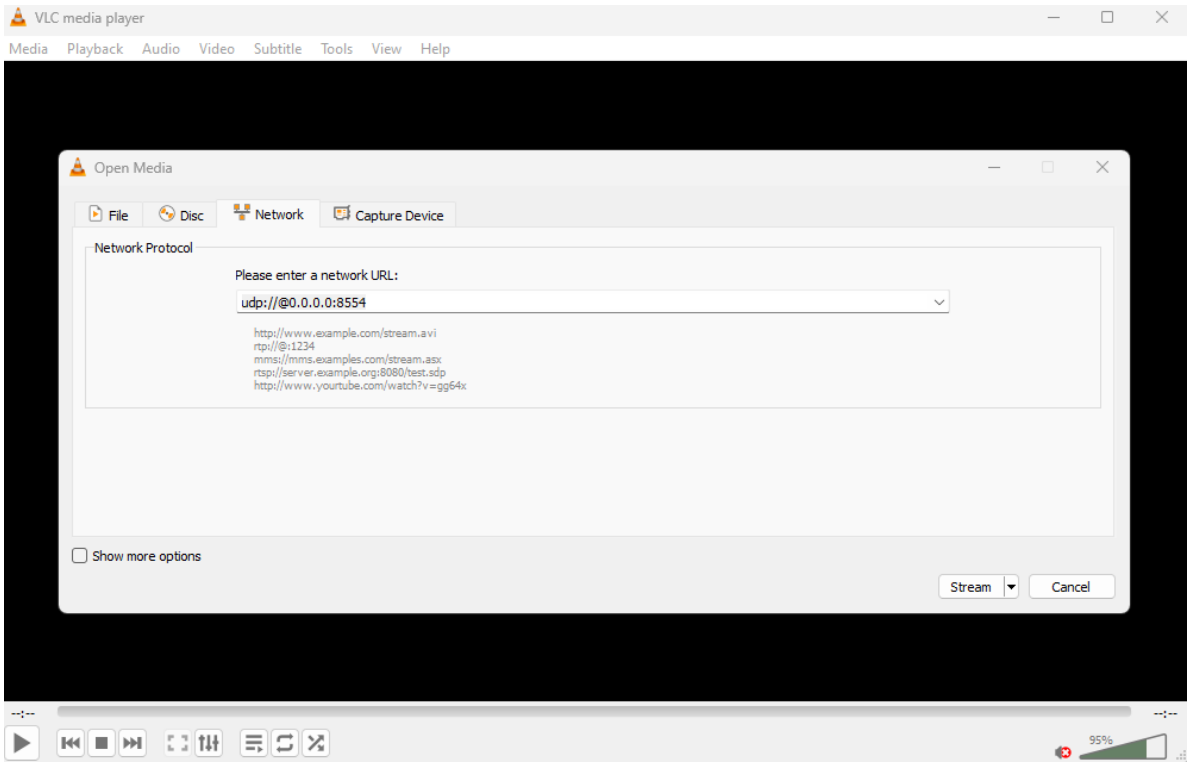
2.2 Connecting GoPro and Streaming Video

1. If you haven't done the firmware updates yet, please follow the instructions that come with the GoPro product package. You can also refer to this official GoPro instruction.
2. Place the GoPro near your laptop and run the code below. Please make sure you have set up the required Python environment.

```
import argparse
  import asyncio
  from rich.console import Console
  from open_goPro import WirelessGoPro
  from open_goPro.logger import setup_logging

  console = Console()

  class GoProStreamer:
      def __init__(self, args):
          self.args = args
          self.logger = setup_logging(__name__, args.log)
          self.goPro = None

      async def connect_goPro(self):
          """Connects to the GoPro camera."""
          console.print("Attempting to connect via Bluetooth...")
          self.goPro = WirelessGoPro()
          await self.goPro.__aenter__()
          if self.goPro:
              console.print("Successfully connected to GoPro.")
          else:
              console.print("Failed to connect to GoPro.")

      async def start_stream(self):
          """Starts the live stream."""
          try:
              # await self.stop_stream()
              console.print("Starting live stream...")
              response = await self.goPro.http_command.webcam_start()
              if response.ok:
                  stream_url = "udp://@0.0.0.0:8554"
                  console.print(f"Streaming URL: {stream_url}")
                  return stream_url
              else:
                  console.print(f"Failed to start stream: {response}")
```

```
            except AttributeError as e:
                console.print(f"Error starting stream: {repr(e)}")
            except Exception as e:
                console.print(f"Error starting stream: {repr(e)}")
            return None

        async def run(self):
            """Runs the streamer."""
            await self.connect_goPro()
            stream_url = await self.start_stream()
            if stream_url:
                return stream_url

    def parse_arguments():
        parser = argparse.ArgumentParser(description="GoPro WiFi Command Tutorial")
        parser.add_argument("--log", type=str, default=None,
        help="Path to save the log file")
        return parser.parse_args()

    def main():
        args = parse_arguments()
        streamer = GoProStreamer(args)
        stream_url = asyncio.run(streamer.run())
        if stream_url:
            console.print(f"Stream URL: {stream_url}")

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

3. Above code will stream GoPro video over the URL “udp://@0.0.0.0:8554” and you can simply check the current streaming using VLC player.



4. If you have failed to stream from the GoPro, follow these steps, which were helpful for us in establishing a GoPro connection:

    - Change the Wi-Fi setting of GoPro from 2.4GHz to 5GHz.
    - Try using GoPro Quik App to see if you can connect your GoPro for streaming video. This process can help you quickly check the connection when installing your GoPro on site.
    - Reboot the GoPro or reset it to factory settings.

2.3 Camera Setting

1. Further, you can also change camera settings via HTTPS requests, as shown in the code below. Here is the link to the related GoPro SDK documentation. Additionally, I've included images for your reference. There are not many use cases on the web, so this information might be very helpful.

```
async def start_stream(self):
        """Starts the live stream."""
        try:
            console.print("Starting live stream...")
            response = await self.goPro.http_command.webcam_start(resolution="12"
            ,fov="4", port="8554", protocol="TS") # Set FOV to Linear
            if response.ok:
                stream_url = "udp://@0.0.0.0:8554"
                return stream_url
            else:
                console.print(f"Failed to start stream: {response}")
        except AttributeError as e:
            console.print(f"Error starting stream: {repr(e)}")
        except Exception as e:
```

```
            console.print(f"Error starting stream: {repr(e)}")
        return None
```

2. In the Benchmark UNSW project, we set the maximum resolution to 1080p and the camera angle to linear to minimize distortion and enhance homography transition, which will be described in detail in the grid setting session.

3. Vision Model

In the Benchmark UNSW project focused on urban sensing, we selected YOLO v8 as our baseline model due to its high accuracy and low inference speeds. YOLO v8 is a single-stage detection model known for its faster inference compared to state-of-the-art two-stage detection models like Faster R-CNN and Mask R-CNN. This speed is crucial for analysing streaming data from a GoPro in real-time, enabling us to process urban activity efficiently. The model achieved an inference speed of approximately 100ms on an NVIDIA Jetson, with a mean Average Precision (mAP) exceeding 99% on the training data.

We trained the YOLO v8 model using annotated frames from videos, focusing on classes such as small and large benches, as well as sitting and standing pedestrians. After training, the model was deployed to detect and record the positions of these objects over a two-week period. This data collection allowed us to measure average pedestrian dwell-time and the number of pedestrians within the observed region, providing valuable insights into how public spaces are utilised and how people interact within them.

In the following section, we will provide a detailed explanation of how to collect sample data using a GoPro streaming connection, accompanied by example code. Additionally, we will guide you through training a custom model using Roboflow, a platform that simplifies the process of managing and annotating your datasets for machine learning projects. This approach ensures that all aspects of urban sensing, from data collection to analysis, are thoroughly covered, providing a comprehensive resource for similar projects using Vision AI to study urban environments.

3.1 Collect Training Data

Once you have set up your streaming GoPro, you can collect sample data using the code snippet provided below. This script will record a 2-minute video every hour from 6 AM to 12 AM over the course of one day. With a frame rate of 1 frame per second, this approach will yield 120 frames per video, resulting in a total of 2,304 frames for labelling. If data collection is required under specific weather conditions, such as rain or snow, additional labelled data for those events may be necessary. However, based on our experience, this one-day baseline data is generally sufficient to capture the necessary objects.

```
import subprocess
import time from datetime
import datetime, timedelta

 def capture_stream_with_ffmpeg(stream_url, duration=120
 , output_directory='{output path}'):
     """Capture video stream using FFmpeg and save to file."""
     current_time = datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
     output_file = f"{output_directory}video_{current_time}.mp4"

     ffmpeg_command = [
         'ffmpeg',
         '-i', stream_url,
         '-t', str(duration),
         '-c:v', 'libx264',
         '-preset', 'fast',
         '-pix_fmt', 'yuv420p',
         output_file
     ]

     process = subprocess.Popen(ffmpeg_command)
     try:
```

```
            process.wait(timeout=duration + 5)
        except subprocess.TimeoutExpired:
            process.terminate()
            process.wait()

        print(f"Video saved as {output_file}")

def main():
    stream_url = "udp://@0.0.0.0:8554"  # Your stream URL
    duration = 120  # 2 minutes
    start_hour = 6  # 6 AM
    end_hour = 24  # 12 AM
    interval = 3600  # 1 hour in seconds

    for hour in range(start_hour, end_hour):
        print(f"Starting capture for hour {hour}")
        capture_stream_with_ffmpeg(
          f"{stream_url}?fifo_size=100000000&overrun_nonfatal=1"
          , duration=duration)
        if hour < end_hour - 1:  # No need to pause after the last capture
            print(f"Pausing for {interval / 60} minutes")
            time.sleep(interval)

    print("Completed capturing for one day.")

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

3.2 Train Model

In this project, we used the Roboflow platform to train our model. Roboflow is a popular tool for researchers

and developers due to its easy-to-use interface and excellent workflow for training models with YOLO v8. Here's a step-by-step guide on how to get started:

1. Create an Account and Sign In

Head over to the Roboflow website and sign up for an account. Once you've registered, log in to get started. This will give you access to all the tools you need for data management and model training.

2. Create an Object Detection Project

After signing in, create a new project and choose "Object Detection." This sets up your workspace to handle image data and annotations specifically for object detection tasks.

3. Upload Sample Data

Upload the sample data you collected from the collect sample data section. Make sure the images you upload are representative of what you want to detect.

4. Sample Video Frames

Extract frames from your videos at a rate of 1 frame per second. This way, you get a good number of frames without too much repetition, making your dataset both manageable and effective.

5. Label Frames

Now, it's time to label the frames. Draw bounding boxes around the objects of interest and assign them the appropriate labels. This step is crucial for teaching the model what to look for.

6. Split Data into Train/Validation/Test Sets

Divide your dataset into training, validation, and test sets. A common split is 70% for training, 20% for validation, and 10% for testing. This helps you evaluate how well your model performs on new, unseen data.

7. Process Frames

Preprocess your frames by auto-orienting them to fix any rotation issues and resizing them to 640x640 pixels. This ensures all your images are the same size, which is necessary for YOLO v8.

8. Augment Images

Enhance your dataset with image augmentation. You can flip images horizontally, adjust the saturation between -25% and 25%, and tweak the exposure from -5% to 5%. These augmentations make your model more robust.

9. Export Dataset in YOLOv8 Format

Once your data is ready, export it in the YOLOv8 format. Roboflow makes it easy to ensure your dataset is structured correctly for training.

10. Copy Dataset Download Code

Roboflow will provide a download code snippet for your dataset. Copy this code; you'll use it to download the dataset in your training environment.

11. Create a Google Colab Notebook

Open Google Colab and create a new notebook. Google Colab is a free, cloud-based tool that provides the computational power needed for training deep learning models. But for training model I would recommend to subscribe Google Colab and use other powerful machine to save your time.

12. Download Data and Modify Configuration File:

Use the download code from Roboflow to get your dataset into Colab. It will be something like code below :

```
!pip install roboflow

from roboflow import Roboflow
rf = Roboflow(api_key="APIKYE")
project = rf.workspace("WORKSPACE").project("PROJECTNAME")
version = project.version(VERSION)
dataset = version.download("BASELINEMODEL")
```

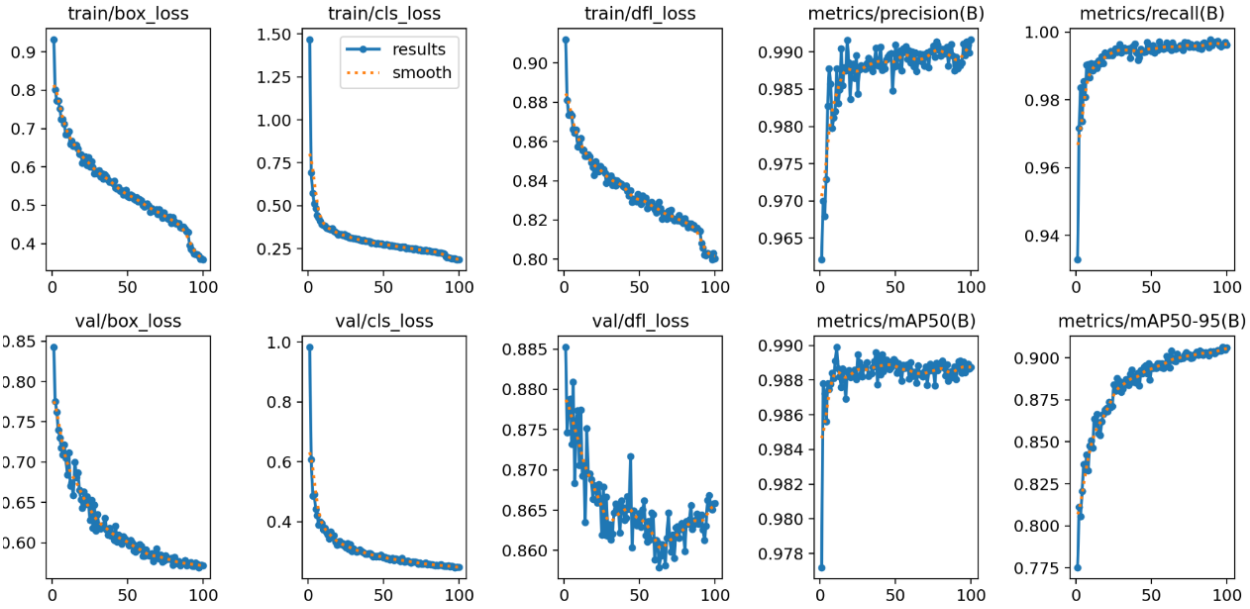
Then, update the yaml configuration file with the correct paths to your training, validation, and test data.

```
names:
- sitting
- standing
nc: 2
roboflow:
  license: CC BY 4.0
  project: benchmark_outdoor_sitting
  url: https://universe.roboflow.com/min-bagi7/benchmark_outdoor_sitting/dataset/2
  version: 2
  workspace: min-bagi7
test: ../test/images  <<< change this into actual test folder downloaded
train: Benchmark_outdoor_sitting-2/train/images
<<< change this into actual train folder downloaded
val: Benchmark_outdoor_sitting-2/valid/images
<<< change this into actual valid folder downloaded
```

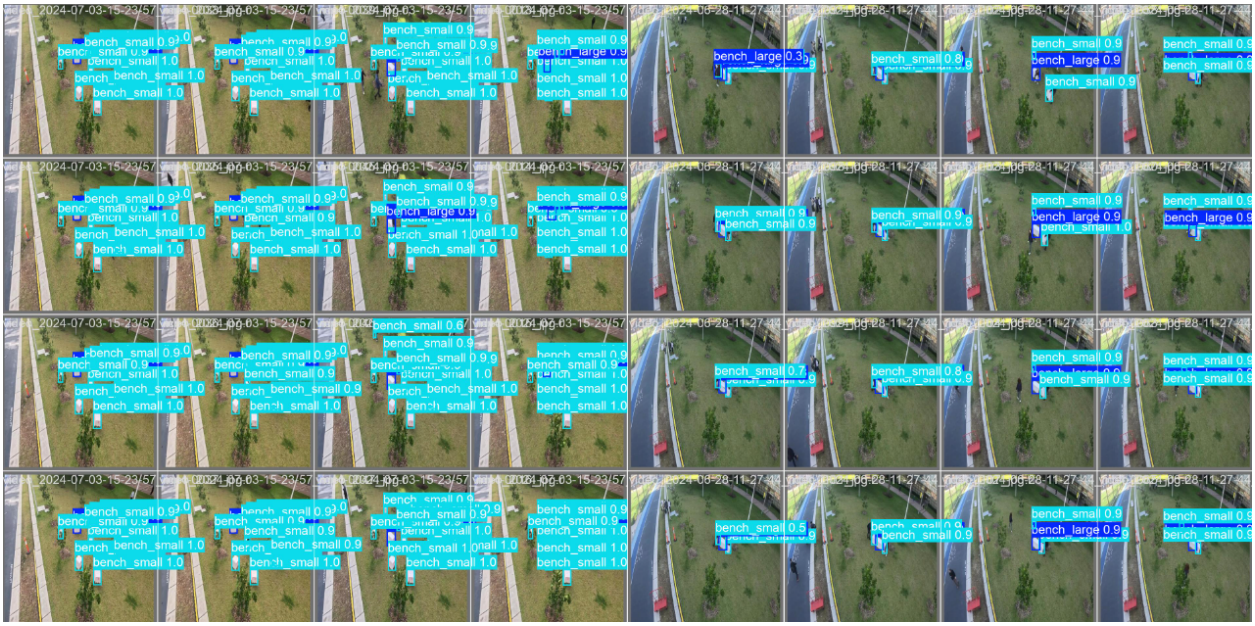
13. Train and Download Model

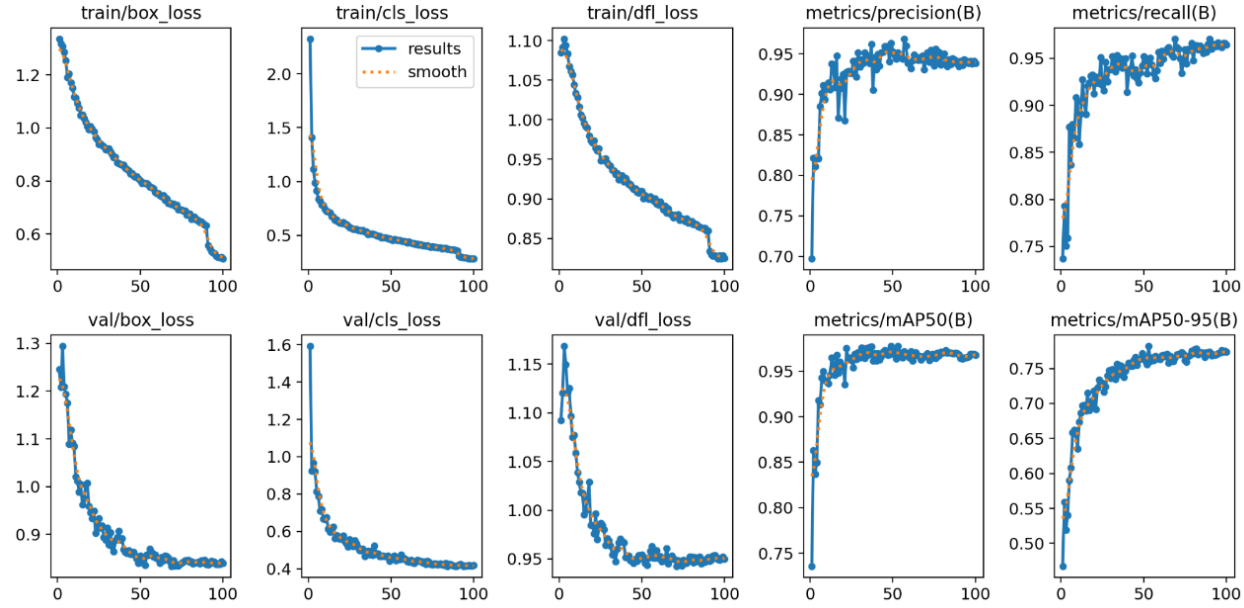
Run the training script to train your YOLOv8 model. Watch the training process and tweak any settings if necessary to get the best performance. Once training is complete, download your trained model and you're ready to go.

```
!yolo task=detect mode=train model=yolov8n.pt data={dataset.location}/data.yaml
epochs=100 imgsz=640
```



the model demonstrates excellent accuracy with a strong balance of precision, recall, and mean average precision scores, especially at the 50% IoU threshold. The slightly lower mAP@50-95 indicates some room for improvement under more stringent localization requirements, but overall the model is performing at a high level.





The model demonstrates strong performance, with precision improving from 0.697 to 0.82 and recall increasing from 0.736 to 0.877 over the epochs, indicating high accuracy in making correct predictions and identifying true positive instances. The mean Average Precision (mAP) at 0.5 IoU (mAP50) reached 0.918, reflecting excellent object detection accuracy, while the more stringent mAP50-95 metric reached 0.59, showing good performance across varying thresholds. Validation losses also consistently decreased, suggesting the model generalizes well to new data.

3.3 Run Model

1. Connect Gopro and Start Stream

```
import argparse
import asyncio
from rich.console import Console
from open_gopro import WirelessGoPro
from open_gopro.logger import setup_logging
```

```
import open_gopro.api

console = Console()

class GoProStreamer:
    def __init__(self, args):
        self.args = args
        self.logger = setup_logging(__name__, args.log)
        self.gopro = None

    async def connect_gopro(self):
        """Connects to the GoPro camera."""
        console.print("Attempting to connect via Bluetooth...")
        self.gopro = WirelessGoPro()
        await self.gopro.__aenter__()
        if self.gopro:
            console.print("Successfully connected to GoPro.")
        else:
            console.print("Failed to connect to GoPro.")

    async def start_stream(self):
        """Starts the live stream."""
        try:
            console.print("Starting live stream...")
            response = await self.gopro.http_command.webcam_start(resolution="12"
            ,fov="4", port="8554", protocol="TS") # Set FOV to Linear
            if response.ok:
                stream_url = "udp://@0.0.0.0:8554"  # Changed the port to 8555
                console.print(f"Streaming URL: {stream_url}")
                return stream_url
            else:
                console.print(f"Failed to start stream: {response}")
        except AttributeError as e:
            console.print(f"Error starting stream: {repr(e)}")
        except Exception as e:
            console.print(f"Error starting stream: {repr(e)}")
        return None

    async def run(self):
        """Runs the streamer."""
        await self.connect_gopro()
        stream_url = await self.start_stream()
        if stream_url:
            return stream_url

def parse_arguments():
    parser = argparse.ArgumentParser(description="GoPro WiFi Command Tutorial")
    parser.add_argument("--log", type=str, default=None
    , help="Path to save the log file")
    return parser.parse_args()

def main():
    args = parse_arguments()
```

```
    streamer = GoProStreamer(args)
    stream_url = asyncio.run(streamer.run())
    if stream_url:
        console.print(f"Stream URL: {stream_url}")

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

2. Run baseline model (here for yolov8n.pt)

```
import cv2
import time
from rich.console import Console
from ultralytics import YOLO

console = Console()

def load_yolo():
    model = YOLO("yolov8n.pt")  # Load YOLOv8n model from Ultralytics
    return model

def play_stream(stream_url):
    model = load_yolo()

    cap = cv2.VideoCapture(stream_url)

    if not cap.isOpened():
        console.print("[bold red]Failed to open the video stream.[/bold red]")
        return

    detection_interval = 1  # seconds
    last_detection_time = 0
    annotated_frame = None

    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            console.print("[bold red]Failed to grab frame.[/bold red]")
            break

        current_time = time.time()
        if current_time - last_detection_time >= detection_interval:
            # Perform object detection
            results = model.track(frame)

            # Render the results on the frame
            if results:
                annotated_frame = results[0].plot()
            else:
                annotated_frame = frame

            last_detection_time = current_time
```

```
        # Display the resulting frame (annotated or original)
        if annotated_frame is not None:
            cv2.imshow('GoPro Stream', annotated_frame)
        else:
            cv2.imshow('GoPro Stream', frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()

def main():
    stream_url = "udp://@0.0.0.0:8554"  # Your stream URL
    play_stream(stream_url)

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

3. Run custom model

you can run any model simply change model path into where your model is located, but the following code needed to be edited for further analsys to align your research questions.

```
import cv2
import time
from rich.console import Console
from ultralytics import YOLO

console = Console()

def load_yolo():
    model = YOLO("modelpath")  # Load YOLOv8n model from Ultralytics
    return model

def play_stream(stream_url):
    model = load_yolo()

    cap = cv2.VideoCapture(stream_url)

    if not cap.isOpened():
        console.print("[bold red]Failed to open the video stream.[/bold red]")
        return

    detection_interval = 1  # seconds
    last_detection_time = 0
    annotated_frame = None

    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            console.print("[bold red]Failed to grab frame.[/bold red]")
            break
```

```
        current_time = time.time()
        if current_time - last_detection_time >= detection_interval:
            # Perform object detection
            results = model.track(frame)

            # Render the results on the frame
            if results:
                annotated_frame = results[0].plot()
            else:
                annotated_frame = frame

            last_detection_time = current_time

        # Display the resulting frame (annotated or original)
        if annotated_frame is not None:
            cv2.imshow('GoPro Stream', annotated_frame)
        else:
            cv2.imshow('GoPro Stream', frame)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break

    cap.release()
    cv2.destroyAllWindows()

def main():
    stream_url = "udp://@0.0.0.0:8554"  # Your stream URL
    play_stream(stream_url)

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

4. References

Here is the code we actually used and successfully collected data for a month, we run three detection model asynchronously which is important to test optimisation before deploying it, here is the code for your references.

```
import cv2
import logging
import torch
import json
import asyncio
import os
import time
import numpy as np
from ultralytics import YOLO
from datetime import datetime
from concurrent.futures import ThreadPoolExecutor
import geopandas as gpd
from shapely.geometry import Point
```

```
# Set up logging
logging.basicConfig(level=logging.DEBUG
, format='%(asctime)s - %(levelname)s - %(message)s')

class GeoJSONSaver:
    def __init__(self, geojson_dir, save_interval=10, grid_path=None):
        self.geojson_dir = geojson_dir
        self.save_interval = save_interval
        self.last_save_time = time.time()
        self.grid_gdf = gpd.read_file(grid_path).to_crs("EPSG:4326") if grid_path
        else None
        self.geojson_features = []
        self.current_date = datetime.now().strftime("%Y%m%d")
        self.create_directory()
        logging.info(f"GeoJSONSaver initialized with directory: {self.geojson_dir}")

    def create_directory(self):
        os.makedirs(self.geojson_dir, exist_ok=True)
        logging.info(f"Directory created: {self.geojson_dir}")

    def get_current_geojson_path(self):
        return os.path.join(self.geojson_dir, f"features_{self.current_date}.geojson")

    def load_existing_features(self):
        geojson_path = self.get_current_geojson_path()
        if os.path.exists(geojson_path):
            with open(geojson_path, 'r') as f:
                existing_data = json.load(f)
                return existing_data.get("features", [])
        return []

    def add_feature(self, x, y, category, confidence, timestamp, object_id
    , keypoints=None):
        point_geom = Point(x, y)
        grid_id = self.get_grid_id(point_geom) if self.grid_gdf is not None else None

        feature = {
            "type": "Feature",
            "geometry": {"type": "Point", "coordinates": [x, y]},
            "properties": {
                "category": category,
                "confidence": confidence,
                "timestamp": timestamp,
                "objectID": object_id,
                "gridId": grid_id,
                "keypoints": keypoints
            }
        }
        self.geojson_features.append(feature)
        logging.debug(f"Feature added: {feature}")

    def get_grid_id(self, point_geom):
        for grid in self.grid_gdf.itertuples():
```

```
            grid_geom = grid.geometry
            if grid_geom.contains(point_geom):
                return grid.grid_id
        return None

    def save(self):
        current_time = time.time()
        current_date = datetime.now().strftime("%Y%m%d")
        if current_date != self.current_date:
            self.current_date = current_date
            self.geojson_features = self.load_existing_features()

        if current_time - self.last_save_time >= self.save_interval:
            geojson_data = {
                "type": "FeatureCollection",
                "features": self.geojson_features
            }
            geojson_path = self.get_current_geojson_path()
            with open(geojson_path, 'w') as f:
                json.dump(geojson_data, f, indent=4)
            logging.info(f"GeoJSON data saved to {geojson_path}")
            self.last_save_time = current_time

class BenchDetection:
    def __init__(self, model_path, geojson_dir, grid_path=None):
        self.model = YOLO(model_path).to('cuda')
        self.geojson_saver = GeoJSONSaver(geojson_dir, grid_path=grid_path)

    async def process_frame(self, frame):
        results = self.model.track(frame, conf=0.2, save=False, persist=True)
        annotated_frame = frame.copy()

        for result in results:
            boxes = result.boxes
            if boxes is not None:
                for detection in boxes:
                    if detection.id is None:
                        continue  # Skip detections without IDs

                    x1, y1, x2, y2 = map(int, detection.xyxy[0][:4])
                    confidence = detection.conf[0]
                    object_id = f'bench_{int(detection.id.item())}'
                    category = self.model.names[int(detection.cls)]
                    timestamp = datetime.now().isoformat()

                    color = (255, 255, 255)
                    cv2.rectangle(annotated_frame, (x1, y1), (x2, y2), color, 2)
                    label = f'{category} {confidence:.2f} ID: {object_id}'
                    cv2.putText(annotated_frame, label, (x1, y1 - 10)
                    , cv2.FONT_HERSHEY_SIMPLEX, 0.5, color, 2)

                    center_x = (x1 + x2) // 2
                    center_y = y1 + 2 * (y2 - y1) // 3
```

```
                    cv2.circle(annotated_frame, (center_x, center_y), 5, color, -1)

                    coordinates_text = f'({center_x}, {center_y})'
                    cv2.putText(annotated_frame, coordinates_text
                    , (center_x, center_y + 20), cv2.FONT_HERSHEY_SIMPLEX, 0.5
                    , (0, 255, 0), 2)

                    self.geojson_saver.add_feature(center_x, center_y, category
                    , confidence.item(), timestamp, object_id)

        return annotated_frame

class PedDetection:
    def __init__(self, model_path, geojson_dir, grid_path=None):
        self.model = YOLO(model_path).to('cuda')
        self.geojson_saver = GeoJSONSaver(geojson_dir, grid_path=grid_path)
        self.keypoint_pairs = {
            'head': [(0, 1), (0, 2), (1, 3), (2, 4)],
            'body': [(5, 6), (6, 12), (5, 11), (12, 11)],
            'arms': [(6, 8), (8, 10), (5, 7), (7, 9)],
            'legs': [(12, 14), (14, 16), (11, 13), (13, 15)]
        }
        self.colors = {
            'head': (255, 0, 255),
            'body': (0, 0, 255),
            'arms': (0, 0, 255),
            'legs': (255, 255, 0)
        }

    async def process_frame(self, frame):
        results = self.model.track(frame, conf=0.2, save=False, persist=True)
        annotated_frame = frame.copy()

        for result in results:
            boxes = result.boxes
            keypoints = result.keypoints
            if boxes is not None and keypoints is not None and
            isinstance(keypoints.data, torch.Tensor):
                keypoints_data = keypoints.data.cpu().numpy().tolist()

                for detection, person_keypoints in zip(boxes, keypoints_data):
                    if len(person_keypoints) < 17:
                        continue

                    x1, y1, x2, y2 = map(int, detection.xyxy[0][:4])
                    confidence = detection.conf[0]
                    object_id = f'ped_{int(detection.id.item()) if detection.id is
                    not None else -1}'
                    category = self.model.names[int(detection.cls)]
                    timestamp = datetime.now().isoformat()

                    cv2.rectangle(annotated_frame, (x1, y1), (x2, y2), (0, 255, 0), 2)
                    label = f'ID: {object_id} {confidence:.2f}'
```

```
                    cv2.putText(annotated_frame, label, (x1, y1 - 10)
                    , cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 0), 2)

                    for part, pairs in self.keypoint_pairs.items():
                        line_color = self.colors[part]
                        for start_idx, end_idx in pairs:
                            if start_idx < len(person_keypoints)
                            and end_idx < len(person_keypoints):
                                if person_keypoints[start_idx][2] > 0.5
                                and person_keypoints[end_idx][2] > 0.5:
                                    start_point = (int(person_keypoints[start_idx][0])
                                    , int(person_keypoints[start_idx][1]))
                                    end_point = (int(person_keypoints[end_idx][0])
                                    , int(person_keypoints[end_idx][1]))
                                    cv2.line(annotated_frame, start_point, end_point
                                    , line_color, 3)

                    bottom_keypoints = sorted([kp for kp in person_keypoints
                    if kp[2] > 0.5], key=lambda k: k[1], reverse=True)[:2]
                    if len(bottom_keypoints) == 2:
                        mid_y = int((bottom_keypoints[0][1]
                        + bottom_keypoints[1][1]) / 2)
                        valid_keypoints = [kp for kp in person_keypoints if kp[2] > 0.5]
                        left_keypoint = min(valid_keypoints, key=lambda k: k[0]
                        , default=None)
                        right_keypoint = max(valid_keypoints, key=lambda k: k[0]
                        , default=None)
                        if left_keypoint is not None and right_keypoint is not None:
                            mid_x = int((left_keypoint[0] + right_keypoint[0]) / 2)
                            cv2.circle(annotated_frame, (mid_x, mid_y), 3
                            , (255, 255, 255), -1)
                            cv2.putText(annotated_frame, f'({mid_x}, {mid_y})'
                            , (mid_x, mid_y + 20), cv2.FONT_HERSHEY_SIMPLEX, 0.3
                            , (0, 255, 0), 1)

                            self.geojson_saver.add_feature(mid_x, mid_y, category
                            , confidence.item(), timestamp, object_id, person_keypoints)

        return annotated_frame

class SittingDetection:
    def __init__(self, model_path_pose, model_path_sitting, geojson_dir, grid_path=None):
        self.model_pose = YOLO(model_path_pose).to('cuda')
        self.model_sitting = YOLO(model_path_sitting).to('cuda')
        self.geojson_saver = GeoJSONSaver(geojson_dir, grid_path=grid_path)
        self.keypoint_pairs = {
            'head': [(0, 1), (0, 2), (1, 3), (2, 4)],
            'body': [(5, 6), (6, 12), (5, 11), (12, 11)],
            'arms': [(6, 8), (8, 10), (5, 7), (7, 9)],
            'legs': [(12, 14), (14, 16), (11, 13), (13, 15)]
        }
        self.colors = {
            'head': (255, 0, 255),
```

```
            'body': (0, 0, 255),
            'arms': (0, 0, 255),
            'legs': (255, 255, 0)
        }

    async def process_frame(self, frame):
        pose_results = self.model_pose.track(frame, conf=0.2, save=False, persist=True)
        midpoints = []
        keypoints_list = []
        annotated_frame = frame.copy()

        for pose_result in pose_results:
            keypoints = pose_result.keypoints
            if keypoints is not None and isinstance(keypoints.data, torch.Tensor):
                keypoints_data = keypoints.data.cpu().numpy().tolist()
                for person_keypoints in keypoints_data:
                    keypoints_list.append(person_keypoints)
                    if len(person_keypoints) < 17:
                        continue
                    for part, pairs in self.keypoint_pairs.items():
                        line_color = self.colors[part]
                        for start_idx, end_idx in pairs:
                            if start_idx < len(person_keypoints)
                            and end_idx < len(person_keypoints):
                                if person_keypoints[start_idx][2] > 0.5
                                and person_keypoints[end_idx][2] > 0.5:
                                    start_point = (int(person_keypoints[start_idx][0])
                                    , int(person_keypoints[start_idx][1]))
                                    end_point = (int(person_keypoints[end_idx][0])
                                    , int(person_keypoints[end_idx][1]))
                                    cv2.line(annotated_frame, start_point, end_point
                                    , line_color, 3)
                    bottom_keypoints = sorted([kp for kp in person_keypoints if kp[2]
                    > 0.6], key=lambda k: k[1], reverse=True)[:2]
                    if len(bottom_keypoints) == 2:
                        mid_y = int((bottom_keypoints[0][1] + bottom_keypoints[1][1])
                        / 2)
                        valid_keypoints = [kp for kp in person_keypoints if kp[2] > 0.6]
                        left_keypoint = min(valid_keypoints, key=lambda k: k[0]
                        , default=None)
                        right_keypoint = max(valid_keypoints, key=lambda k: k[0]
                        , default=None)
                        if left_keypoint is not None and right_keypoint is not None:
                            mid_x = int((left_keypoint[0] + right_keypoint[0]) / 2)
                            midpoints.append((mid_x, mid_y))

        sitting_results = self.model_sitting.track(annotated_frame, conf=0.2, save=False
        , persist=True)
        for result in sitting_results:
            boxes = result.boxes
            if boxes is not None:
                for detection in boxes:
                    x1, y1, x2, y2 = map(int, detection.xyxy[0][:4])
```

```
                    confidence = detection.conf[0]
                    if detection.id is None:
                        continue  # Skip detections without IDs
                    object_id = int(detection.id.item()) if hasattr(detection, 'id')
                    else -1
                    category = self.model_sitting.names[int(detection.cls)]
                    timestamp = datetime.now().isoformat()

                    color = (0, 255, 0) if category == 'sitting' else (255, 0, 0)

                    cv2.rectangle(annotated_frame, (x1, y1), (x2, y2), color, 2)
                    label = f'{category} ID: {object_id} {confidence:.2f}'
                    cv2.putText(annotated_frame, label, (x1, y1 - 10)
                    , cv2.FONT_HERSHEY_SIMPLEX, 0.3, color, 1)

                    closest_midpoint = self.find_closest_midpoint(midpoints, (x1 + x2)
                    // 2, (y1 + y2) // 2)
                    if closest_midpoint is not None:
                        mid_x, mid_y = closest_midpoint
                        cv2.circle(annotated_frame, (mid_x, mid_y), 3, color, -1)
                        self.geojson_saver.add_feature(mid_x, mid_y, category
                        , confidence.item(), timestamp, object_id, keypoints_list)
                        coordinates_text = f'({mid_x}, {mid_y})'
                        cv2.putText(annotated_frame, coordinates_text, (mid_x, mid_y
                        + 40), cv2.FONT_HERSHEY_SIMPLEX, 0.3, (0, 255, 0), 1)

        return annotated_frame

    def find_closest_midpoint(self, midpoints, x, y):
        min_distance = float('inf')
        closest_midpoint = None
        for (mx, my) in midpoints:
            distance = np.sqrt((mx - x) ** 2 + (my - y) ** 2)
            if distance < min_distance:
                min_distance = distance
                closest_midpoint = (mx, my)
        return closest_midpoint

async def run_detection(detection_class, frame):
    return await detection_class.process_frame(frame)

async def frame_producer(queue, cap, max_retries, crop_coords):
    retry_count = 0
    x_start, y_start, width, height = crop_coords
    while cap.isOpened():
        ret, frame = cap.read()
        if not ret:
            retry_count += 1
            logging.error(f"Failed to grab frame")
            if retry_count >= max_retries:
                logging.error("Max retries reached. Exiting.")
                break
            await asyncio.sleep(1)  # Wait a bit before retrying
```

```
            continue

        retry_count = 0  # Reset retry count after a successful frame grab
        cropped_frame = frame[y_start:y_start+height, x_start:x_start+width]

        if queue.qsize() < 1:  # Limit queue size to 1 to ensure frame skipping
            await queue.put(cropped_frame)
        await asyncio.sleep(0)  # Yield control to allow other coroutines to run

    await queue.put(None)  # Signal the consumer to stop

async def frame_consumer(queue, bench_detector, ped_detector, sitting_detector
, image_output_dir):
    while True:
        frame = await queue.get()
        if frame is None:
            break

        start_time = time.time()

        tasks = [
            run_detection(bench_detector, frame),
            run_detection(ped_detector, frame),
            run_detection(sitting_detector, frame)
        ]
        results = await asyncio.gather(*tasks)

        # Calculate duration of detection tasks
        duration = time.time() - start_time

        # Process results
        bench_frame, ped_frame, sitting_frame = results
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")

        bench_image_path = os.path.join(image_output_dir, "bench"
        , f"frame_{timestamp}.jpg")
        ped_image_path = os.path.join(image_output_dir, "ped", f"frame_{timestamp}.jpg")
        sitting_image_path = os.path.join(image_output_dir, "sitting"
        , f"frame_{timestamp}.jpg")

        # Save GeoJSON data
        bench_detector.geojson_saver.save()
        ped_detector.geojson_saver.save()
        sitting_detector.geojson_saver.save()

        logging.info(f"Detection and processing took {duration:.2f} seconds.")

        # Add a 1-second interval before processing the next frame
        await asyncio.sleep(3)

        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
```

```
    cv2.destroyAllWindows()

async def process_stream(bench_detector, ped_detector, sitting_detector, stream_url
, image_output_dir):
    cap = cv2.VideoCapture(f"{stream_url}?fifo_size=100000000&overrun_nonfatal=1")
    if not cap.isOpened():
        logging.error("Error opening video stream")
        return

    queue = asyncio.Queue(maxsize=1)
    max_retries = 5
    crop_coords = (420, 200, 1920-420, 1080-200)  # Crop coordinates (x_start, y_start
    , width, height)

    producer_task = asyncio.create_task(frame_producer(queue, cap, max_retries
    , crop_coords))
    consumer_task = asyncio.create_task(frame_consumer(queue, bench_detector
    , ped_detector, sitting_detector, image_output_dir))

    await asyncio.gather(producer_task, consumer_task)

    cap.release()
    cv2.destroyAllWindows()


if __name__ == "__main__":
    bench_detector = BenchDetection('modelpath', 'resultpath', grid_path='gridpath')
    ped_detector = PedDetection('modelpath', 'resultpath', grid_path='gridpath')
    sitting_detector = SittingDetection('modelpath', 'modelpath2', 'resultpath'
    , grid_path='gridpath)
    stream_url = "udp://@0.0.0.0:8554"
    image_output_dir = 'image-output-dir'
    asyncio.run(process_stream(bench_detector, ped_detector, sitting_detector
    , stream_url , image_output_dir))
```

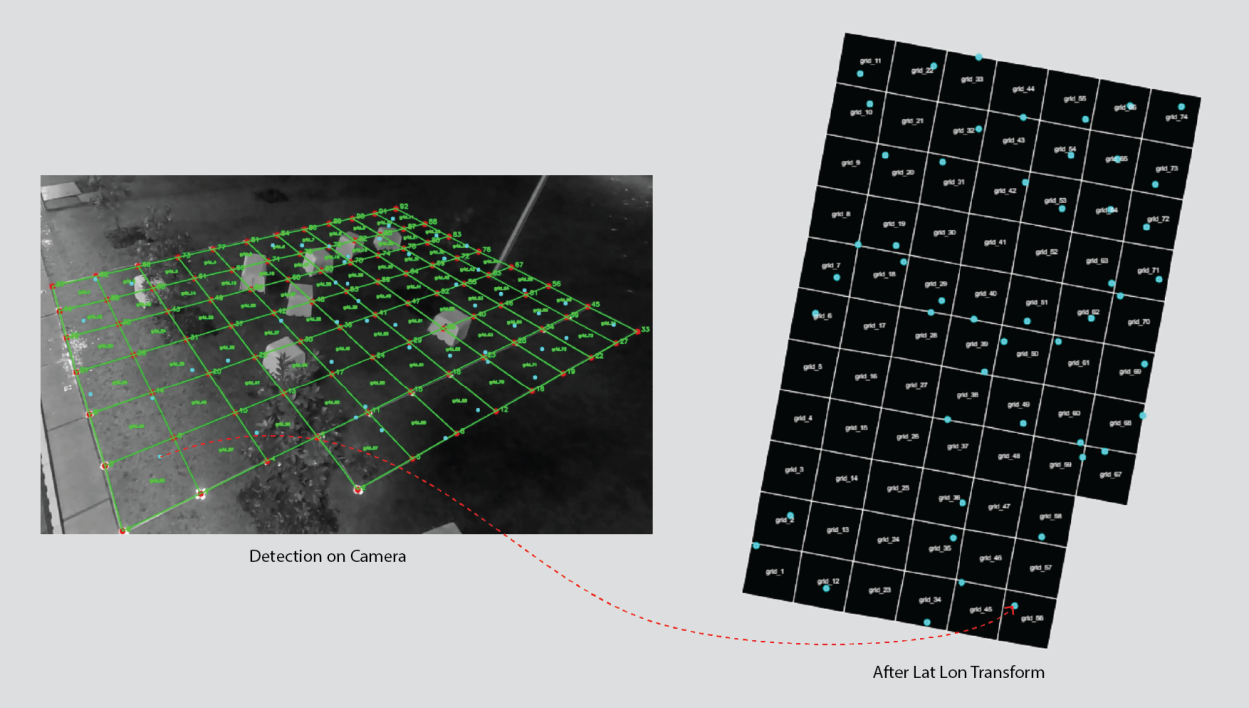
4. Study Area Setting

4.1 Detection Area Boundary

The detection area depends on several factors, including the camera's installation height, the objects being detected, and the camera's resolution. As a result, the detection area can vary by project. Below, we outline the specifications for our detection area, which may be useful for anyone pursuing a similar project.

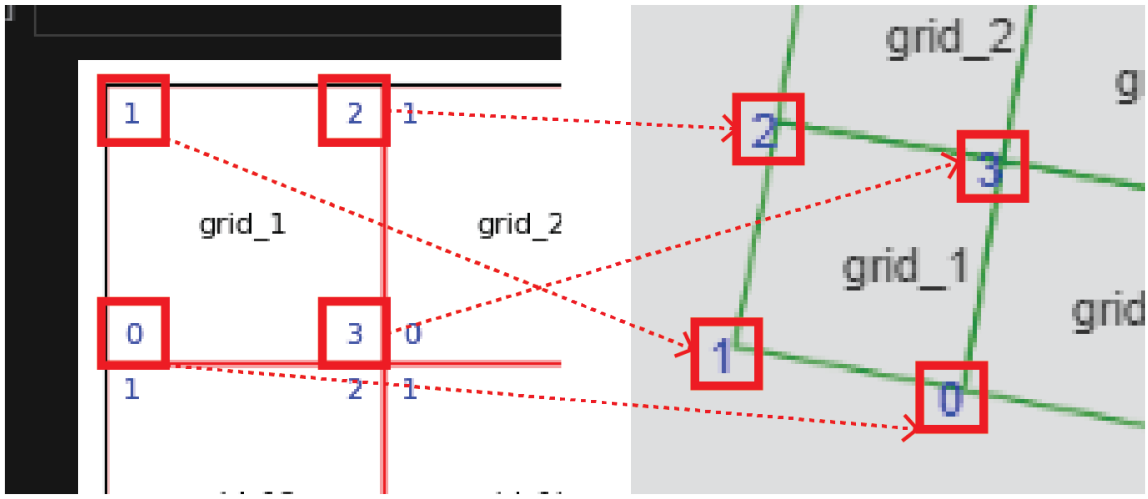
In the Benchmark NSW project, we installed the GoPro at a height of approximately 5 to 6 meters. With this setup, we were able to effectively monitor an area of around 100 square meters.

4.2 Setup Grid & Grid transition



A key component of this project involves converting camera-based detections into usable coordinate data by applying a transformation matrix. However, a challenge arises due to the minimal latitude and longitude variance in each detection, which prevents us from using OpenCV's built-in getPerspectiveTransform function. This is because it relies on float32 precision, which is insufficient for accurate coordinate transitions. To address this, we implemented a custom function that calculates the transformation matrix using float64 precision. If you're interested in learning more about this approach, you can refer to the linked blog post

Before running the code, two grid systems need to be manually created: the original grid, which represents the camera view, and the target grid, represented in a GeoJSON file, which corresponds to the map coordinates. For this project, we marked a 1m x 1m grid on-site, captured reference images, and used them as a baseline. The target grid was created using QGIS software and exported with lat-long coordinates in the appropriate CRS format.



The code below generates multiple transformation matrices for each grid_id. The number of grids may vary from project to project, but you can use the code as a starting point.

```
import json
import numpy as np

# Load the JSON data
origin_grid = 'origin_grids.json'
target_grid = 'target_grids.geojson'

# Load the geojson files
with open(origin_grid, 'r') as file:
    origin_grid_data = json.load(file)

with open(target_grid, 'r') as file:
    target_grid_data = json.load(file)

# Function to extract grid coordinates by grid_id
def get_grid_coordinates_from_json(data, grid_id):
    for grid in data:
        if grid['grid_id'] == grid_id:
            return grid['corners']
    return None

# Function to extract grid coordinates by grid_id from geojson
def get_grid_coordinates_from_geojson(geojson, grid_id):
    for feature in geojson['features']:
        if feature['properties']['grid_id'] == grid_id:
            return feature['geometry']['coordinates'][0]
    return None

def compute_perspective_transform(src, dst):
    assert src.shape == (4, 2)
    assert dst.shape == (4, 2)
```

```
    A = []
    B = []

    for i in range(4):
        x, y = src[i][0], src[i][1]
        u, v = dst[i][0], dst[i][1]
        A.append([x, y, 1, 0, 0, 0, -u*x, -u*y])
        A.append([0, 0, 0, x, y, 1, -v*x, -v*y])
        B.append(u)
        B.append(v)

    A = np.array(A, dtype=np.float64)
    B = np.array(B, dtype=np.float64)

    H = np.linalg.solve(A, B)
    H = np.append(H, 1).reshape((3, 3))
    return H

# Dictionary to store transformation matrices and transformed polygons
transformation_matrices = {}
transformed_polygons = []

# Iterate through each grid feature in plan_grid_data
for grid in origin_grid_data:
    grid_id = grid['grid_id']
    origin_grid_coords = get_grid_coordinates_from_json(origin_grid_data, grid_id)
    target_grid_coords = get_grid_coordinates_from_geojson(target_grid_data, grid_id)

    if grid_coords is None or target_grid_coords is None:
        print(f"Skipping {grid_id} due to missing coordinates.")
        continue

    # Convert to numpy arrays with high precision
    src_pts = np.array(origin_grid_coords, dtype=np.float64)
    dst_pts = np.array(target_grid_coords[:-1], dtype=np.float64)

    # Compute the perspective transformation matrix
    M = compute_perspective_transform(src_pts, dst_pts)

    # Store the matrix in the dictionary
    transformation_matrices[grid_id] = M.tolist()

    # Apply the transformation to the source points to verify
    src_pts_homogeneous = np.hstack([src_pts, np.ones((4, 1), dtype=np.float64)])
    transformed_src_pts_homogeneous = src_pts_homogeneous @ M.T
    transformed_src_pts = (transformed_src_pts_homogeneous[:, :2].T
    / transformed_src_pts_homogeneous[:, 2]).T

    # Print debug information
    print(f"Grid ID: {grid_id}")
    print("Source Points (plan_grid_corners.json):")
    print(src_pts)
    print("Destination Points (map_grid.geojson):")
```

```
    print(dst_pts)
    print("Transformation Matrix:")
    print(M)
    print("Transformed Source Points:")
    print(transformed_src_pts)
    print("Difference (Transformed - Destination):")
    print(transformed_src_pts - dst_pts)
    print()

    # Create a new polygon feature with the transformed source points
    transformed_polygon = {
        "type": "Feature",
        "properties": {
            "grid_id": grid_id
        },
        "geometry": {
            "type": "Polygon",
            "coordinates": [transformed_src_pts.tolist() + [transformed_src_pts
            .tolist()[0]]]  # Close the polygon by repeating the first point
        }
    }
    transformed_polygons.append(transformed_polygon)

# Create a new GeoJSON feature collection for the transformed polygons
transformed_geojson = {
    "type": "FeatureCollection",
    "features": transformed_polygons
}

# Save the transformation matrices and transformed polygons to JSON files
with open('transformation_matrices.json', 'w') as outfile:
    json.dump(transformation_matrices, outfile, indent=4)


print("Transformation matrices saved to transformation_matrices.json")
```

Once you've obtained the transformation matrices, you can test the coordinate transitions using the code provided below.

```
import json
import numpy as np

# Load the transformation matrices
transformation_matrices = 'transformation_matrices.json'
with open(transformation_matrices, 'r') as file:
    transformation_matrices = json.load(file)

# Load the GeoJSON points file
points_geojson = 'detection_point_data.geojson'
with open(points_geojson, 'r') as file:
    points_geojson = json.load(file)

# Function to apply transformation matrix to a point
```

```
def apply_transformation(matrix, point):
    point_homogeneous = np.array([point[0], point[1], 1], dtype=np.float64)
    transformed_point_homogeneous = point_homogeneous @ matrix.T
    transformed_point = (transformed_point_homogeneous[:2]
    / transformed_point_homogeneous[2]).tolist()
    return transformed_point

# Iterate through each point feature in the GeoJSON file and apply the transformation
transformed_features = []
for feature in points_geojson['features']:
    grid_id = feature['properties'].get('gridId')
    if grid_id and grid_id in transformation_matrices:
        matrix = np.array(transformation_matrices[grid_id], dtype=np.float64)
        original_point = feature['geometry']['coordinates']
        transformed_point = apply_transformation(matrix, original_point)

        # Create a new feature with the transformed point
        transformed_feature = {
            "type": "Feature",
            "properties": feature['properties'],
            "geometry": {
                "type": "Point",
                "coordinates": transformed_point
            }
        }
        transformed_features.append(transformed_feature)

# Create a new GeoJSON feature collection for the transformed points
transformed_points_geojson = {
    "type": "FeatureCollection",
    "features": transformed_features
}

# Save the transformed points to a new GeoJSON file
transformed_points_geojson = 'lat_lon_data.geojson'
with open(transformed_points_geojson, 'w') as outfile:
    json.dump(transformed_points_geojson, outfile, indent=4)

print(f"Transformed points saved to {transformed_points_geojson}")
```

4.3 Sensor Installation (Network & Electricity)

To support this sensor kit, you will need at least one power source to supply power to three devices: a GoPro, an Nvidia Jetson, and a mobile hotspot router. It is crucial that the mobile hotspot router includes an Ethernet port; without this, the connection between the GoPro and Nvidia Jetson may become unstable. We highly recommend using a hotspot router with an Ethernet port for this set-up. In this project, we used a device recommended by a local provider.

5. Design Movable Bench

5.1 Design Criteria

Mobility

- Movable furniture units are essential in studying human interaction and movement within a public space. This design approach allows us to observe how people align seating more fluidly and naturally to maximize social interaction.
- Good public spaces offer individuals a choice of where and how they would like to sit. The bench design is lightweight, enabling visitors to easily move them and select their ideal spot. By studying how people reposition the benches, we gain insights into how they cluster seating to foster social interactions.

Modular

- The design is intended to be modular, with units that can easily connect to encourage playfulness on site. With sufficient modules, the system can be extended to form a long bench for lying down or a circular gathering spot for a group of friends.

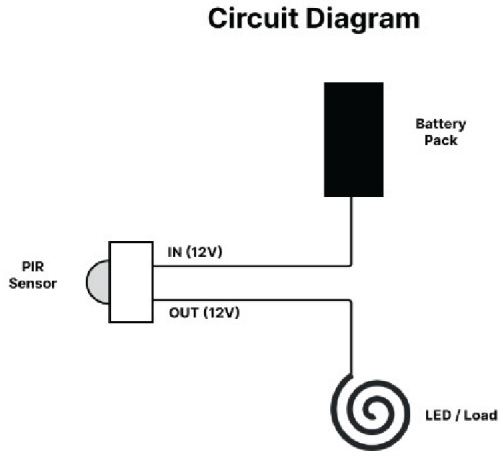
Colour, Texture and Material

- It is crucial that the benches have a high colour contrast with their background. This feature makes the benches more distinguishable from their surroundings, facilitating easier differentiation and recognition by detection models.
- Textures can provide additional details that help in distinguishing the benches. Select materials and textures that maintain visibility under varying lighting conditions(use materials that enhance visual recognition during night-time).
- Light the bench when someone is sitting on it. Make this light distinct enough that it can be easily detected by the camera.

Shape, Size, and Proportion

- We aim for unique and consistent shapes that are easy to recognize and distinguish from other objects in the environment.
- The benches should be large enough to be clearly captured by camera sensors from a distance. Maintaining consistent proportions from various angles can also aid the model in learning and detecting the benches across different scenes.

5.2 Motion detection light



The motion detection sensor uses a PIR (passive infra-red sensor) to observe objects passing across its lens. When the PIR triggers, it sends voltage to a connected LED strip indicating that an object was detected. PIR sensors are made up of two pyroelectric elements which generate a voltage difference when there is a difference in the intensity of the infra-red radiation between the two elements. Therefore, when the amount of incoming infra-red radiation is the same (i.e. both elements receiving the ambient temperature), there will be no detection. This leads to less false detections if, for example, the sensor was exposed to a flash of light. Only when an object passes in front of the plane parallel to the motion detector will a voltage be outputted. The voltage passes to the LED strip, illuminating it and acknowledges the observation of a passing object.

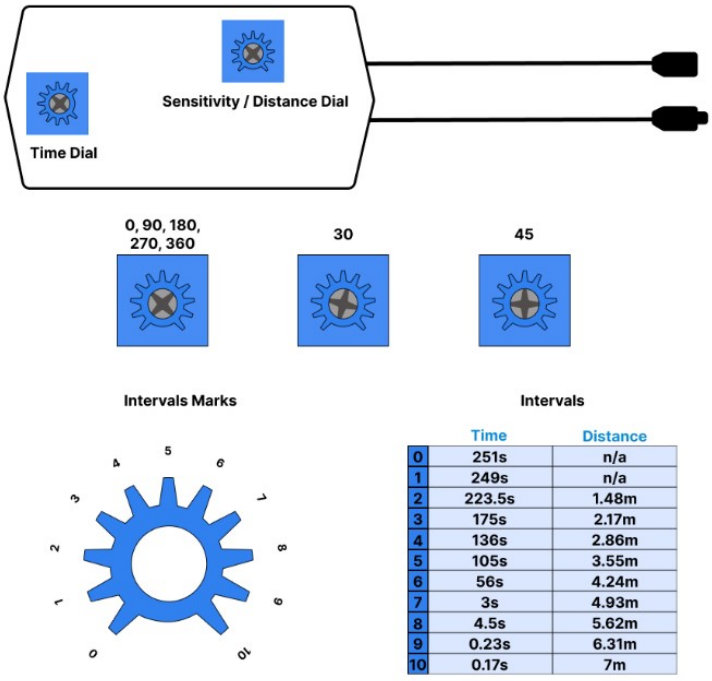
Required Items

- Sensky DC 12V PIR Sensor (link)
- Talentcell Rechargeable 12V Lithium-Ion Battery Pack (link)
- Male to Male DC Cable (included in a Battery pack)
- DC LED Light Strip (link)
- (additional) needs to consider waterproofing

Setup Guide

1. Set PIR Sensor Settings: Adjust the time delay and sensitivity of the sensor as desired by rotating the appropriate dials. Refer to the PIR sensor diagram for guidance.

PIR Sensor Guide



	Time	Distance
0	251s	n/a
1	249s	n/a
2	223.5s	1.48m
3	175s	2.17m
4	136s	2.86m
5	105s	3.55m
6	56s	4.24m
7	3s	4.93m
8	4.5s	5.62m
9	0.23s	6.31m
10	0.17s	7m

2. Set-up LED: Affix the LED strip along the wall or surface. Avoid leaving the LED strip in the reel for extended periods of time as it may cause overheating.
3. Mount Sensor: Position the PIR Sensor in a stable location Ensure it has a clear line of sight to the area you wish to monitor. Avoid placing it in areas with continuous movement to prevent excessive triggering.
4. Connect Sensor: Use a male to male DC cable to connect the PIR Sensor to the battery pack. Connect the female input connector of the PIR Sensor to one of the male connectors of the cable, and attach the other male connector of the cable to the battery pack. Join the male output connector of the PIR sensor with the female input connector of the LED strip.
5. Activate and Test Set-up: Flip the switch on the battery pack to activate the motion detection system. Walk in front of the PIR sensor within the set distance range to verify that the sensor triggers the LED strip as expected.
6. Demo video:

- Assembling instruction video
- Lighting demo video

5.3 Benchmark NSW Design

The Benchmark NSW design process centred around the creation of public outdoor seating that is both functional and aesthetically engaging, while also integrating smart technology, safety, and sustainability. The process unfolded in several stages, each involving collaboration between UNSW Industrial Design students and staff and advanced urbanism experts from MIT. From concept development to final fabrication, the design process emphasised inclusivity, flexibility, and technological innovation.

Co-design process

- Phase 1 : Project Briefing & Scope Definition

  This phase focused on introducing the project objectives, scope, and constraints. The co-designers were briefed on the challenge: designing smart seating to improve safety for women and girls in public spaces. Key considerations such as user experience, material choices, and environmental sustainability were highlighted, setting the foundation for their independent research and ideation.

  The 1-hour project briefing was held with the project team and the co-designers.

  To meet the objectives of the project, the following design criteria were established for the seating solutions:

  – Mobility and flexibility. The seating must be lightweight and easily movable so that users can reposition the units as needed. This flexibility will foster social interactions and enable users to arrange the seating in ways that suit their preferences, enhancing the usability and social dynamics of the space.

- Modularity. The seating must be modular, allowing for easy reconfiguration into various forms such as long benches or circular setups for groups. This adaptability encourages playfulness and accommodates the diverse ways that people use public spaces.
- Visibility and material use. The seating must be highly visible, especially at night, to enhance safety. Materials should be chosen for their ability to stand out in different lighting conditions. Additionally, the seating must include a distinctive light that activates when the seating is in use, improving detectability.
- Shape, size, and proportion. The seating must have unique, recognisable shapes that are easy to identify from various angles, aiding surveillance and safety monitoring. The design should balance engaging aesthetics with durability, ensuring that the seating is robust yet inviting.
- Technology integration. The seating should include Ultra-Wideband (UWB) sensors to detect accurate bench locations in real-time, contributing to management and user safety (note: the final sensor kit ended up not needing this technology). Other technologies could also be integrated to enhance interactivity and functionality, such as motion-activated lighting.
- Ecological impact. The seating must be designed with sustainability in mind, using materials with minimal environmental impact. Post-consumer recycled plastics preferred to ensure durability, weather resistance, and alignment with circular economy principles.
- Passive surveillance. The seating design must not obstruct sightlines, promoting visibility and natural monitoring of the environment by users and passersby, contributing to a sense of safety in the public space.
- Inclusive and accessible. The seating must include ergonomic features that ensure comfort and prolonged use. It must cater to a diverse range of users, with designs that prevent falls and ensure accessibility for people of different physical abilities.
- Aesthetic and cultural integration. The design must reflect Sydney's urban character and cultural values. It should blend into public spaces while providing a fun and engaging addition to the urban landscape.

The success of the project was evaluated based on the following criteria:

- Innovation and originality in design. The extent to which the design introduces new ideas or innovative approaches to public seating.
- Effectiveness in responding to the needs of target groups. The degree to which the design addresses the specific behaviours and requirements of local women, girls, and gender diverse people.
- Aesthetic and functional integration into public spaces. How well the seating complements the surrounding environment, both visually and functionally.
- Practicality of the design. The ease with which the seating can be manufactured, installed, and maintained.
- Ease of fabrication using local facilities. The feasibility of producing the seating using local resources and available manufacturing technologies.

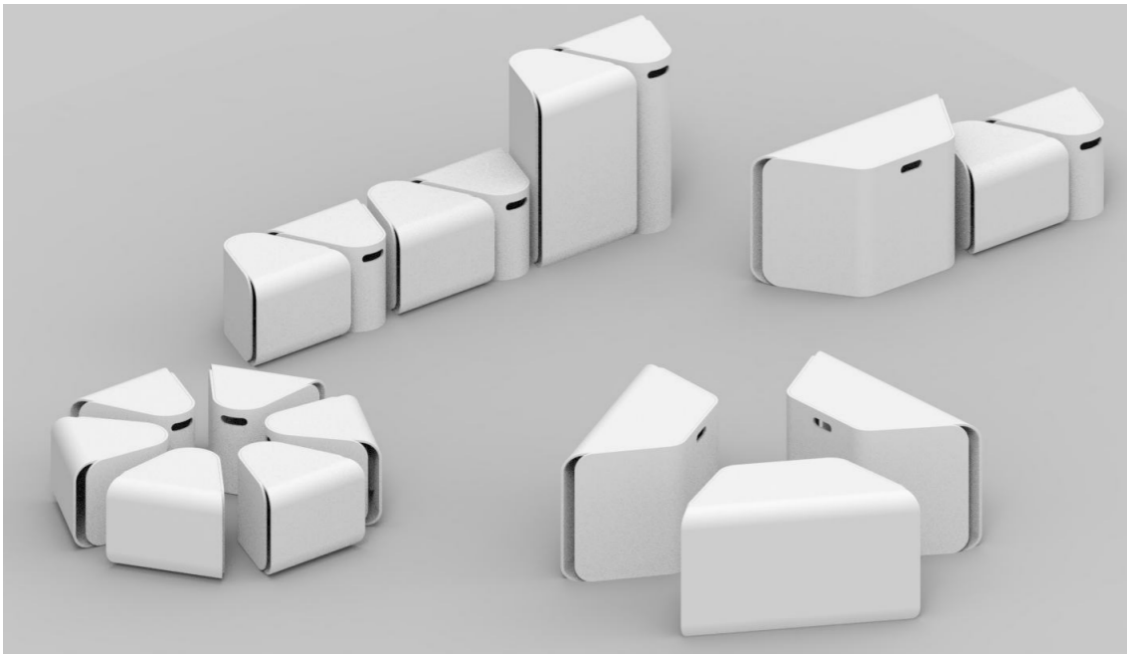
- Phase 2 : Contextual Research & Initial Concept Presentation

  Following the establishment of the design criteria, this phase saw the development of a variety of seating solutions that aimed to meet the project's goals of inclusivity, perceptions of safety, and interaction. Each concept was driven by the need to create flexible, engaging public seating that reflects the values of the Safer Cities Program while also integrating smart technology and sustainable practices.

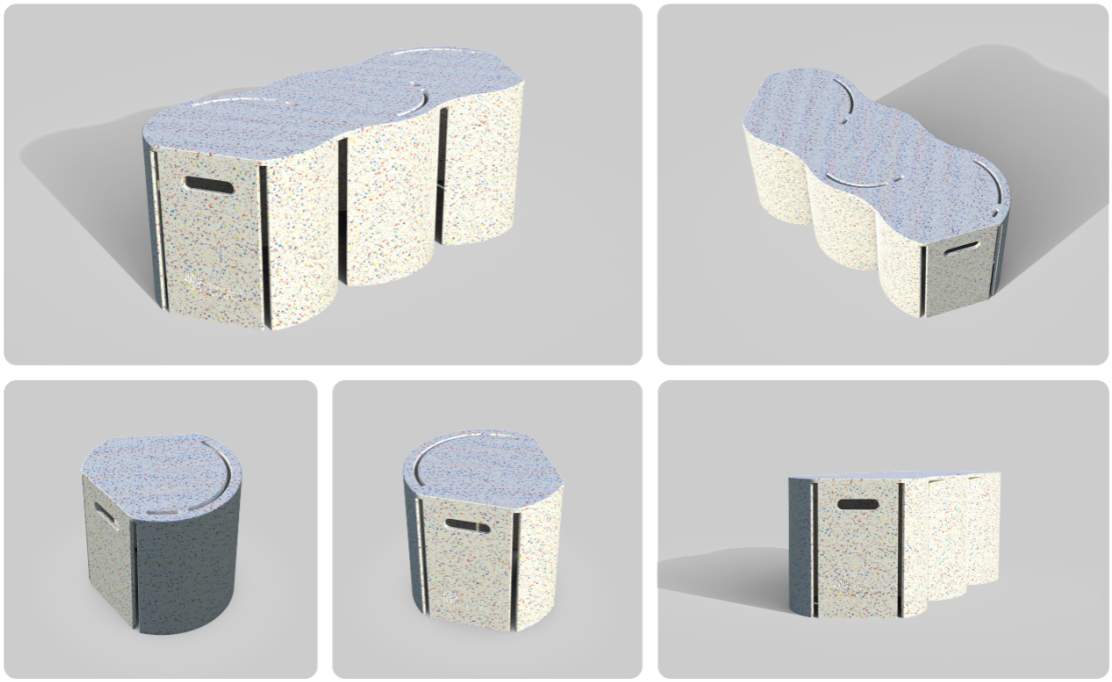
  After conducting individual research—including site visits and distilling inspirations into mood boards—the co-designers presented their insights alongside their initial concepts at the end of May

2024.

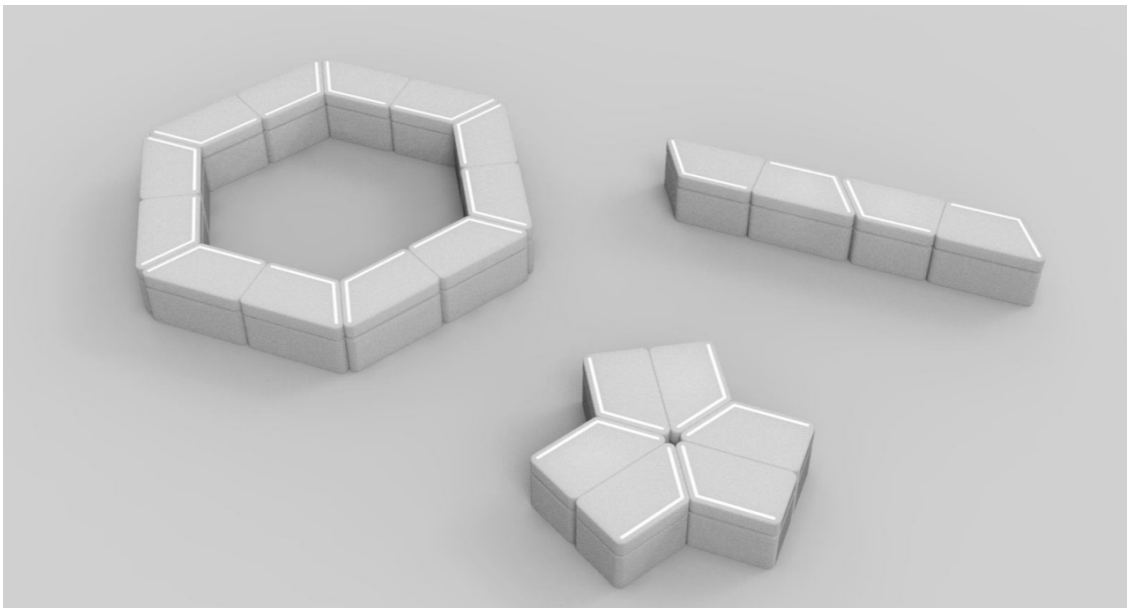
– concept design render by Eleanor Tang



– concept design render by Eugenia Cheung



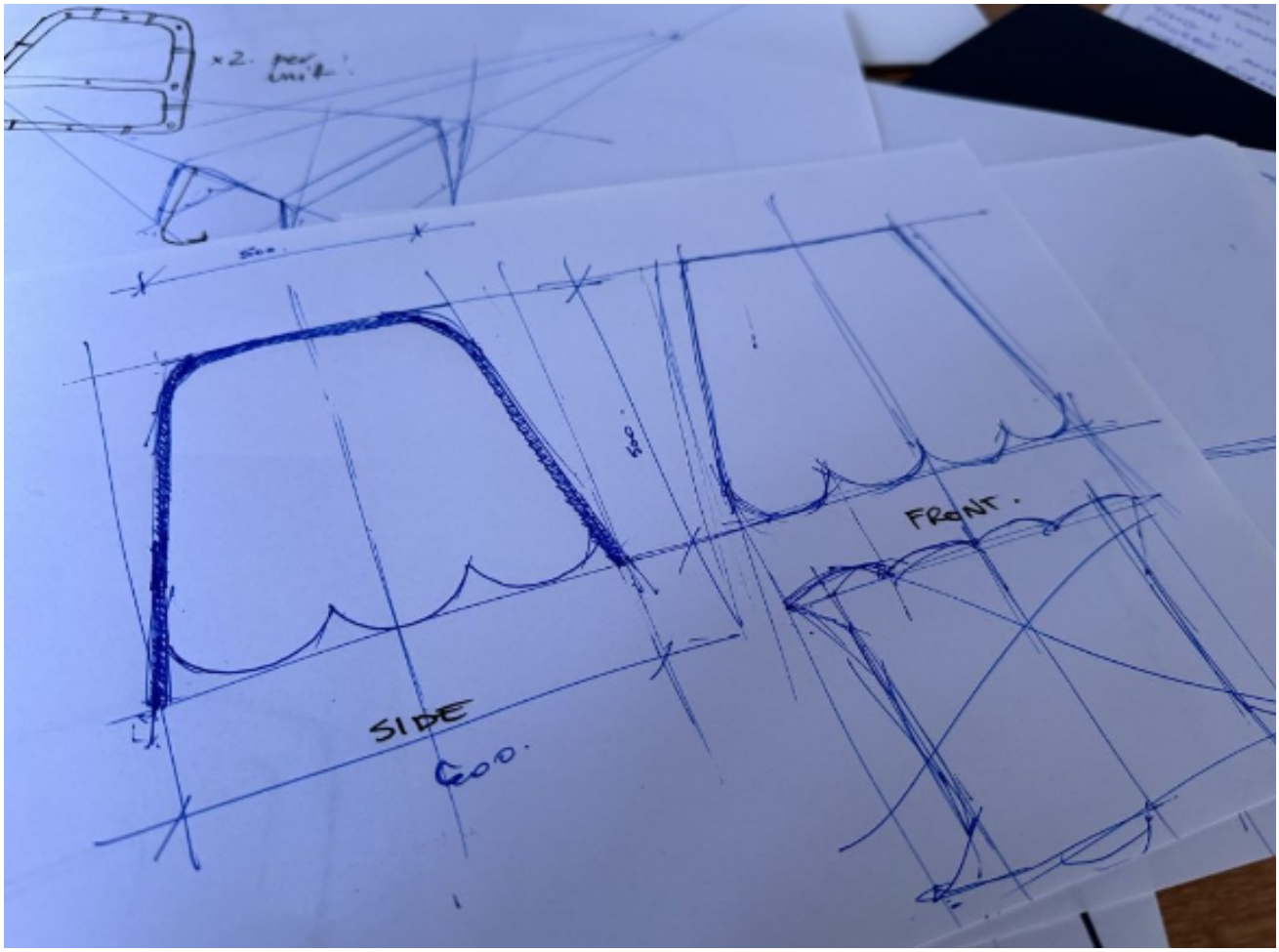
– concept design render by Christina Chen



– concept design renders by Grace Wong

- Phase 3: Concept Development & Feedback Iteration

  The co-designers advanced their initial concepts by incorporating feedback and iterating on the original designs. Each co-designer refined their ideas, addressing issues of modularity, safety, comfort, and interaction while integrating further insights from feedback. The presentations demonstrated adjustments based on feasibility, material choices, and technology integration. Overall, there were notable evolutions and adjustments in design direction, but many core principles from Phase 2 remained.



- Phase 4: Final Presentation & Prototype Testing

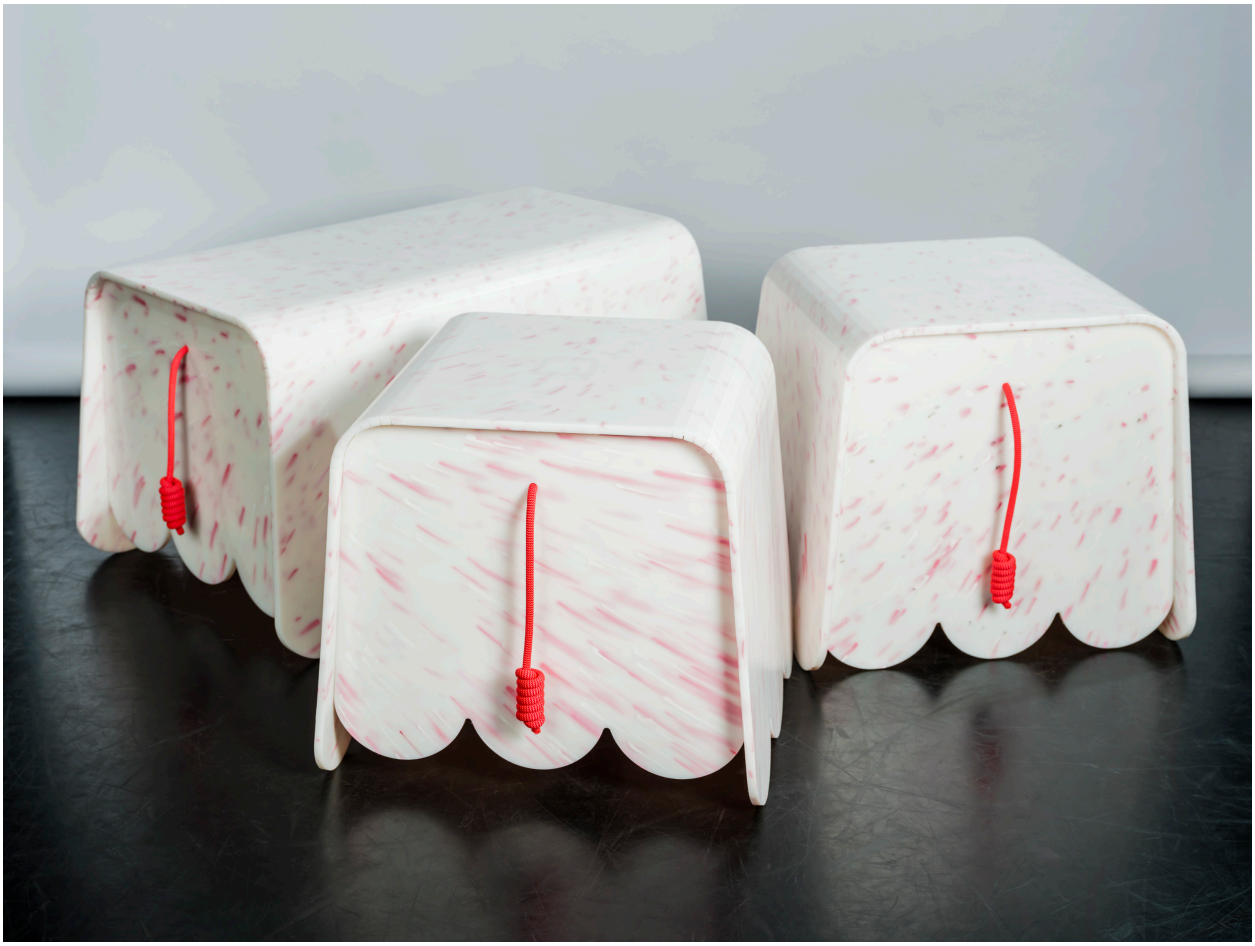
  The co-designers presented their final seating designs, incorporating the feedback and refinements from the previous phases while showcasing their fully developed concepts. These final presentations demonstrated significant progress, with an emphasis on ergonomics, safety, sustainability, and smart technology integration. The designs were geared toward real-world implementation, with considerations for material choices, manufacturing processes, and user interaction. While certain elements from Phases 2 and 3 were retained, this phase focused on refining the solutions into fully functioning prototypes.

- Phase 5: Design Synthesis & Full-Scale Fabrication

  After the completion of Phase 4, which focused on the final presentations of the co-designers, Phase 5 marked a pivotal shift towards the synthesis of the co-designed ideas and the development of the full-scale prototype. This phase involved the UNSW Project Lead synthesizing the insights gathered throughout the co-design process and making critical design decisions to finalise the seating solutions. The result was a full-scale, fabricated installation that differed in some key aspects from the original proposals while still reflecting the overarching goals of the project. The final design emphasised playfulness, flexibility and visibility, and compatibility with the advanced AI sensor technology, tailored to enhance the public space experience for women, girls, and gender diverse people.

Final Product Design

The final product design for Benchmark NSW brought together aesthetic appeal, functionality, technological innovation, sustainability, and a carefully considered fabrication process. The seating was designed to create an inviting public space while meeting the objectives of improving feelings of safety, engagement, and inclusivity for women, girls, and gender diverse people. This section breaks down the key aspects of the final design, including its aesthetics, technology, material selection, and fabrication process.

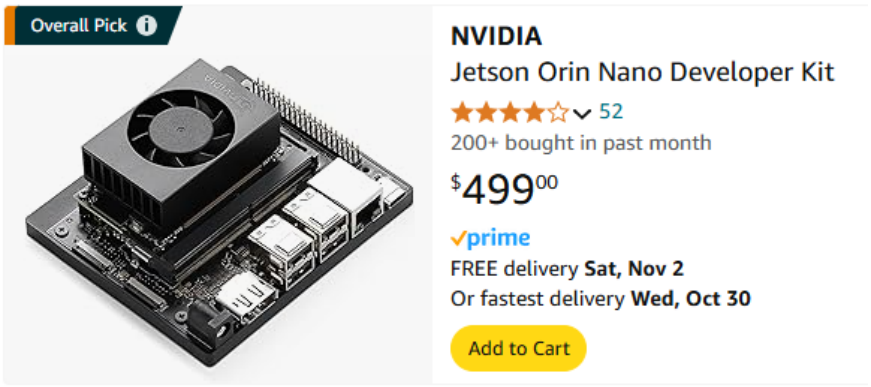


6. Appendix

6.1 Hardware list

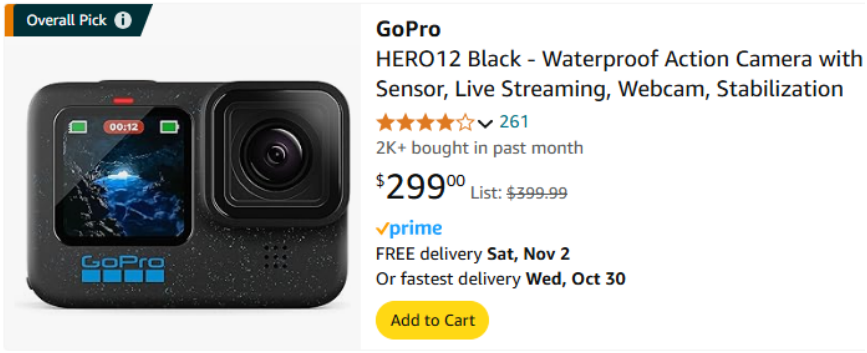
1. Nvidia Jetson

- Usage: On-site computer to run inference from streaming real-time video and anonymize, encrypt all the data.
- Model: NVIDIA Jetson Orin Nano Developer Kit (Ram: 8 GB)
- Price: 499.00 USD



2. GoPro

- Usage: Streaming video footage
- Model: GoPro HERO12 Black
- Price: 399.99 USD

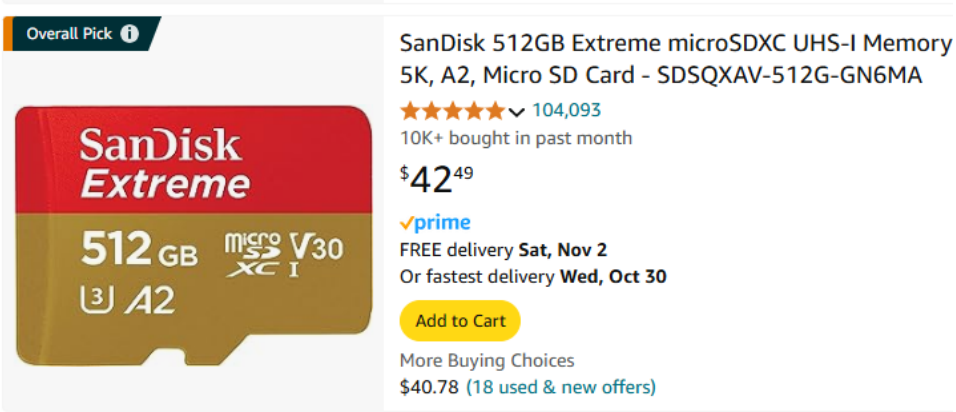


3. SSD

- Usage: Nvidia Jetson hard drive; needs to be installed manually as there is no default drive setup on Jetson.
- Model: 500 GB NVMe M.2 SSD
- Price: 70 USD (varies by manufacturer)

4. SD card

- Usage: Required for the GoPro even if we are not collecting video footage.
- Model: 512GB Micro SD Card
- Price: 70 USD (varies by manufacturer)



5. GoPro Power Adapter

- Usage: Provides stable, constant power for the GoPro; the battery needs to be removed to prevent overheating.
- Model: 100W USB C Charger
- Price: 30 USD (varies by manufacturer)

6. Mobile Hotspot device

- Usage: Provides a stable internet connection for data transfer and remote access to the machine. The Nvidia Jetson has an Ethernet port and one Wi-Fi network card by default, which we use to connect to the GoPro. To ensure a stable connection, this device is recommended to have an Ethernet port. If the local provider does not offer a model with an Ethernet port, you will need to set up a secondary Wi-Fi network card.
- Model: 5G, Ethernet supported

- Price: varies by local provider

7. Waterproof-case for Jetson

- Usage: Provides waterproofing for the Nvidia Jetson, network device, and power adapter. We used a waterproof, ventilated junction box affixed to a pole near the site.
- Model: ABS Electrical Junction Box, Ventilated Design, Cable Grommets, IP65 Waterproof Enclosure, Indoor/Outdoor Use with Mounting Panel (Clear Cover, 11.8”x7.9”x5.1”)
- Price: 40 USD

8. Waterproof-case for GoPro

- Usage: While the GoPro itself is waterproof, if we need to power it 24/7, we need to waterproof the area where the power connector attaches to the GoPro.
- Model: Suptig Waterproof Housing Case
- Price: 20 USD