ECE:4890 — ECE Senior Design
Team 11 Documentation

Joe Spielbauer, Trey Vokoun, Yusuf Halim, Zach Ramsey

Introduction

Robotics and control applications often require wireless communication in challenging environments, whether it be over long distances, through attenuating materials, in locations with high ambient interference, or even with deliberate interference. Such situations are usually handled with explicit data redundancy, channel-switching, or simple retransmission. The aim of this project is to provide implicit error-correction on noisy channels to improve transmission fidelity without sacrificing throughput using an autoencoder-based compression framework.

Our framework consists of APIs for control devices which transmit control signals and receive video data, and remote devices which receive control signals and transmit video data. The control device software provides an API for encoding user control signals into a more expressive representation and an API for reconstructing encoded video data from the remote device. Similarly, the remote device software provides an API to encode video frames into a smaller data footprint and an API for extracting control signals from the control device.

The operation of the APIs is demonstrated on a remote device and control device composed of standard off-the-shelf hardware components. Cross-platform compatibility is demonstrated with the specialized communication system of the prototype Mars rover built by the Robotics at Iowa club.

Project Outcome

Critical Assessment

The project's major components have been designed, prototyped, and tested independently. A user can now control the remote device from the control unit, but only while the remote remains within visual range. This limitation stems from hardware constraints on the control unit: it can display the incoming video stream, yet the frame-rate is so low due to system lag that piloting the remote device beyond line of sight is impractical.

The control unit runs a lightweight GUI on a small screen. It connects to the remote device and clearly renders the video feed, but the slow frame updates hamper real-time control. On the positive side, the controller interface follows a familiar drone-style layout, so it is easy to learn and use.

The remote device is equipped with a 4 K USB-C camera. It captures the video feed at a minimum quality of 480p and streams it to the control unit by sending raw frames as UDP packets. The remote also listens for commands from the control unit translates and executes them reliably.

The entire software stack targets Linux-based operating systems because of library dependencies and performance considerations. The server component is lightweight and runs comfortably on almost any Linux hardware. In contrast, the client responsible for decoding and displaying the video has high CPU demands due to constant frame updates, making it unsuitable for many single-board computers and lower-powered devices.

Project Deviations

We had to deviate from the original plan to use the Esp32-S3 due to insufficient capabilities. We opted for a microprocessor over the microcontroller to boost the compute power and make the control GUI integrate more seamlessly with the demonstration. Additionally, the demo drive base has two power supplies as opposed to one. One for the edge camera, and another for the drive base, its motors, and its antenna. The reason for this deviation was to avoid current draw limitations and noise by the buck converter we had handy. We chose to utilize a usb-c battery pack alongside the li-po to solve this issue.

Additionally, we promised a web-based GUI to view the live web feed from the remote device using our API. However, we were unable to cleanly implement this and decided to implement the video feed via an application window in C++. This was because browsers don’t have a clear straightforward way of communicating with UDP, so we opted for the straightforward and simpler way which was just a C++ client that communicates directly with the server.

Team Member Roles

Zach developed the machine-learning aspect of the project. This included determining a suitable model and training data for the task; developing the model, training pipeline, and inference executable compilation; and developing the C++ API for performing inference with the executable.

Trey did the design and electrical work for the prototypes, including CAD, electrical prototyping, debugging, and assembly of the physical drive-base demo. Trey also contributed to writing the software for the demo and provided the team with computational resources for training the autoencoder model.

Yusuf researched streaming libraries for the remote orange pi. He also made sure that the control device can connect to the remote device seamlessly without any issues and without an IP address since an IP address keeps changing. Yusuf also worked on the web-based GUI stream for the control device to stream the feed coming from the remote device. Yusuf was responsible for testing the Remote and Control device software components to make sure they work and communicate together seamlessly.

Joe designed the infrastructure for the Robotics at Iowa deliverable, including all hardware selection and software to network between the rover and the control station. He also designed 2 custom Yagi-Uda antennas for the control station to communicate with the omni-directional antenna on the rover.

Project Management

We utilized GitHub projects to follow agile scrum methodology and to track and implement the project roadmap, create/assign tasks, as well as manage and track completed and in-progress tasks. We held biweekly meetings as a team and met one-on-one with each other to work or update each other on complete and in-progress tasks. We also have a Microsoft Teams group for general communication regarding meeting plans and sharing updates and documents. A document was also created as a high-level reference of processes and software that needs implemented on the control and remote devices.

Factors in Unmet Goals

We initially planned to use an ESP32-S3 for the remote device but ultimately decided to use an OrangePi Zero-2w instead. This was due to concerns about the ESP32 being powerful enough to run the inference needed to properly run the autoencoder at the desired specifications.

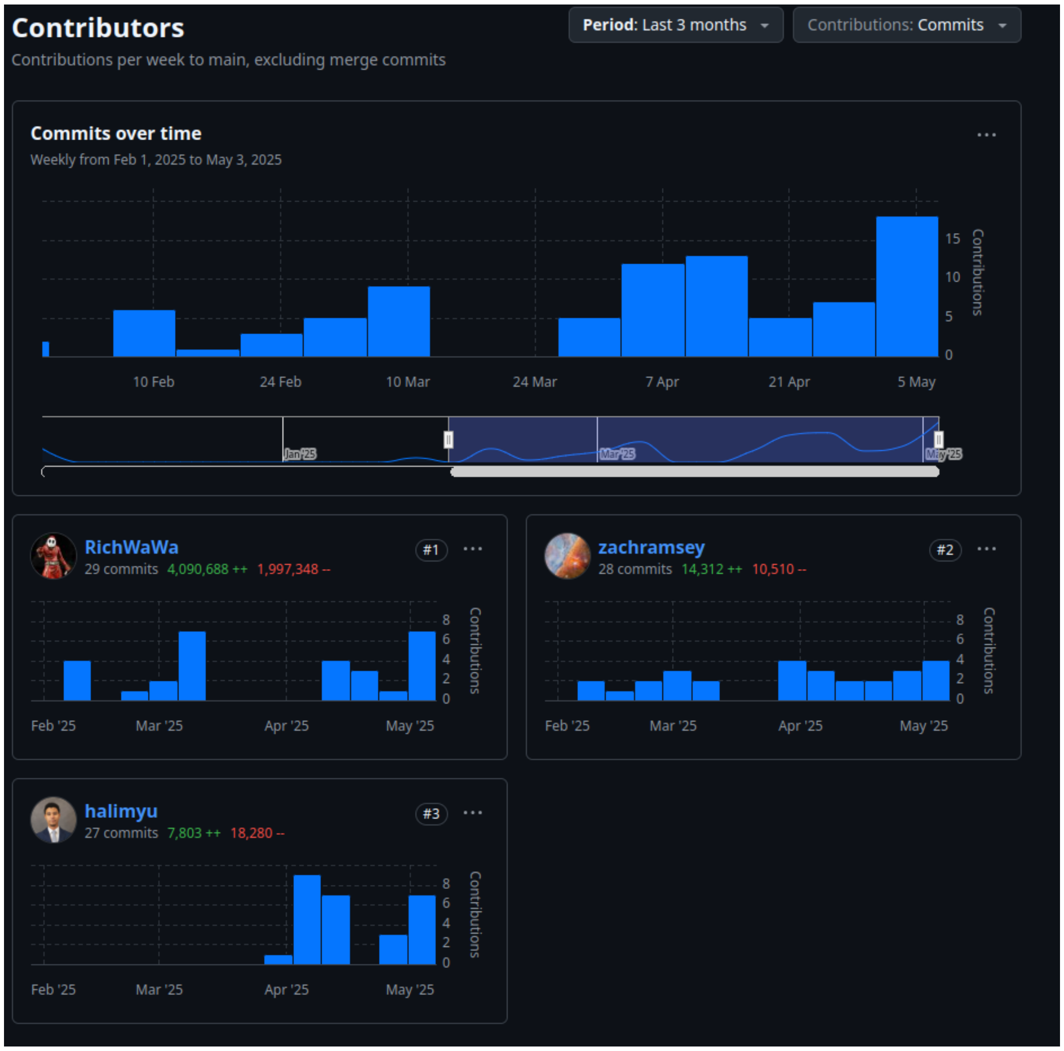
We intended to have a single power source for the demo drive-base, however due to hardware limitations, we were unable to obtain the correct converters. The challenge was going from a Li-Po battery to a clean and regulated 5V source line. Using the LM2596 buck converter caused too much electrical interference for us to adequately filter, so we opted for a WolfWhoop step-down 5v buck converter, which has a lower current rating. This factor required us to use two separate power sources.

We also ran into issues with training the model. The system being used for training ran into storage and permissions issues. These system issues not only set back the training and model optimization process, but also development of the C++ API component for running the models, subsequent debugging, and integration with the target hardware.

Building the APIs for deployment on the edge hardware also proved to be difficult and limited our ability to test its performance as designed. The OrangePi Zeros are not powerful enough to build the models themselves, and this required us to research cross-compilation with CMake.

Regarding video streaming on the control unit with the Orange Pi Zero, we were unable to achieve a seamless feed because of hardware limitations. The client-side streaming code is highly CPU-intensive and needs GPU acceleration for smooth performance. Although the Orange Pi has a GPU, it turned out to be disabled by default, and the code must explicitly enable and use it.

Commit Statistics


Contributors

Contributions per week to main, excluding merge commits

Period: Last 3 months

Contributions: Commits

Commits over time

Weekly from Feb 1, 2025 to May 3, 2025

RichWaWa #1
29 commits 4,090,688 ++ 1,997,348 --

zachramsey #2
28 commits 14,312 ++ 10,510 --

halimyu #3
27 commits 7,803 ++ 18,280 --

** Joe's software contributions were committed directly to the Robotics at Iowa repository, so commit statistics were unavailable.*

Design Documentation

Design Concept

The initial design concept proposed two primary devices: a control unit and a remote unit. The control unit was originally envisioned as a user-centric interface, featuring a modular input mechanism for dispatching commands and a display to show live video feeds from the remote unit. Commands were planned to be encoded into robust signals using inference from a pre-trained autoencoder, then sent via a 2.4GHz transceiver for remote execution.

As we developed, we became aware of the wide applicability of autoencoder compression, so we wanted to create a modular library as a distinct deliverable from the physical device. Furthermore, we met with members of Robotics at Iowa, who were especially interested in video compression on their rover. In all, our project diverged into three deliverables, centered around the compression library. The first is the autoencoder compression library itself, which is applied to both the physical drive-base demonstration (second deliverable) and the video communication system we developed for Robotics at Iowa (third deliverable).

Design Solution

The solution centers around the autoencoder video compression model. We designed this to be a modular library such that it could operate in a wide range of systems and environments. In application, we developed a complete, enclosed three-wheel drive-base with a remote control and on-board screen using this library for the device's video feed. The second application was the communication system for Robotics at Iowa on-board their prototype Mars rover.

The autoencoder model is an implementation of the Scaled Hyperprior model. This model includes not only the standard encoder and decoder, but also a secondary hyper-encoder and decoder which is analogous to encoding side information about the primary encoding. At the bottleneck of both autoencoder components, the natively floating-point data is quantized into integer representations. [9] The quantized information is packaged as signed 8-bit integers. Signed 8-bit integers are the underlying datatype of a "char" in the C language, which maps to the data type expected by digital wireless network protocols. Thus, the encoder output and decoder input can seamlessly integrate with standard transceiver setups. Likewise, the encoder input and decoder output are packaged as unsigned 8-bit integers, the data type for a standard byte. This maps to an intuitive representation of raw image data where each RGB pixel is represented by 3 bytes for each color channel.

Using the ExecuTorch framework for PyTorch, the autoencoder model can be compiled ahead-of-time into instructions for the pure C/C++ Executorch runtime to execute. Use of such an executable with the runtime is exposed as an API by our custom C++ library such that highly optimized inference can be performed on the model with minimal effort by the application.

The two primary devices on the drive-base are built on OrangePi Zero 2W microprocessors. DietPi with LXDE is installed on the control device to display the GUI; DietPi is installed on the remote device to minimize overhead.

The Server design is built to send camera data in a minimum of 480p quality and in Raw BGR 8-bit format over UDP-sockets to the client. The design has no user display at all sense it will be mounted to the drive base hardware. The client software was designed to take in the data sent from the server and display it in an application window coded in C++ in the same data format it was received. The C++ window has the option to zoom in and take snapshots to the control device.

The Robotics at Iowa rover required reliable communication to at least 1km, even without direct line of sight. For better communication between devices, a 2x2 MIMO Yagi antenna was designed for the control station in place of an omni-directional antenna for vastly improved distance and signal strength. This was applied to a Ubiquiti Rocket M2 modem with 28dBm output power. The high output power with directed antenna gain means a better signal-to-noise ratio (SNR) and therefore better data rates.

The hardware deliverable was designed to showcase the flexibility of our library and ease of implementation into edge solutions. Our Kiwi-Drive robot was chosen due to its availability, and accessibility to higher powered batteries; e.g., a 25C Li-Po. This has more than enough capacity to power all of our electronics. Additionally, a high output USB-C battery pack was chosen to power the remote device.

The drive base uses a teensy4.0 microcontroller, 2.4ghz receiver, Pololu HV motor drivers, and TurnABot DC motors. These were all readily available components that the team had on hand, and they were small enough to fit in our compact package. Our control solution was to use a standard 2.5ghz radio with an OrangePi Zero and a small usb-c powered screen mounted on it. Additionally, we use a dual-port USB-A battery pack to power the OrangePi and the display on the control handheld.

Design Suitability Discussion

The control device assembly is sufficient for demonstration with all necessary components. The remote device is a complete edge solution, but due to the drive-base having been originally designed for a microcontroller as opposed to the higher power drawing microprocessor, the mechanical design is not optimal.

Simulated testing in a python environment show that the model is capable of compressing image frames down substantially and subsequently reconstruct the images with high fidelity, making transmission far more data-efficient and suitable for real-world scenarios. The use of ExecuTorch for more performant inferencing makes the use of the pre-trained model viable for use in edge solutions.

Solutions and Tradeoffs Analysis

The OrangePi Zero has significantly more processing power than the Esp32-S3, however this comes at the cost of power. The esp32-S3 draws a peak of 250mA, as opposed to the OrangePi Zero which draws a whopping 2 Amps.

There are a few well-established machine learning-based solutions for denoising corrupted signals. There are standard variational autoencoders which have been well studied for the purpose and novel solutions like Vector Quantized Generative Adversarial Network and Scale Hyperprior. Many other novel methods are based on the Scale Hyperprior as it has been shown to be very effective at reconstruction. Due to its prevalence and extensive study, the Scale Hyperprior was chosen. [7, 9]

The original design of the scale Hyperprior specifies that lossless entropy coders are situated at the bottlenecks of the model to further compress the quantized information. Particularly, a range asymmetric numeral system (rANS) is used in the CompressAI implementation of the Scale Hyperprior model. [8] While adding additional compression would further reduce the size of the latent space for transmission, further analysis found that the rANS coder, and other entropy-based coders by extension, are numerically unstable to even small errors. This instability would have effectively nullified the error-correcting properties of the autoencoder. Therefore, it was decided not to further compress the latent data at the bottlenecks to preserve stable image reconstruction capabilities.

On the rover communication system, several solutions were considered regarding modem and antenna options. The rover already possessed generic omni-directional antennas with modems for sending basic movement instructions, but the data-rate was nowhere near adequate for live video. 2x2 MIMO modems were chosen for their ability to differentially isolate noise. On the rover, it was decided that omni-directional would still be the best choice since controlling a directional antenna would require additional motors and power draw, which it could not afford due to the rover's weight restrictions and efficiency requirements. However, the control station has no such requirements, so a directional antenna is ideal because communication only exists between two devices. A Yagi-Uda antenna design was chosen for its simple construction, lightweight design, modularity (additional directors are easy to add), and high directivity.

Constraints

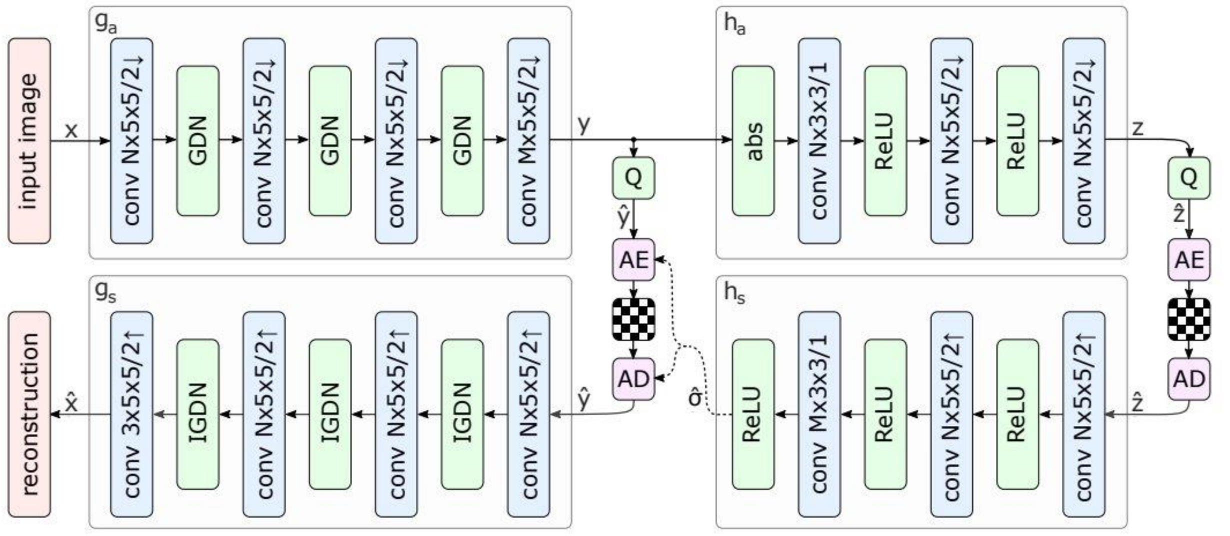
1. Component construction shall be small, robust, and modular.
2. Device power shall be long-lasting and easy to restore.
3. Devices shall have sufficient computational resources.
4. Communication latency shall be low and reliable.
5. Video communication shall have a resolution of at least 480p at 30fps.
6. Video and control signals shall see no significant loss in video feed to at least 1km.
7. Webpage and video streaming should be lag free and have minimal to low data loss.

Standards

1. 2.4 GHz Communication — Widely used in radio-controlled applications for wireless communication.
2. RF Connectors – must be 50Ω coax with SMA connectors
3. CMake — Compiler-independent tool for building C++ applications.
4. C++ — Compiled, high-level, general-purpose programming language extensively used for embedded applications.
5. Python — Interpreted, high-level, general-purpose programming language, extensively used for machine learning applications.
6. PyTorch — Python library for building deep learning pipelines that calls upon algorithms implemented in C++.
7. ExecuTorch — PyTorch framework that performs ahead-of-time compilation of PyTorch models into an executable format, which is then run with the ExecuTorch runtime, written in C++, for performing inference on underpowered edge devices like microcontrollers, microprocessors, and mobile device.

Architecture

Autoencoder Overview

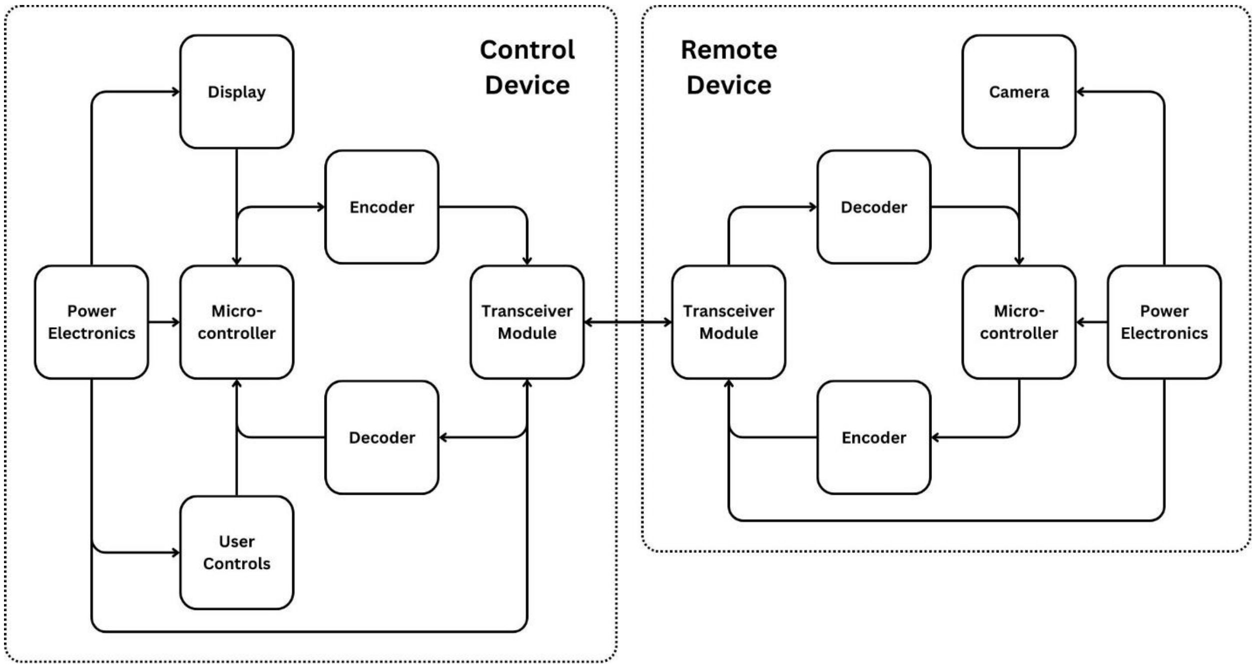


J. Ballé, D. Minnen, S. Singh, S. J. Hwang, and N. Johnston, "Variational image compression with a scale hyperprior," May 01, 2018

The blocks labeled encoder and decoder in the system architecture correspond to the upper two and lower two blocks in the autoencoder model architecture above labeled g_a/h_a and g_s/h_s, respectively. The left two blocks are the standard image autoencoder and the right two are the hyper autoencoder, which encodes side information about the latent image. Q indicates quantization of the floating-point values; AE and AD indicate arithmetic encoder and decoder, referring to an arbitrary choice of lossless encoding that may be situated at the autoencoder bottlenecks for further compression. The checkered patterns represent the compressed data itself, which the transceivers handle. [9]

Drive-base Demo

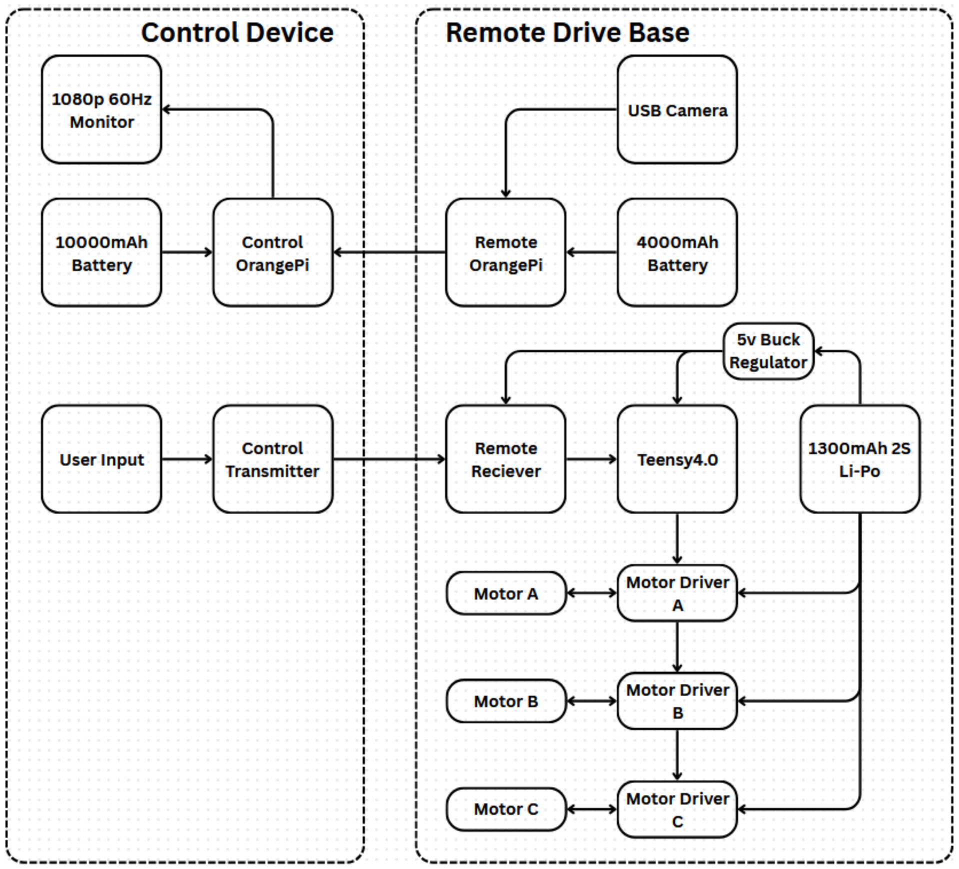
System Overview



The design consists of two primary devices: the control device and the remote device. The control device facilitates user operation, featuring a user interface to dispatch commands and a display to show live video from the remote device. The remote device is designed for mobility and integration, featuring an interface for a camera to capture video and controlled hardware. The captured video is encoded into a robust and data-efficient representation by inferencing a pre-trained autoencoder. These signals are transmitted back to the control device where they are decoded and displayed to the operator.

Both devices rely on OrangePi Zero 2W microprocessors to handle computation, including video processing, signal encoding/decoding, and hardware control. Each device has power electronics to support its components and ensure a runtime of at least three hours. Communication between the devices is enabled by modular 2.4GHz transceivers, allowing efficient and reliable data exchange over significant distances.

Hardware Overview



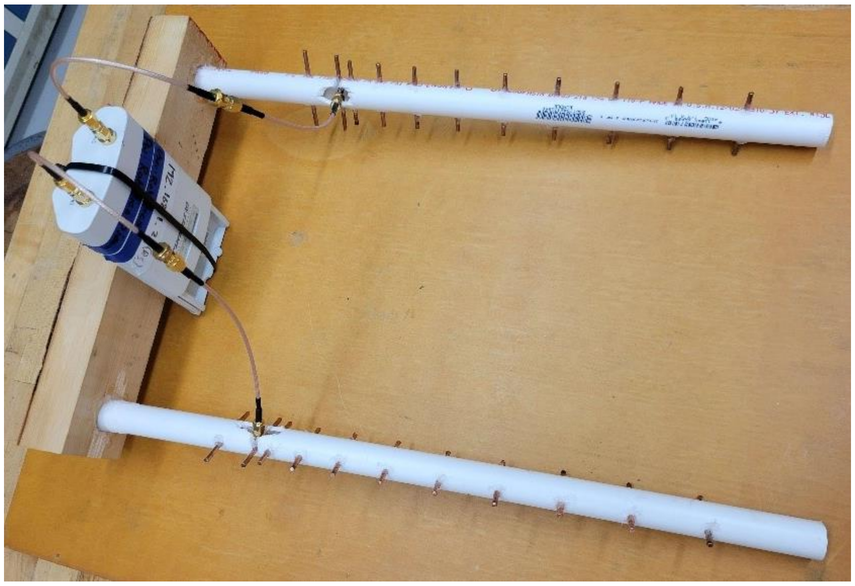
The control device is what the user uses to interface with the remote drive base. This handheld was designed to accommodate all the components needed to run the SoC as well as control the drive base. We utilized a 10000mAh battery pack that has two usb-A ports to power the pi as well as the screen. Our tests showed that we got around 4 hours of battery life with the display as max brightness. This could be improved by lowering the brightness and the refresh rate of the display. However, the display is bright enough to be viewed outdoors. The transmitter we used is a FlySky 2.4Ghz transmitter. This transmitter offers exceptional features and range for its price. While we were unable to transmit all of our control signals via the encoder, the transmitter and receiver combination we used work exceptionally for the demo.

The remote drive base was more complex to set up and design. This demo unit consisted of a Teensy4.0 at its core, which is responsible to reading the input signals from the 2.4Ghz receiver, running those inputs through our drive algorithm, then sending the corresponding drive PWM's to the motor drivers. The receiver outputs an X, Y, and rotate signal as a 50Hz PWM pulse. This pulse has a minimum duty cycle of 5% and a max of 10%. Using this, we are able to read these pulses and determine how the transmitter stick is oriented. The Teensy4.0 reads these inputs, runs them through the drive algorithm, then outputs how fast and in what direction each motor needs to spin to make the drive base move that direction. Our drive base design is formally known as a “Kiwi Drive” and its drive algorithm utilizes vectors to determine the motor output. For the motor drivers, we opted to use Pololu DRV8874. These motor drivers are controlled via a direction pin and a PWM pin for speed. This motor driver offers a continuous current rating of 2.1A at a peak voltage of 37V. These larger capacity drivers were required for the motors that we chose. The motors we chose are the Turnabot N20 motors, which have a peak current draw of 2 amps. This component selection was chosen mostly because this is what we had available on hand.

The hardware construction is a mix of rigid 3D printed parts, and improvised cardboard. The frame of the demo drive base is a custom design, 3D printed for our kiwi drive-base, and the electronics are zip-tied to the chassis’ lightning holes. Additionally, the drive base uses custom designed omni wheels which are more than 50% lighter than other available options. The axles for these wheels are simply steel paperclips that were cut to length. Aluminum paper clips were initially used but found to deform too much over time.

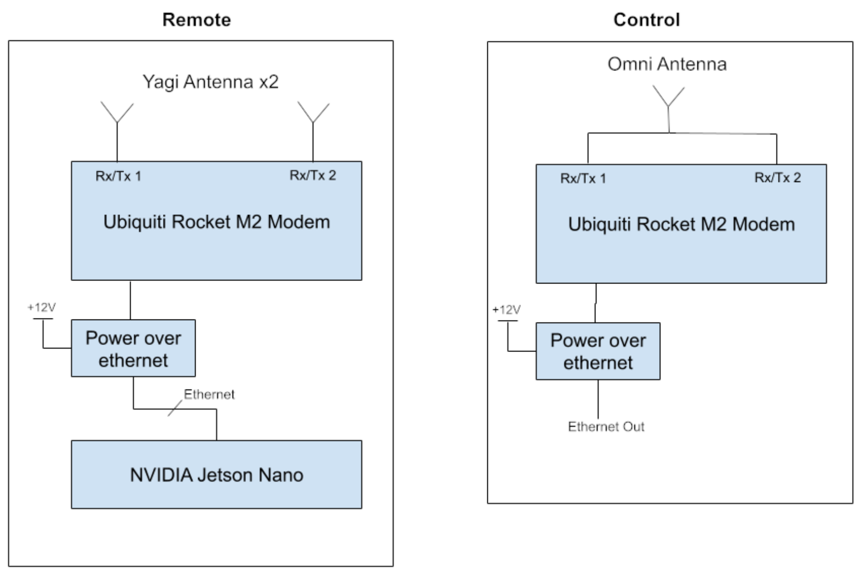
The enclosure for the remote camera was designed to be a versatile enclosure for any use case. The enclosure is fully 3d printed and securely mounts the camera, Pi, and internal wiring. Additionally, it features airflow holes to help prevent overheating during extended use. Since the remote device is in such a small package, we had to custom make a USB-C cable for the camera to connect it to the pi. We used a 56kΩ resistor wired as a “sink” to indicate that the connected device needs to “sink” power from the USB-C port of the OrangePi.

Rover Deliverable



Dual antenna with modem

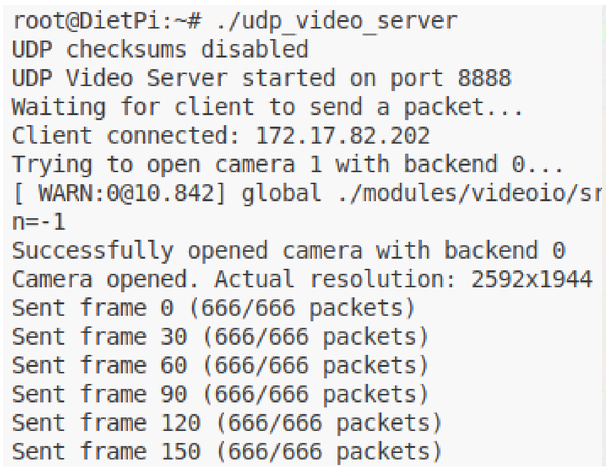
The Rocket M2 is a 2x2 MIMO 2.4GHz modem with 3 non-overlapping 20MHz bandwidth channels available. With 2x2 MIMO, both ports act as transceivers simultaneously, combining the received signals for more reliability. The modem has automatic channel switching depending on which has the least interference and strongest signal. The Rocket M2 transmits at 28dBm, and the Yagi antennas have a broad gain of 10dBi across the bandwidth.



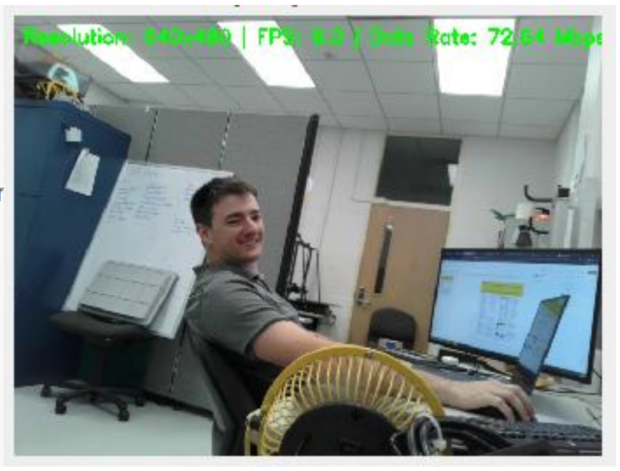
The Omni antenna is a Ubiquiti airMAX Omni AMO_2G13, compatible with the Rocket M2 modem. On both control and remote, the modem is powered by power-over-ethernet (POE) injectors. The rover (remote device) operates on an NVIDIA Jetson Nano for all processing, while the control station is designed to work with any laptop running Linux.

UI/UX

User Interface



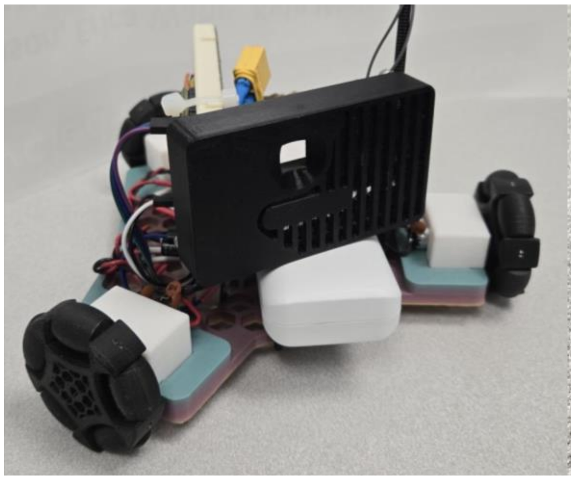
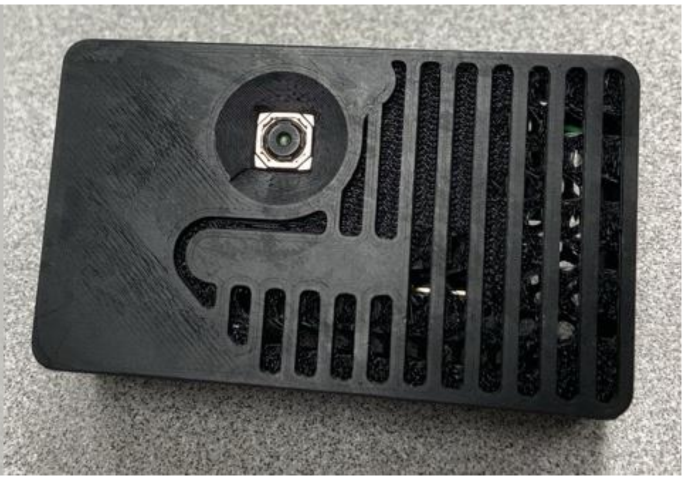
```
root@DietPi:~# ./udp_video_server
UDP checksums disabled
UDP Video Server started on port 8888
Waiting for client to send a packet...
Client connected: 172.17.82.202
Trying to open camera 1 with backend 0...
[ WARN:0@10.842] global ./modules/videoio/sr
n=-1
Successfully opened camera with backend 0
Camera opened. Actual resolution: 2592x1944
Sent frame 0 (666/666 packets)
Sent frame 30 (666/666 packets)
Sent frame 60 (666/666 packets)
Sent frame 90 (666/666 packets)
Sent frame 120 (666/666 packets)
Sent frame 150 (666/666 packets)
```


Resolution: 640x480 | FPS: [illegible] | Data Rate: 72.64 Mbps

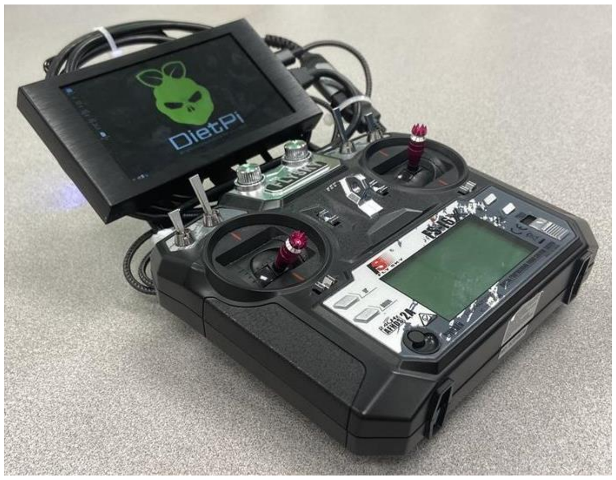
Terminal output from the remote device when the streaming server starts (left) and the video stream received by the client (right).

On the right, the client display shows the video stream at a minimum resolution of 480p. The image quality is good enough to discern what is happening in the scene. On the left is the server's terminal output, confirming successfully camera initialization, client connection was established, and packets are being sent. This server output is not shown to end users; it is intended solely for debugging and logging.

User Experience

The drive-base demonstration unit is shown above. The image on the right is the remote device itself consisting of the OrangePi and camera within a custom 3D-printed enclosure. The image on the left shows the device mounted to a robotic drive-base with power attached via a port accessible through the enclosure.



The control device for the drive-base demonstration unit is shown to the right. The OrangePi is contained in a 3D-printed enclosure behind the top display. The display is also mounted in a 3D-printed housing. The OrangePi and display are connected for data communication and each to power through externally accessible ports through their respective housings.

UI/UX research and testing

Our original plan was to display the video on a webpage via HTTP using an MJPEG-streaming library. We successfully tested this: the remote device captured frames with a pre-built MJPEG library cloned from GitHub, sent them through a port, and the client rendered them in a browser interface.

After further tests and discussions with Zach about integrating the machine-learning model (ExecuTorch .pte files), we realized this approach would not suffice. The model requires the stream to arrive over UDP sockets, with checksums disabled, and in raw 8-bit RGB format. UDP's connectionless, no-retransmission nature prevents the model from stalling on lost packets, while disabling checksums ensures every frame, even if partially corrupted, reaches the model for inference.

Accordingly, we implemented a C++ server-client pair: the server runs on the Orange Pi remote device, captures camera frames, and sends them as raw RGB over UDP; the client receives the packets and hands them to the machine-learning pipeline while also displaying the video.

We benchmarked both the original webpage approach and the new UDP solution. For the demo, the server ran on the Orange Pi remote and the client on a laptop (the control-side Orange Pi lacked a GUI OS at the time). The UDP stream delivered noticeably lower latency and smoother playback. However, when we later tested the client on the control-side Orange Pi, its limited hardware struggled to refresh the display fast enough for a completely lag-free experience.

Maintenance

Our main deliverable is a set of APIs that enable users to optimize video feeds. Assuming an existing model executable file, building the APIs from source only requires Cmake, Executorch, and dependencies automatically installed by Executorch. A pre-compiled instance of the APIs works out-of-the-box in C++ applications. The model training pipeline requires PyTorch, TorchEval, CompressAI, and the automatically installed dependencies of these Python packages. The setup process is described in the technical documentation and GitHub repository. Currently, such a setup would be required for installation and upon software updates.

On both physical deliverables, batteries need to be charged occasionally. Additionally, as hardware ages, components may need to be replaced. This is especially true of the drivebase assembly, which includes several discrete components that may break easily such as capacitors and motors. On the robotics system, SMA connectors for the antennas should be maintained by keeping them clean and using a torque wrench to apply no more than 0.6Nm to the joint. Additionally, care should be taken not to damage the antennas, which are somewhat delicate as the elements are made of copper. If these elements are shifted or bent even by a few millimeters, it can drastically affect the performance of the antennas.

Future Work

The model training pipeline is currently narrow in its extensibility. This first iteration of the software is designed to train a specific model, and it would not be immediately clear to an end user how to extend it to a unique situation. More organized maintenance of data associated with past training runs, automatic – or at least simpler – purging of failed runs, and automatic hyper-parameter tuning would substantially improve usability. A more streamlined and hands-off path from trained model to edge deployment would make the entire software package much more cohesive and appealing to the end user. The most substantial improvement for useability would be to package and publish the C++ libraries to common Linux package manager repositories to minimize the complexity of installation and updates for the user.

For the rover communication system, the most significant bottleneck is still the RF signal strength, although the custom Yagi antenna vastly improved this area. The most cost-effective improvement would be to simply add an impedance matching network. After testing the antenna with a network analyzer, the input impedance deviated somewhat from the ANSYS HFSS simulation, so the actual antenna was not matched to the 50Ω modem ports. This matching can be most easily accomplished with a simple pi network.

For Server/Client software, If given more time we could have the client implemented to utilize the Orange Pi Zero GPU so run and update streamed data frames seamlessly without lagging and lost packets.

Test Report

SR 1 User shall be able to operate the remote device.

SR 1.1 Input to the control device by the user shall directly govern actions of the remote device.

SR 1.2 Video captured by the remote device shall be displayed to the user on the control device such that the remote device may be operated beyond visual range.

SR 2 The control device shall receive commands and transmit video.

SR 2.1 The control device shall capture control signals, encode controls into latent signals, and transmit the latent signals to the remote device.

SR 2.2 The control device shall receive latent video data from the remote device, decode the latent video data, and output video to the display.

SR 2.3 Received video shall have a resolution of at least 480p at 30fps.

SR 2.4 99th percentile of latency between video capture and video display shall not exceed 50ms.

SR 2.5 Video signal shall see no significant loss to at least 1km with the long-distance test.

SR 3 The remote device shall transmit commands and receive video.

SR 3.1 The remote device shall have a camera capable of capturing video at a quality sufficient for operation beyond visual range.

SR 3.2 The remote device shall capture video, encode the video data, and transmit the latent video data to the control device.

SR 3.3 The remote device shall receive latent control signals, decode the latent control signals, and carry out actions corresponding to control signals

SR 3.4 Control signals shall see no significant loss to at least 1km with the long-distance test

SR 4 The system shall be modular in design.

SR 4.1 The system software shall be structured in a way that makes it easy to implement on a range of processors.

SR 4.2 The system software shall be structured such that it can be used on a Linux or Windows-based PC.

SR 4.3 The system software shall be designed such that it is compatible with a range of peripheral devices. Ex, controllers, keyboards, cameras, motor drivers, and antennas.

Requirements Traceability Matrix

		Test Case ID									
		TC01	TC02	TC03	TC04	TC05	TC06	TC07	TC08	TC09	TC10
System Requirement	SR 1.1	x							x	x	x
	SR 1.2		x					x		x	x
	SR 2.1	x							x		
	SR 2.2		x		x			x			
	SR 2.3		x		x						x
	SR 2.4		x							x	x
	SR 2.5										x
	SR 3.1		x					x			

	SR 3.2		x	x					x		
	SR 3.3	x						x			
	SR 3.4										xfuture
	SR 4.1	x	x			x	x	x	x		
	SR 4.2						x				
	SR 4.3	x	x			x		x	x		

Tests Performed

Test Case ID	Test Description	Success / Fail	Explanation
TC01	The RC drive base is governed by input from the control device.	Pass	The Control Device can successfully control the remote device seamlessly
TC02	Video captured by the RC drive base is displayed on the controller.	Fail	Control device had some hardware constrained where it couldn't display the sent packets quickly and efficiently without a lot of lagging and lost packets
TC03	Encoder can represent image data in a smaller latent space.	Pass	Raw data to latent data compression ratios: ~4:1 (default model), ~8:1 (small model), ~12:1 (very small model).

TC04	Decoder can reconstruct images from latent space with high fidelity.	Pass	Reconstruction has a PSNR of 29.4 in the default model & 25.8 in the small model.
TC05	Encoder/decoder C++ libraries are cross-compatible with diverse hardware.	Pass	Executorch compiler backend and CMake project builder can be configured to operate on different hardware with the same API and model.
TC06	The system software is easy to implement on a range of processors.	Fail	Client software runs a little slow on SBC as it requires a little more processing and GPU
TC07	Robotics control station receives video transmitted by the rover	Pass	Successfully streamed video at roughly 10 feet apart, and displayed on the control station.
TC08	Robotics rover receives commands transmitted by the control station	Pass	Rover successfully received commands, which were printed to the rover console.
TC09	Robotics control station continues to receive video and send instructions to rover without direct line of sight (small obstructions: trees, cars, around corners)	Pass	Minimal loss in video with small obstructions, Video was lost behind large buildings but instructions remained albeit with significant latency.
TC10	Robotics control station continues to receive video (at least 480p 30fps, with 99th percentile of latency within 50ms) and send instructions to rover with direct line of sight at a distance greater than 1 km.	Pass	Received video was 1080p, 30fps with virtually no latency at 1km without obstructions.

Appendices

1. Rocket M2 Modem Datasheet
2. airMAX Antenna datasheet
3. Avahi Deamon DNS
4. MJPEG streamer documentation
5. MJPEG streamer github library
6. OpenCV
7. M. Naseri, P. Ashtari, M. Seif, D. P. Eli, P. H. Vincent, and A. Shahid, “Deep Learning-Based Image Compression for Wireless Communications: Impacts on Reliability,Throughput, and Latency,” arXiv.org, 2024. https://arxiv.org/abs/2411.10650v1.
8. J. Bégaint, F. Racapé, S. Feltman, and A. Pushparaja, “CompressAI: a PyTorch library and evaluation platform for end-to-end compression research,” Nov. 05, 2020, arXiv: arXiv:2011.03029. doi: 10.48550/arXiv.2011.03029.
9. *J. Ballé, D. Minnen, S. Singh, S. J. Hwang, and N. Johnston, “Variational image compression with a scale hyperprior,” May 01, 2018*