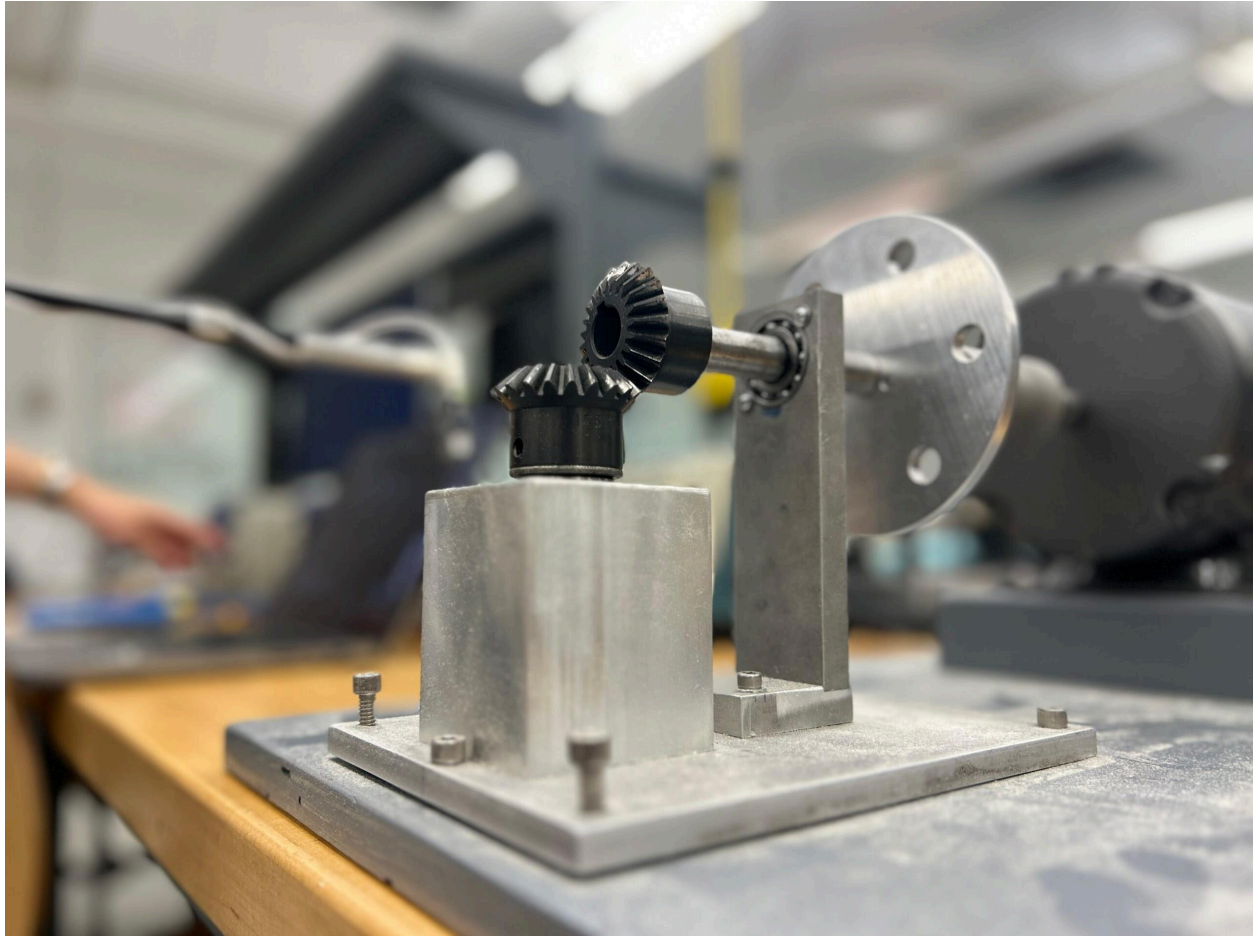


Machine Health Sensor and Mechanical Test Bench

Final Report



Oliver Annon, Emma Wang, Raffael Venterea

PROBLEM DESCRIPTION

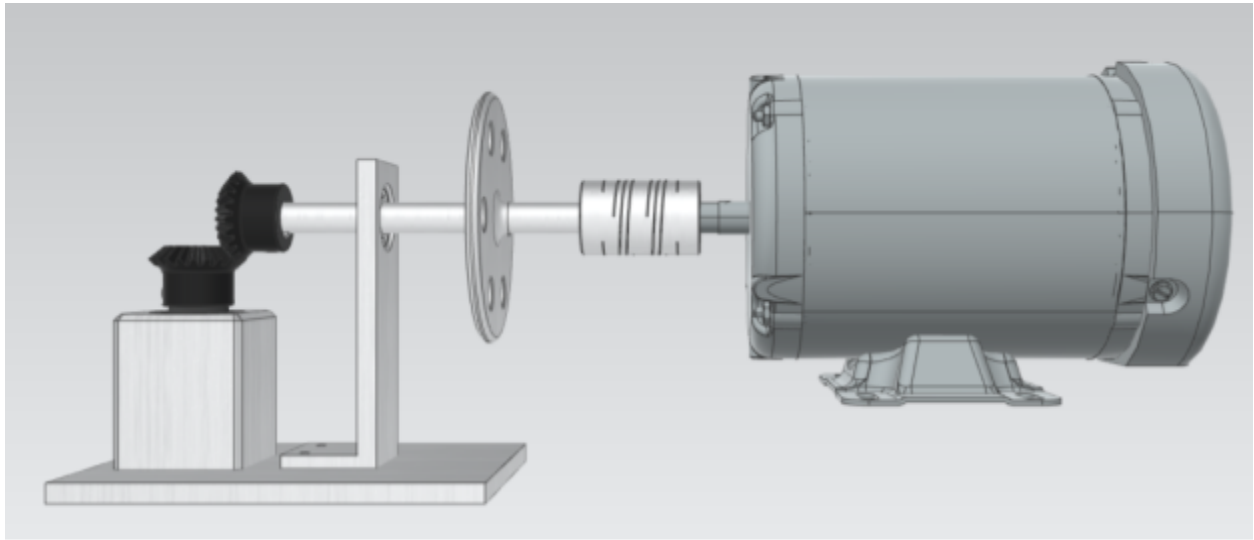
The time required to fix broken technology in a commercial or industrial setting is called *Unscheduled Downtime*. For any company dealing with machinery, whether it's production lines or construction, unscheduled downtime is the most damaging expense possible. Not only does it cost money to repair, but it also bleeds any potential profit from unproduced products or site progress, and it causes delivery delays. Another cost of failure is the loss of resources and exposure of workers to dangerous chemicals. Especially regarding machinery that handles fluids, damaged or failing machinery can result in a release of chemicals that may harm or kill workers.

The target of this project was to develop a remote sensor that can detect mechanical failures when they occur and use acoustic cues to predict component lifespans. This would inform technicians which repairs are imperative and reduce unnecessary downtime. Current solutions are expensive and typically reserved for high-end machinery. Developing a budget-friendly option creates an opportunity to deploy this technology to a wide range of systems. To achieve this, a custom mechanical test bench was designed, and an Arduino Nano was loaded with a machine-learning model trained on collected acoustic data to detect faults.

IMPACT

The most common solution to this problem is scheduled downtime. This means planning out gaps in operation to inspect and repair machinery. This allows for predictable changes in throughput or capability, which can be communicated to customers. This does not change the fact that it is still downtime, which cuts into profits and capability. Full and extensive inspections can vary from hours to days, depending on the context. Then there is the added time required to complete necessary repairs, but there is also a chance that inspections turn up no faults.

CONSIDERATIONS, CONSTRAINTS, AND PROCESS FOR THE DESIGN OF THE MECHANICAL TESTBENCH



The final assembly, modeled in Siemens NX

Considerations

Testbench

When drafting the testbench design, the goal was to create a repairable failure environment. This meant that components could be intentionally damaged in a known way and then returned to regular operation after testing concluded. The prior research by Ferraz Júnior et al. utilized a compact design consisting of an electric motor, shaft coupling, steel disc, and a mechanical transmission.[1] The final design for our project closely mirrors the original, with a few key differences due to material costs and accessibility.

Four main failure points were selected to emulate common faults in real-world mechanical environments. These were: axle misalignment, rotational unbalance, multiple gear transmission issues, and ball bearing failure. We designed 3 main components: The balanced wheel, bearing housing, and jackplate. Then, pre-fabricated parts were attached to complete the system.

Repeatable trials are critical for conclusive experimental results, so each of these failure points needed to be controllable and repairable. This was achieved for each through the following methods:

- Axle misalignment: Shaft coupling with 2° of freedom and a baseplate with jack screws.
- Rotational unbalance: Balanced wheel with off-center mounting points for weight
- Gear Transmission issues: Miter junction with a swappable set of healthy and damaged gears.
- Bearing Failure: Housing containing two slip-fit bearings, one of which can be removed and swapped with a damaged bearing.

The details and design of each of these is covered in the process section. The list of purchased parts is available in Appendix C.

Another important factor was the operator's safety. The testbench needed to be secure enough that parts would not fly off and strike anyone observing. There are plans to create a plexiglass shell to cover the assembly while still allowing for visual observation, but we were unable to acquire the materials in time. This shell would also provide the operator with physical protection from spinning and grinding components. Long hair can easily be caught on the rotating axle, and the gear transmission is more than capable of chewing up fingers with irreversible consequences. Our team did not encounter any of these problems during testing, but they are critical to remember during operation.

The future use of this device and its upgradability were slight yet considered aspects at the end. The design is modular enough that swapping out sections is feasible, if not mildly labor-intensive. The most notable change that may be made in the future, if the budget allows, is the transition from the current straight-cut gear transmission to a helical gear transmission. Slightly different mounting distances and angles would mean adjustments to the drive axle position and bearing housing height, but is entirely achievable.

Constraints

Mechanical Constraints

The primary constraint for this project was the cost of mechanical parts. In favor of the project, an old Fort Wayne electric motor was sourced from the University's inventory, which saved in excess of \$150. A drawback of not purchasing a new motor was the lack of feature selection. This meant that the bench needed to be built around the motor's characteristics and dimensions. The ¼ hp rating was the correct specification, but the 1700 RPM output meant that any downstream rotational component needed to handle at least 2000 RPM, close to the recommended 20% maximum overhead. Another issue with the motor's fixed speed was the startup impulse. Upon activation, the generated torque would cause the motor to lurch, potentially damaging the assembly if mounted improperly.

The solution to this problem was the acquisition of a Baldor M3452 3-phase motor. Rated at ¾ hp and 1725 max RPM, it sat close enough to the operating range parts were selected for. Additionally, the physical dimensions were sufficiently similar to the original motor that only a minor adjustment to the mount was needed. It required a variable-frequency driver (VFD) to control the RPM and ensure accurate, repeatable testing. This also fixed the startup impulse issue and ensured safe operation of the testbench without exceeding maximum ratings.

After the VFD, the gears were the most expensive components. Initially, the bench was designed to use a worm gear and worm wheel, but a brief examination of the available parts on McMaster-Carr showed that the financial restraints would not meet the physical requirements. The second design was going to use helical-cut miter gears, which offer a quieter overall noise profile; however, they also cost almost 3x as much per gear compared to straight-cut gears (Figure 1). In addition, the helical gears need to be purchased in matching pairs. This would have required buying a total of 6 gears in order to achieve the testing goals.

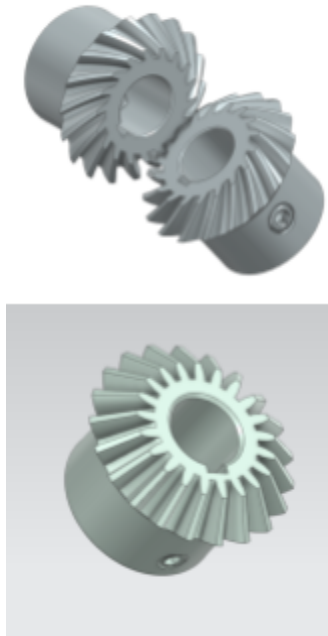
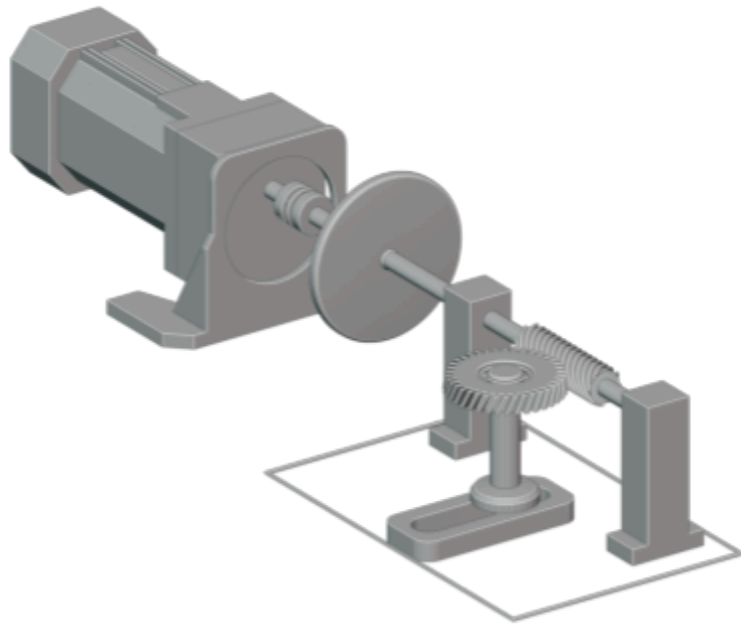


Fig. 1

The second constraint was size. The unit needed to be portable, so it had to be relatively compact and lightweight. To address this, wood was chosen for the baseplate to reduce weight. All other structural components were milled from aluminum.

Process

The rough draft of the testbench was created in Blender 3D 4.1 and provided a sketch of the desired components and stages. This draft was not to scale and only had rough approximations of part shapes. The very first draft still shows the original worm-gear-to-worm-wheel transmission; in the first revision, it is replaced with the miter gears.



The design concept drafted in Blender 3D

The final modeling of components was completed in Siemens NX10. The process started with taking measurements of the initially available motor and creating a placeholder model with the mounting points and drive shaft in the correct locations. When the new motor was acquired, it was expected that some adjustments would be needed to match the new size.

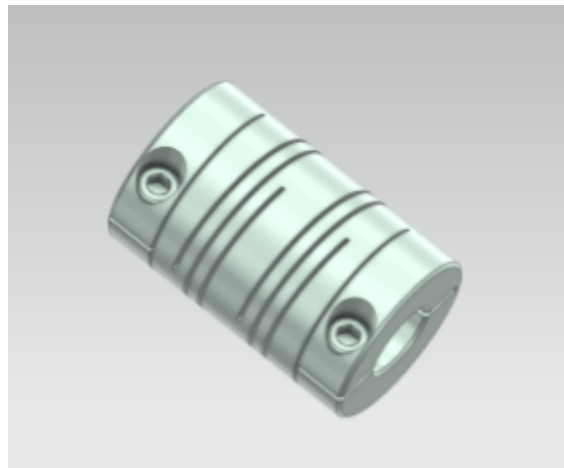


Fig. 2

The shaft coupling model (Figure 2) was sourced from McMaster-Carr and was attached to the motor. Following that was the balanced wheel and main axle. The selected version had an

allowable tolerance of $\pm 2^\circ$ for bending off the center axis of rotation. This was ideal, as it meant we did not need to tilt the jackplate far to induce strain on the system.

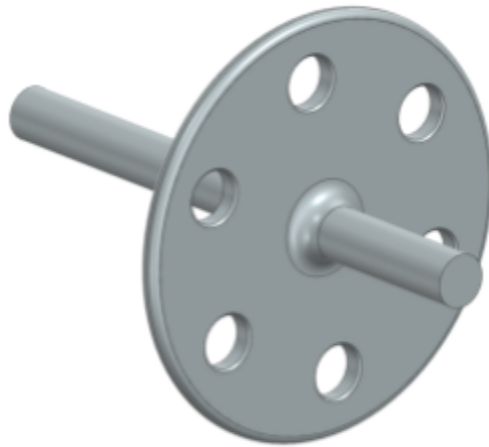


Fig. 3

The balanced wheel diameter needed to be large enough that mounting holes could be drilled through it for attaching weight, but it couldn't be too wide such that it would collide with the baseplate. This resulted in a 5" diameter with $\frac{3}{4}$ " mounting holes drilled. In Figure 3, the model has the axle fused to the wheel, but the actual parts were fabricated separately and then attached during assembly. In theory, this disc could have been made larger, as the motor would eventually be elevated using a block of wood, adding several more inches of clearance, changes to the wheel were deemed unnecessary. The axle is made from low-carbon steel, allowing for easy milling. It is also supported by the L-bracket (Figure 4), which holds a press-fit ball bearing. The bearing served no role in testing and was not designed as a point of failure. This helps with putting stress in the correct locations when tilting the jackplate and helps to keep everything aligned during operation.

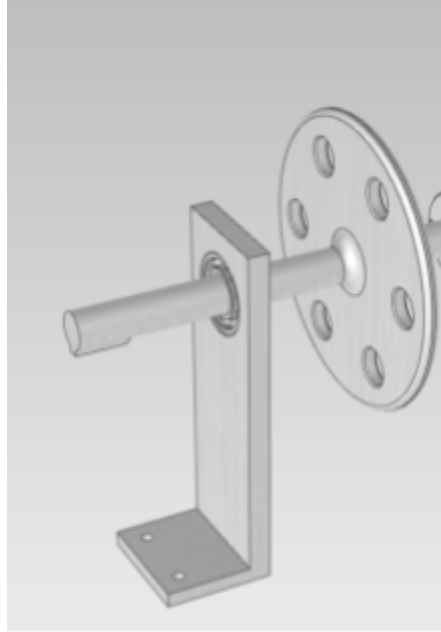


Fig. 4

The driving gear is attached to the end of the drive axle and engages with the receiving axle at 90° (Figure 5). This receiving axle is contained within the bearing housing. As previously mentioned, the straight cut miter gears were selected for their affordability, not their effectiveness. While the efficiency of power transmission isn't monitored, the aforementioned noise profile is the point of focus. With a reduction in constant noise, this would allow easier detection of auditory anomalies created by the mechanical faults. During testing, an issue was found in which the gears would not properly engage due to an incorrect mounting distance on the vertical axis. To fix this, spacer washers were added underneath the receiving axle gear to elevate it to the correct position.

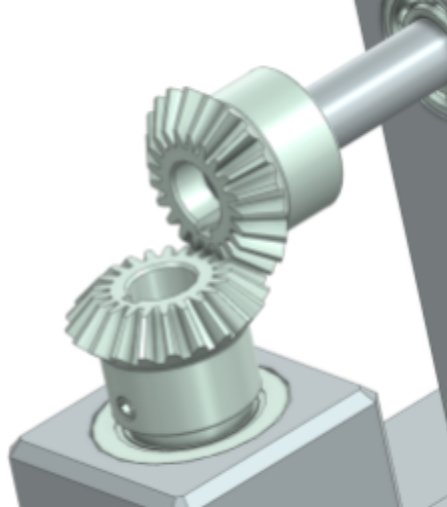


Fig. 5

The transmission needed to be easily serviceable for interchanging the gears, but also secure enough it wouldn't self-dissassemble during operation. The inherent nature of the vertical configuration, in conjunction with the bearing housing, meant that the drive axle gear prevented the receiving axle gear from shifting vertically. However, to swap out the receiving gear, the drive axle needed to be shifted backward, toward the motor. To accomplish this, the drive shaft was milled down so that it could slide freely through the ball bearing mounted in the L-bracket. This meant that by loosening the clamp in the shaft coupling, the drive axle could move freely and be pulled back, allowing the removal of the receiving axle.

The final stage of the assembly was the bearing housing. It needed to serve two purposes: to contain the bearings for testing and to provide a stable mounting system for the receiving axle. In the original concept, the bearings were integrated with a receiving gear, but after transitioning to a miter transmission, a vertical assembly was necessary. The first proposed design was a two-piece collar that would clamp around a bearing sunk into the jackplate. In Figure 6, a cutaway view is shown on the left, with the collar in red, bearing in green, and the jackplate in blue. On the right is the evidence for why this component was deemed over-engineered. For the time frame given, and the machining faculty and facilities at our disposal, it was unreasonable to expect these parts to be available in time.

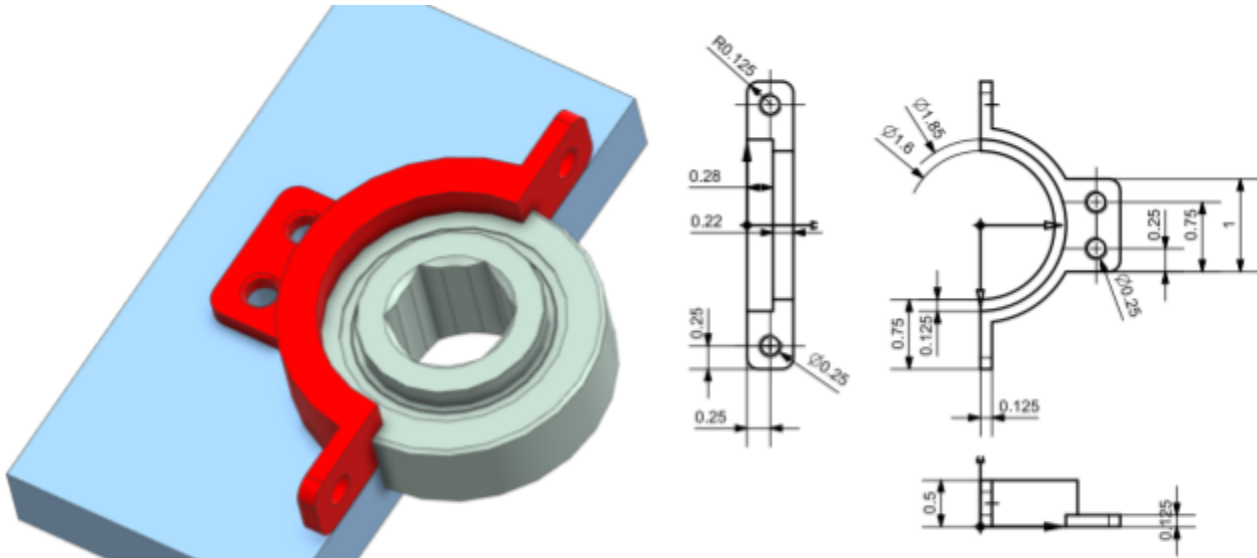


Fig. 6

The reasoning for designing a low-profile part like this was the original vertical clearance from the miter gear transmission. With approximately 2.5" of clearance, the bearing being 0.9", and the miter gear being about 1.2", there was zero effective spacing to work with. This was the cause of elevating the motor higher off the baseplate. This change allowed for the design of the current bearing housing.

The revision of this was to make the part as simple as possible, so it started with a solid block of aluminum. An inset was cut into the top to seat the hex bearing, and a cylinder was milled out of the center along the vertical axis to allow the axle to pass through. A design change was made after the draft (Figure 7) was sent over, adding a second inset at the bottom for another hex bearing. Having two ensured the stability of the receiving axle at high speeds. Four screw-holes were drilled in the bottom for mounting.

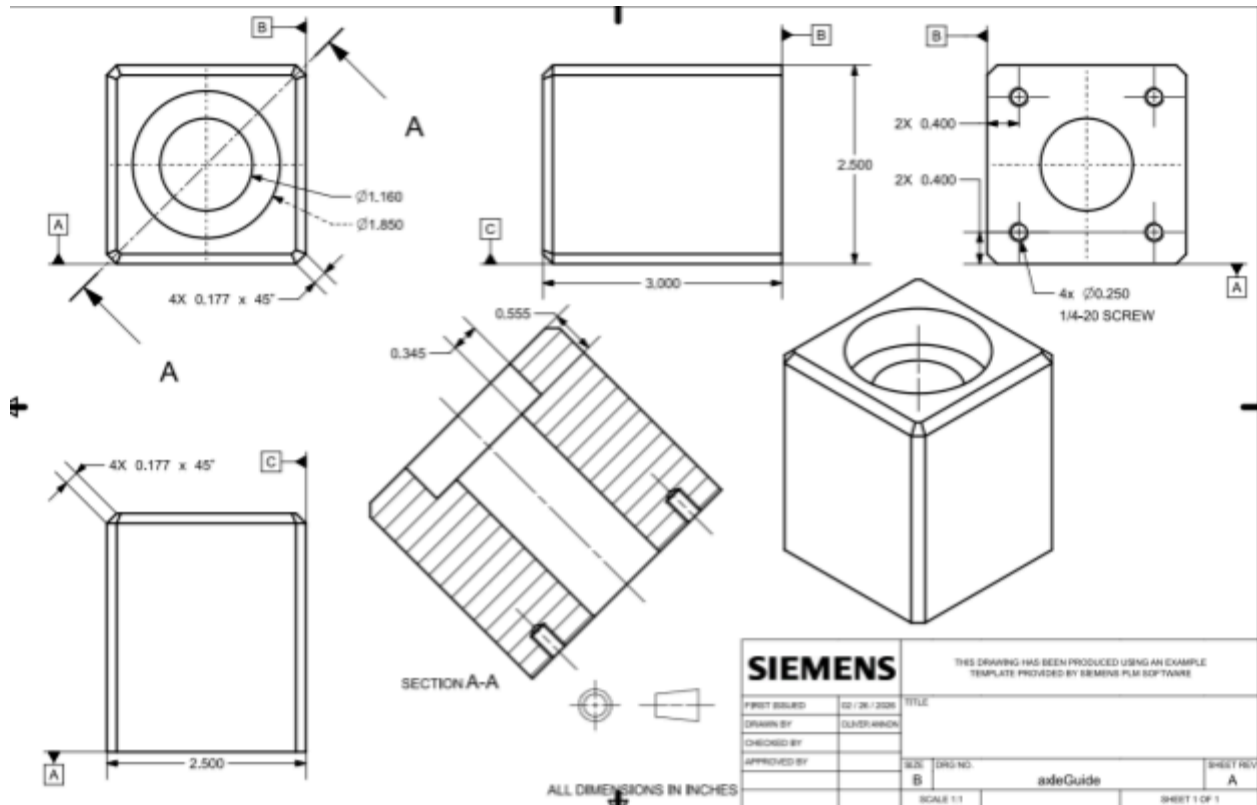


Fig. 7

The jackplate was the last component to be designed, as it was the simplest and relied on the completion of all other components. The holes needed to be drilled in locations that mirror those of the L-bracket and bearing housing. Two more holes were drilled at the corners for anchoring, and then three more were drilled across the opposite side to change the angle of the plate.

The team would like to extend our appreciation to Paul Osborne in the ECE Department at the University of Rochester, who completed all fabrication and assembly of the testbench and provided feedback during the drafting phase and component selection.

STANDARDS

All physical measurements were marked in inches, and angles were measured in degrees. Screw holes were UNC Inch 1/4-20 threads. The individual components were also selected using standard measurements in inches and rotations per minute. This was for the outer diameter, shaft diameter, thickness, mounting distance, and maximum rotational speed. The two motors used in the process were single-phase and three-phase, respectively, with power measured in horsepower. These required proper power distribution from the wall. In the case of the three-phase motor, there was a variable frequency drive with voltage, hp, and current ratings.

CONSIDERATIONS AND CONSTRAINTS OF THE MODEL TRAINING

Considerations

Arduino Sensor

Our team landed on using an Arduino Nano 33 BLE Rev 2 sensor – a microcontroller with an onboard PDM microphone sensor shown in Figure 8. It is an “edge device” that is able to perform all tasks on board - collecting data, processing data and analyzing data. In other words, all actions are performed “locally” on the sensor without the need to send data back and forth to a remote cloud for processing. This board aligns with the project’s broader goal, which is to demonstrate that an effective fault detection model can run on such accessible, low-cost hardware rather than only on specialized, high-end equipment. We wanted to find out how much capability we can push out of a compact and multi-functional board like this.

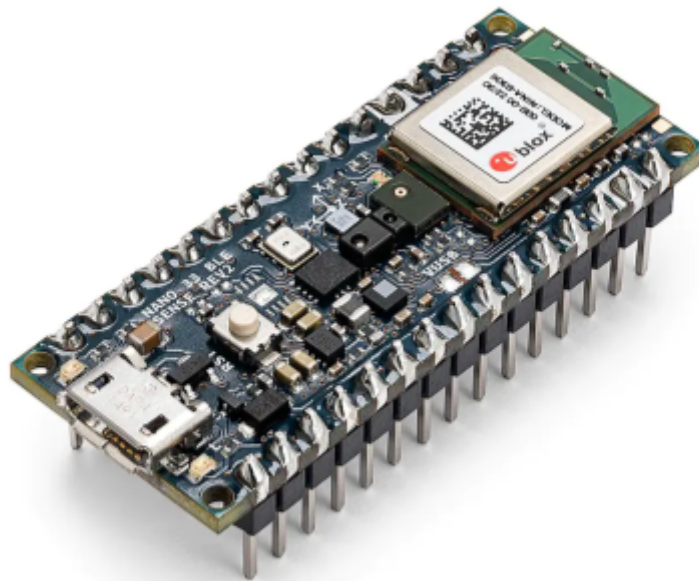


Figure 8. An Arduino board

Edge Impulse

Edge Impulse is an embedded machine learning platform [2]. We chose Edge Impulse as an optimized model training environment for the Arduino sensor. Instead of streaming sensor data to remote servers for processing and sending results back, Edge Impulse models support local inference on an edge device like the Arduino sensor. The platform consolidates and simplifies the full machine learning workflow – data collection, signal pre-processing, model training and evaluation, and deployment – into a single web-based interface [2], shown in Figure 9. It's essentially a pipeline that takes raw input, passes it through processing blocks for feature extraction, then through a learning block that produces the final output.

Edge Impulse is advantageous compared to other alternatives like TensorFlow Lite Micro (TFLM) in which code needs to be manually written for audio capture, feature extraction like FFT and MFE filter bank, and embedded deployment. TFLM also uses a generic runtime interpreter that can execute any TFLM model. Such generality results in an expensive overhead that costs the RAM and flash space of the microcontroller. While Edge Impulse handles quantization to int8, auto-generates the full embedded library with the feature extraction code, and uses a Edge Optimized Neural (EON compiler) that reduces the usage of limited RAM and flash. Specifically, the EON compiler saves flash by generating code for executing a specific model of the user's choice instead of writing a generic engine that can run any model [3]. It also avoids the interpreter overhead by generating straight-line C++ code that executes the model directly rather than, like typical TFLM, using an interpreter that scans through the model at run time.

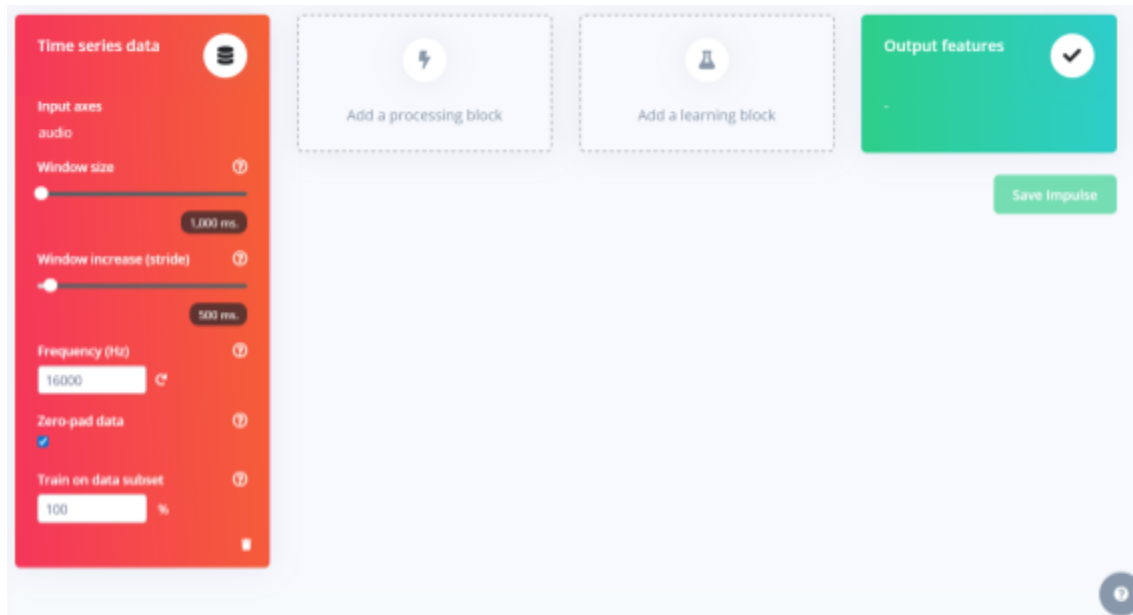


Figure 9. Edge Impulse create impulse page

For this project, we mainly configured “Spectrogram” and “Audio(MFE)” as processing blocks, and “Classification” and “Anomaly Detection (K-Means)” as the learning blocks.

Spectrogram is an effective block for audio feature extraction. While the model can learn from raw audio waveforms directly, it’s rarely efficient because most of the details from one second of audio – which contains thousands of samples – isn’t relevant to distinguishing between different machine states. More importantly, the pattern that would distinguish between healthy and failure states aren’t really visible in the waveform that is in amplitude over time.

The distinctive features for detecting machine faults lie in specific frequency bands that present themselves during runtime. Figure 10 is a spectrogram, essentially a 2D image that shows how those frequencies change over time. With the dimension formed by time on the horizontal axis and frequency on the vertical axis, the color of each pixel shows how much energy is present at that frequency at that moment. Spectrograms turn audio classification into an image recognition problem that better suit convolutional neural networks.

Specifically, the spectrogram processing block produces this representation by first slicing the audio into short overlapping windows. Then, within each window, it measures the energy of frequency present in each band, and outputs an image in the time-frequency domain. The learning block will treat this image as input.

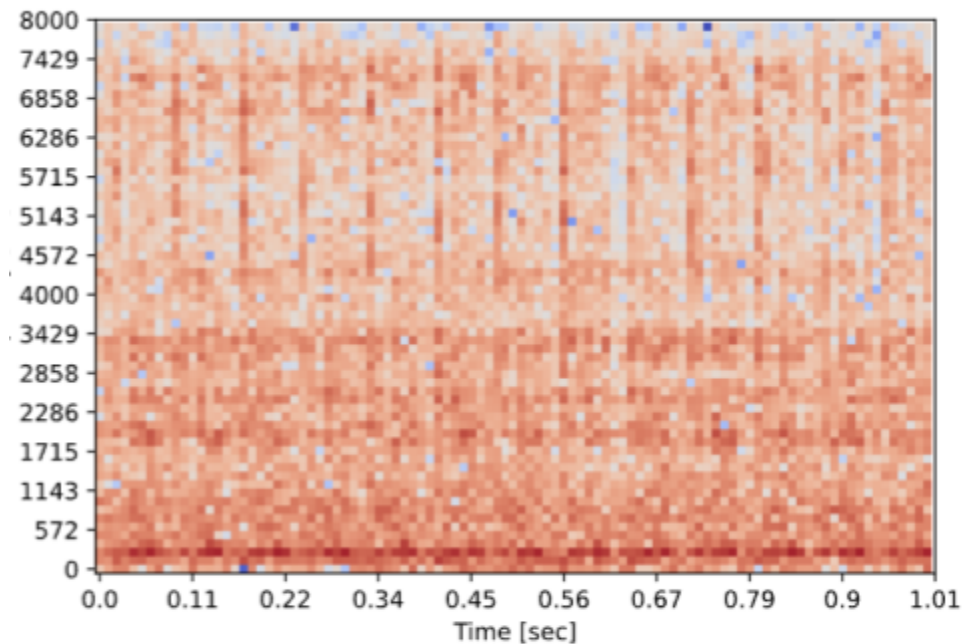


Figure 10 Spectrogram plotting for testbench data

The Audio (MFE) processing block is essentially a spectrogram block that applies Mel-scale, a scale that maps physical frequencies to align with human pitch perception. This means when grouping the frequency into bands, there is more resolution at lower frequencies. While the Mel scale was developed for speech recognition, its emphasis on lower frequencies also suits mechanical fault detection, as many fault features may appear in the low mid frequency range.

The learning block – Classification. A neural network consists of several layers, each layer made up of a number of neurons. The system will randomly initialize the weights of the connections between two adjacent layers of neurons. When given training data as input, the network passes the value through its layers, with each layer adjusting the input based on the current weights. The final layer outputs a probability score for how likely the input belongs to a certain category.

The training works by comparing this output to the correct answer, and, based on that, adjust the weights of the connections between the neurons in the layers, so that future predictions are closer to truth. The process repeats over many inputs and passes using the input data until the network is able to produce reliable predictions on input it has never seen before.

The Anomaly Detection block is a binary detector that will flag a high anomaly score on any samples exhibiting abnormal behavior compared to the feature learned from the input. Thus, this block will only be trained on the healthy dataset in order to flag any non-healthy state, including non-diagnosed faulty state.

Constraints

Sensor Constraints

The scope of this project casts a natural constraint on the choices of equipment and platform. The downside towards this budget-friendly decision is the huge device resource constraints that come with the sensor, which can be considered as a Tiny Machine Learning System. The arduino board only has an available SRAM of 256kB for run time memory, while typical machine learning often requires gigabytes of working memory and storage [2]. As a result, the following factors are repeatedly considered between different model versions:

1. The machine learning model itself has to be small enough to be uploaded on board while leaving enough memory for real-time computation.
2. The inference will run only if there is enough room to store a single neuron layer output before the next layer takes it. This requires careful consideration of the number of neuron layers assigned.
3. Audio window size. Raw audio at 16kHz means 16000 samples per second. Given each sample is about 2 bytes, one second of audio is 32kB of raw data. Having too long of a window size will instantly take up the room for model inferencing.

Edge Impulse Constraints

With a limited knowledge of machine learning going into this project, the team kept mostly default or autotuned parameters that Edge Impulse offers. Committing to Edge Impulse also means accepting its framework - limited menu of processing and learning blocks, and hyper-parameters with bounded range. Even though a certain degree of customization is allowed, a multi-model system customization is proved to be not possible.

To expand on the sensor constraints regarding model training tradeoffs, most of the spectrogram block parameters cost the available SRAM. For example, longer windows and higher FFT length would take up more memory during inferencing. Higher FFT resolution increases the number of computation cycles every inference, which may cause latency. There's also the time-frequency tradeoff in resolution. With our scope focusing on transient events such as clicks from gear failure, time precision matters more than frequency precision. However, for untested failures that can potentially generate continuous mechanical sounds, frequency precision matters more. With our machine producing both cases, the parameter choice is a compromise.

Logistical Constraints

The data collection is naturally constrained because we could only establish a finite set of data within a finite number of environments. Even if we tried to promote data variability, there's still a great chance that the model will be exposed in an acoustically unfamiliar environment when put in real-world use. We also ended with limited types of faulty data - the testbench was only designed to simulate four types of faults, while in the real world there are many more failure modes that the current dataset cannot represent.

The definition of a “healthy” system is limited. What we defined was a newly assembled machine that runs faultlessly in the lab setting. However, our healthy dataset isn't able to count

for the shift of healthy acoustic characteristics over time because, for example, a machine starts wearing off, or even accumulates dust.

There was also significant time consumption when it comes to model training. Depending on the type of trial and error, retraining the model sometimes involves building a whole new project and re-extracting thousands, sometimes tens of thousands, of features from each block used. Given the grand timeline of this project, we had to accept some trade-off between model accuracy and time consumption within the approximate one month of data collection time on the testbench.

Considerations for Model Training

Short-Time Fourier Transform (STFT)

The Fourier transform transforms audio data from time domain into frequency domain. However, one reason a regular Fourier transform isn't enough for audio classification is that it only gives information about what frequencies are present but not the time, while the scope of our project involves the analysis of signals with time-varying frequencies.

The Short-Time Fourier Transform is a solution towards the limitation. STFT divides the signal into smaller “windows” or segments, each is separately analyzed by computing its Fourier transform. Each slice is selected by multiplying the original signal with a finite length window function to make the signal finite within the window and zero outside. Among all windowing choices, the Hann window is commonly used to best avoid spectral leakage, and that is the default window option in Edge Impulse [4]. Consecutive windows offset each other by an amount defined by the term “hop length”, or “window stride” in Edge Impulse.

The STFT is defined as:

$$X[k, m] = \sum_{n=0}^{N-1} x[n + mH] \cdot \omega[n] \cdot e^{-j2\pi kn/N} \quad [5]$$

In the equation, $\omega[n]$ is the windowing function, N is window length, H is hop length, k is frequency bin and m is time frame. When H is less than N , consecutive windows overlap.

After iterating through the input sample length, the result of STFT will be a 2D array that captures both the frequency content and time information. However, there is a tradeoff between the time and frequency resolution from STFT - longer window size provides better frequency resolution but worse time resolution. Each time slice covers more of the signal, so the exact time

of transients that occur within the window will be blurred from averaging. While shorter windows have inverse impact, providing better time localization with less precision in frequency.

For this project, within Edge Impulse, the STFT was configured with following parameter values:

- Frame length: 0.04s
- Frame stride: 0.02s
- FFT length (number of frequency bins): 256
- Noise floor: -52dB

At a sample rate of 16kHz, the frame length in samples at 16kHz:

$$0.04s \times 16000 \text{ samples/sec} = 640 \text{ samples}$$

The frame stride in samples:

$$0.02s \times 16000 \text{ samples/sec} = 320 \text{ samples}$$

The frame overlap percentage:

$$1 - \frac{\text{Frame stride}}{\text{Frame length}} = 1 - \frac{0.02}{0.04} = 0.5 = 50\%$$

The frequency resolution:

$$\frac{\text{Nyquist frequency}}{\text{FFT length}} = \frac{16000/2}{256} = 62.5 \text{ Hz}$$

The total bin number spanning across the 8000Hz Nyquist frequency:

$$\frac{256}{2} + 1 = 129 \text{ bins}$$

Considering the recurring clicks of the faulty state being very close together, we started with a 40ms frame length that provides enough resolution. We selected 50% overlap between each window in order to capture rapid change in frequency content, and compensate for the attenuation at both ends of the Hann window by the subsequent frame. The final configuration was set after rounds of experimental tuning aiming to extract around 3000 features, which is a reasonable amount of board memory use.

Mel-Scale and MFE

The standard Fourier Transform uses a linear frequency scale. While human hearing does not perceive sound linearly, the Mel scale is a rough logarithmic scale used as a better approach to adapt feature extraction in a non-linear manner. The standard formula is as follows:

$$Mel(f) = 2595 \log_{10} \left(1 + \frac{f}{700} \right) \quad [6]$$

The Mel scale is roughly linear at frequencies below 1kHz and starts to grow logarithmically at higher frequency. In the project, the gear failure sound likely falls in the high frequency range. Thus, its harmonics space further from each other, which might result in model over-fitting to specific high-frequency bins that might change with slight motor speed variations. Using the Mel scale can prevent this problem by grouping the higher frequencies more closely together.

The Audio MFE block is a feature extraction process that computes STFT and feeds the results into the Mel filterbank. Mel filterbank is a way to get the energy distribution of the signal that consists of a set of triangular filters in Mel spacing being applied to the power spectrogram [7]. A mel spectrogram can be created by multiplying the power spectrogram value to the weight of the triangular filter, then sum the results. In our experimental project with Audio MFE processing block, 257 frequency bins were configured with FFT length set to 512, and the filter number was set to 32. The block will effectively transform a 257-dimensional vector into a 32-dimensional vector. This transformation forces the model to look at only the energy bands that are perceptually significant, which contributes to the robustness of the model. On the other hand, it also saves a considerable amount of board memory.

Classification Convolutional Neural Network (CNN)

Following feature extraction, a classification-based Convolutional Neural Network (CNN) was used to identify machine operating states from the generated Mel-Frequency Energy (MFE) features. Convolutional neural networks are commonly used in audio and image classification applications because they are effective at learning spatial and spectral patterns within structured input data. In this project, the CNN was responsible for learning relationships between spectral features and the corresponding machine operating conditions.

The classifier used throughout this project was generated and trained within Edge Impulse using its built-in classification learning block. The model architecture primarily consisted of sequential one-dimensional convolutional layers followed by pooling and dropout operations before the final output classification layer. The convolutional layers function by applying learnable kernels across the feature input in order to detect local feature patterns associated with different machine states. During training, these kernels gradually adapt to emphasize spectral characteristics that are useful for distinguishing between Healthy, Gear_Fail, No_Load, and Machine_Off operating conditions.

K-means Anomaly Detection

In Edge Impulse, the Anomaly Detection block is trained only on the healthy data. The model will target several specific clusters of data that represent different spots a healthy state's frequency contents shows up. In this case, the cluster is in the form of frequency bins. The output of the Anomaly Detection block is an anomaly score that reflects the data that are "out-of-distribution".

The K-means equation:

$$W(x) = \sum_{i=1}^k \int_{S_i} |z - x_i|^2 dp(z) [8]$$

The variable S_i is the resulting partition, or clusters, of healthy data. The goal of k-means is to minimize $W(x)$, the variance within the cluster, and to ensure the sum of the squared distance, $|z - x_i|^2$, between each data point and its cluster mean is as small as possible. The process starts with k initial groups, each initialized as a single random point in the feature space. During training, each new data point is added to the group whose mean the new point is the closest. Then, the mean of that group is adjusted to account for the new point, shifting the cluster mean to better represent the normal frequency content of the machine[8].

For anything that falls outside of the definition above, the Anomaly Block will flag them as out-of-distribution. For example, if the gear fails in a way that wasn't recorded for training, the classifier network may be confident that it's "Healthy" because it is the closest thing the network knows. However, the K-means algorithm prevents such guessing by flagging anything that is outside of the established "healthy" boundary. This also includes any undiagnosed mechanical fault.

Air Compressor Dataset Preparation Experiments

Preparation for Data Collection

Parallel to the design and manufacturing process of the workbench, our team started developing a methodology for data collection based on an existing dataset. The goal at this stage was to surface roadblocks early, so we could identify variables our methodology would need to account for during the real data collection process.

Some recognized factors are as follows:

- **Environment** – Background noise varies across spaces.

- **Recording distance** – Microphone distance to source affects signal amplitude.
- **Source directionality** – Source direction relative to the microphone affects signal frequency profile.
- **Model robustness** – Model may overfit to remember specific recording samples.

Data Pre-Processing

The air compressor dataset used throughout early experimentation contained several machine fault categories, each with approximately 250 audio samples. During initial testing, our team observed that environmental conditions, microphone selection, and recording distance had a significant impact on inferencing performance. Because of this, much of our methodology focused on reducing uncontrolled variability between training and deployment conditions.

To maintain acoustic consistency, short audio samples from the dataset were first reconstructed into longer continuous recordings before being played back through a pair of stereo speakers in the same environment used for later inferencing tests. These playback recordings were then rerecorded using the onboard microphone of the Arduino Nano 33 BLE Sense Rev2[2]. This process allowed the dataset to better represent the acoustic characteristics of the final deployment environment rather than relying solely on the original recordings.

Early testing also revealed that models trained using one microphone but deployed using another produced significantly worse inferencing results. This suggested that the model was learning microphone-specific acoustic characteristics in addition to machine fault features. To further reduce variability caused by inconsistent recording distance and playback volume, a custom MATLAB script was developed to apply Root Mean Squared (RMS) normalization with a target RMS value of 0.05. RMS normalization standardized overall signal amplitude across the dataset while preserving the relative structure of the audio waveform.

Several preprocessing and augmentation strategies were also explored in order to improve model robustness and generalization. The first approach involved rerecording the dataset while

introducing a low level of white noise into the environment during playback. This effectively expanded the dataset by allowing each sample to exist in both clean and noisy forms while maintaining consistent spatial acoustics. All recordings were performed using the same speakers, microphone, and environmental conditions in order to minimize additional sources of variability.

The second approach used digital noise augmentation during model training rather than physical rerecording. Edge Impulse applies SpecAugment as part of its preprocessing pipeline [3] [15]. SpecAugment randomly masks regions of the generated spectrogram in both time and frequency domains. This forces the model to learn more generalized fault-related spectral patterns rather than relying too heavily on isolated frequency components or background noise artifacts. Comparing both augmentation approaches suggested that while physical rerecording introduced realistic environmental variability, digital augmentation produced more controlled and repeatable training conditions.

Feature Extraction

Following preprocessing, Mel-Frequency Energy (MFE) feature extraction was selected as the primary signal representation for model training. MFE preprocessing converts raw audio waveforms into compact spectral representations that emphasize meaningful frequency-domain information while significantly reducing data dimensionality. This approach was chosen because machine faults often produce repeatable spectral signatures that are more easily identified in the frequency domain than in raw time-domain audio.

The MFE feature extractor operates by dividing audio into small overlapping windows before applying a Short-Time Fourier Transform (STFT) to generate localized frequency information over time. The resulting frequency bins are then mapped onto the Mel scale, which compresses higher frequency regions while preserving lower-frequency detail commonly associated with mechanical machine behavior. This produces a spectrogram-like representation that can be processed efficiently by convolutional neural networks.

MFE feature extraction was also selected because of its computational efficiency on embedded hardware. Compared to processing raw audio directly, MFE features dramatically reduce input dimensionality and memory requirements while preserving the most informative spectral characteristics of the signal. This made the feature extractor particularly well suited for deployment on the Arduino Nano 33 BLE Sense Rev2, where memory and processing resources were limited.

Window size and frame stride settings were also important design considerations during feature extraction. Smaller windows provide better temporal resolution but may lose long-term frequency information, while larger windows improve frequency resolution at the cost of increased computational requirements. Similarly, frame stride determines the overlap between consecutive windows and directly affects both feature continuity and processing complexity.

Although MFE preprocessing proved effective for embedded deployment, several limitations were observed. Transforming raw audio into spectral representations removes portions of the original temporal waveform information, which may contain additional fault-related characteristics. Additionally, feature quality was highly dependent on window size, frame stride, and environmental consistency during recording. Despite these limitations, MFE preprocessing provided the best balance between computational efficiency, feature quality, and embedded deployment feasibility throughout experimentation.

Training Confusion Matrix

Proceeding to the feature extraction stage, we configured the Audio MFE block. We started from Edge Impulse's auto tuned parameters (frame length 25 ms, stride 10 ms, FFT length 512, 41 filters). These values provided balance between frequency and time resolution for general audio classification. In the validation run, these parameter settings produced strong accuracy reflected through the training confusion matrix (Figure 11) in early experiments. So, we kept these parameters and were ready to use them as a start off points for the official test bench model training.

Confusion matrix (validation set)

	BEARING_FAIL	FLYWHEEL_FAIL	HEALTHY	LIV_FAIL	NOISE
BEARING_FAIL	100%	0%	0%	0%	0%
FLYWHEEL_FAIL	1.6%	98.4%	0%	0%	0%
HEALTHY	0%	0%	100%	0%	0%
LIV_FAIL	0%	0%	0%	100%	0%
NOISE	0%	0%	0%	0%	100%
F1 SCORE	0.99	0.99	1.00	1.00	1.00

Figure 11. The confusion matrix during model training stage of the air compressor dataset

Testing Confusion Matrix

When we tested the model on a new set of testing data that the model has never seen, in Figure 12, the accuracy dropped from roughly 99% to about 78-89%. This informed us about the accuracy gap between training and testing stage that strong accuracy doesn't necessarily reflect how a model behaves on testing in a real-world environment. This shaped how we approached test bench data collection and model training.

Actual\Predicted	Bearing	Flywheel	Healthy	LIV	Noise
Bearing	89.30%	3.60%	0%	7.10%	0%
Flywheel	12.50%	87.50%	0%	0%	0%
Healthy	0%	21.40%	78.60%	0%	0%
LIV	0%	0%	0%	100%	0%
Noise	0%	0%	0%	0%	100%

Figure.12. The confusion matrix during actual testing stage of the air compressor dataset

The Test Bench Dataset

Data Collection

As soon as the Test Bench was completed data collection began. Considering the aforementioned variables when capturing data we applied the following to our procedure of capturing audio from the Test Bench.

- Multi-Directional Recordings
- Variable Recording Spaces
- Multiple Takes For Each Machine State
- Variable Frequency Driver (VFD) Consistency

The machine was recorded from three different directions to help improve our models ability to identify new data. This robustness in our raw data was designed to help the model gain more information about the machine and thus have more accurate predictions. Additionally, the machine was moved to and recorded in three different locations. These locations varied in negligible amounts of background noise (HVAC vs Non HVAC) and room acoustics (Carpet vs Stone Walls) but were still considered as a means to help improve the models accuracy.

Furthermore, every time the machine was recorded on a new take the machine was disassembled and reassembled into as proper of a state as possible. This allowed us to capture multiple takes of the same machine state from the same position, while still having the new data provide something more to the data set as a whole. Finally, another variable that had to be kept in check was the speed of the motor. To keep this as consistent as possible we had our machine operator turn the Variable Frequency Driver (VFD) to the same position during operation. Regardless of room, state or recording position the VFD was always set to 50%. This was an attempt to remove any audible changes in the motor as it was not a component we were considering for fault.

Below you can see our machine set up for recording in one of our recording spaces.

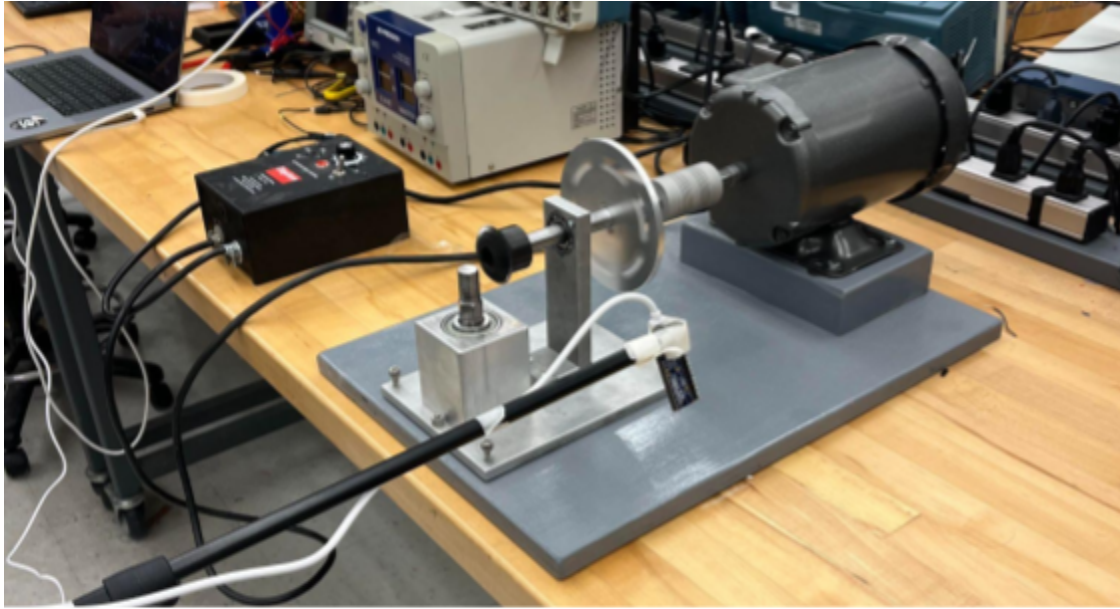


Figure 13. Recording from Position 1, Running in No_Load State

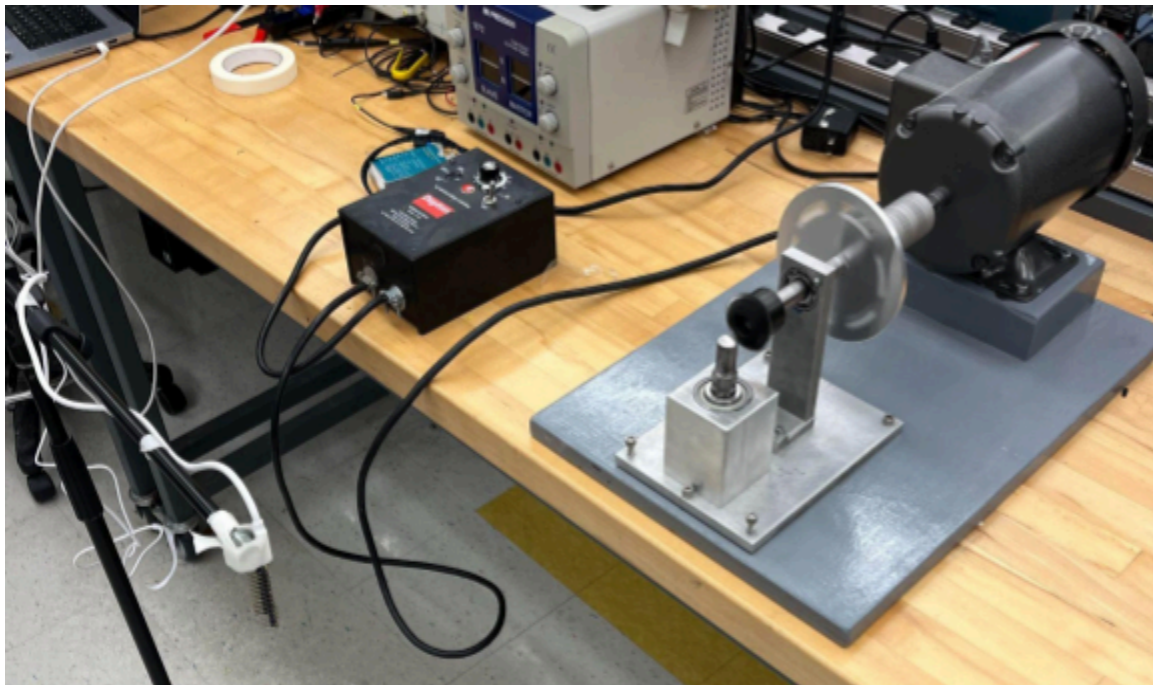


Figure 14. Recording in Position 2, Running in No_Load State

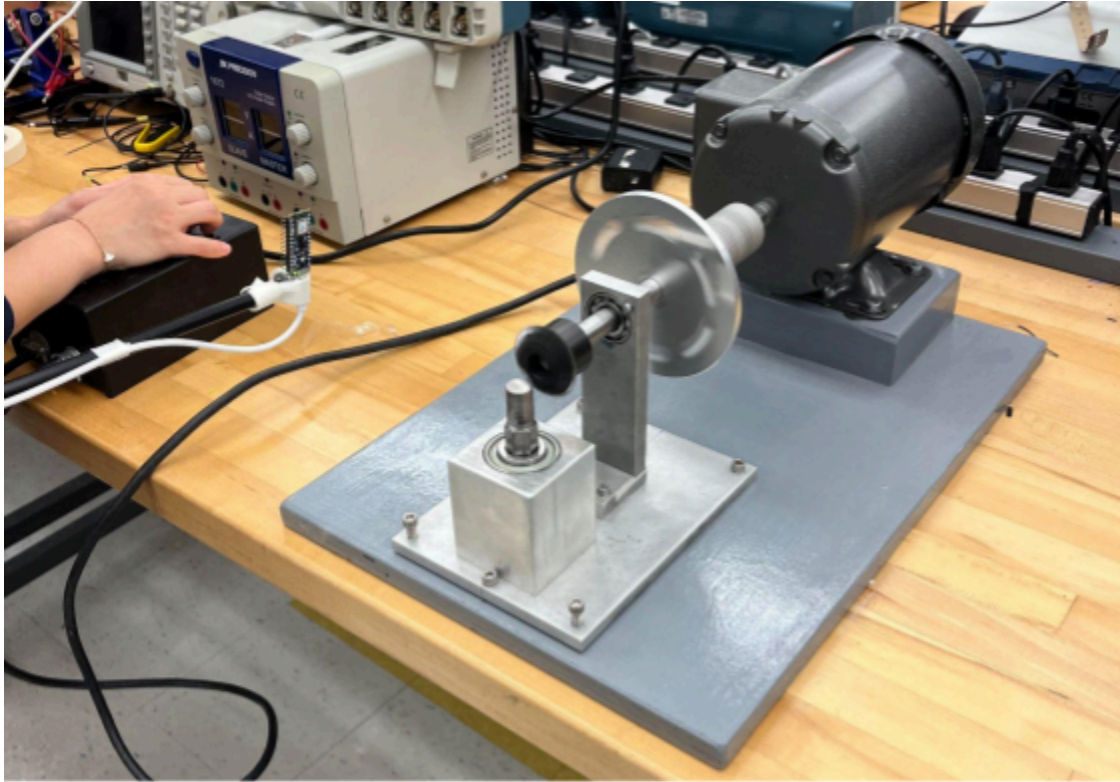


Figure 15. Recording in Position 3, Running in No_Load State

The recording positions here all place the microphone one foot away from the machine at varied angles. With position 1 at 0° or 360° , Position 2 at 90° , and Position 3 at 180° we believed that our recordings would allow us to observe the machine at any point along this 180° arc.

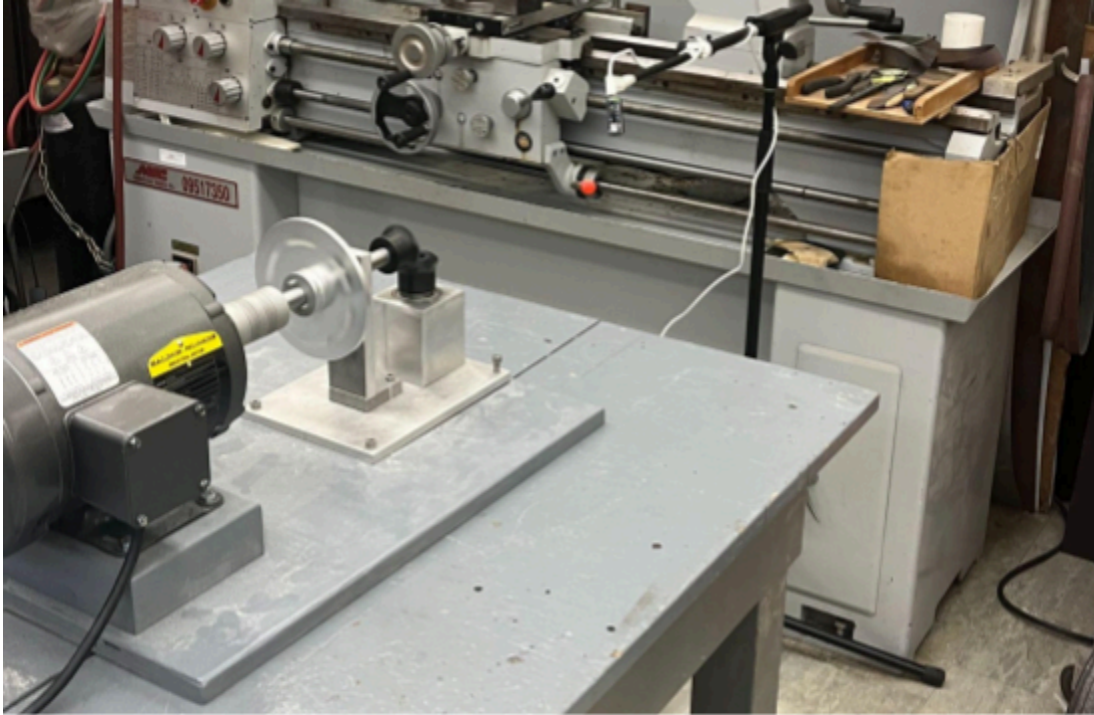


Figure 16. Testbench running with induced gear fault

Model Training

With our new data we were able to make a variety of different models that considered both model hyperparameters and different model architectures. However, our search to find these optimal hyperparameters and model architectures was limited by our project's scope, as the need to hire an engineer to design a monitoring system would be an unnecessary expense for a small business. Thus, we limited ourselves to editing our models solely via Edge Impulse. [2] This did allow us to change windowing parameters for our feature extractor, number of training cycles, learning rate, type of feature extractor, data augmentation, and learning processes. (Classification vs Anomaly Detection)

With all this at our disposal we first made a simplified model that was tasked with distinguishing between two different machine states. This model performed well after being trained on a set of data obtained from a single recording session. From here more data was collected and more complex models were developed to include four different machine states in total. A limitation

that we encountered during testing was the amount of usable memory on our Arduino Board. [2] When developing different models we were not able to deploy more complex feature extractors or learning architectures. This forced us to only consider a classifier that either had a spectrogram or a Mel-Frequency feature extractor.

Results

After varying the previously discussed preprocessing and model parameters within Edge Impulse, several machine learning models were trained and evaluated using live verification testing on the custom machine testbench. The final selected model produced the following verification confusion matrix shown in Figure 17:

Actual\Predicted	<i>Gear Fail</i>	<i>Healthy</i>	<i>No Load</i>	<i>Machine Off</i>
Gear Fail	100%	0%	0%	0%
Healthy	87%	13.00%	0%	0%
No Load	0%	0%	100%	0%
Machine Off	0%	0%	0%	100%

Figure 17. Verification confusion matrix for the final model

The final deployed model used a one-dimensional convolutional neural network (1D CNN) consisting of two convolutional layers with Mel-frequency feature extraction and four output labels corresponding to the machine states:

- Healthy
- Gear_Fail
- No_Load
- Machine_Off

Although the model did not achieve the level of reliability necessary for practical deployment, several important performance trends were observed. During live inferencing, all faulty machine states, Gear_Fail, No_Load, and Machine_Off, were identified correctly in every recorded test sample. Additionally, the model never classified a faulty machine state as Healthy.

However, the model demonstrated significant difficulty distinguishing between Healthy and Gear_Fail operating conditions. Approximately 87% of healthy samples were incorrectly classified as Gear_Fail during verification testing. This resulted in a high false-positive rate for fault detection despite the model's ability to consistently recognize abnormal operating conditions.

In addition to the primary classifier model, a proposed multi-model implementation combining both classification and anomaly detection techniques was also explored within Edge Impulse. Due to memory and processing limitations of the Arduino Nano 33 BLE Sense Rev2, this implementation could not be fully deployed for live verification testing. However, training results for the proposed architecture were successfully generated.

	ANOMALY	GEAR_FAIL	HEALTHY	MACHINE_OFF	NO_LOAD	UNCERTAIN
ANOMALY	-	-	-	-	-	-
GEAR_FAIL	13.4%	84.6%	0%	0%	0%	2.0%
HEALTHY	0%	4.3%	93.7%	0%	0%	1.9%
MACHINE_OFF	0%	0%	0%	100%	0%	0%
NO_LOAD	0%	0%	0%	0%	100%	0%
F1 SCORE	0.00	0.89	0.97	1.00	1.00	

Figure 18. Training confusion matrix for proposed multi-model implementation

The multi-model approach demonstrated the potential for combining known fault classification with anomaly detection in order to identify previously unseen machine states.

Standards and Frameworks

ISO 13374 – Condition Monitoring and Diagnostics of Machines

ISO 13374 provides a framework for developing machine condition monitoring and diagnostic systems [9]. The standard outlines how monitoring systems should collect raw machine data directly from sensors in order to evaluate machine health and detect abnormal operating conditions. Additionally, the standard describes the use of data manipulation techniques to process and standardize collected data without excessively distorting the original signal. ISO 13374 also defines a structured monitoring process in which systems detect faults, assess machine health, and potentially predict future machine conditions. Finally, the standard emphasizes that monitoring systems should communicate machine states and diagnostic information clearly to the user.

This project applied several principles described in ISO 13374 throughout the system design process. First, all training and inference data were collected directly from the machine using the onboard microphone of the Arduino Nano BLE Sense Rev2. Next, preprocessing techniques such as RMS normalization and Mel-frequency feature extraction were used to standardize audio samples prior to classification. Finally, the completed system displayed predicted machine states and inference confidence values during live testing.

Although the project follows many of the architectural principles described in ISO 13374, the final system has several limitations. The model was designed primarily for current-state classification rather than long-term prognostic assessment. As a result, the system could identify present machine states but could not predict future degradation or estimate remaining machine life. Additionally, the project relied solely on acoustic data from a single microphone rather than the multi-sensor approaches commonly used in industrial condition monitoring systems.

ISO 10816 – Mechanical Vibration Evaluation of Machines

ISO 10816 provides standardized methods for evaluating the mechanical condition of rotating machinery using vibration measurements [10]. The standard defines acceptable vibration severity ranges and operating conditions that can be used to determine whether a machine is functioning normally or experiencing mechanical faults. The standard outlines procedures for collecting vibration data directly from machinery and evaluating machine condition based on measured vibration amplitudes. ISO 10816 groups machine behavior into severity zones that represent acceptable, unsatisfactory, or potentially dangerous operating conditions. These classifications are intended to support fault detection and preventative maintenance practices in industrial systems.

Although this project did not use traditional vibration sensors, the overall objective closely relates to the goals of ISO 10816. The developed system attempted to identify abnormal machine conditions such as Gear_Fail and No_Load states through acoustic analysis rather than direct vibration measurement. Similar to the purpose of ISO 10816, the project focused on determining whether the machine was operating under healthy or faulty conditions using measurable signals collected directly from the machine. Additionally, the use of real-time inferencing on the Arduino Nano BLE Sense Rev2 aligns with the broader goal of continuous machine condition monitoring.

The project differs significantly from ISO 10816 in several ways. First, the system relied entirely on audio recordings rather than calibrated vibration measurements, which are the primary focus of the standard. Second, the model produced probabilistic classification outputs rather than standardized vibration severity values. Finally, the system did not achieve the level of reliability or repeatability typically required for industrial machine evaluation systems, limiting its suitability for safety-critical applications.

NIST AI Risk Management Framework (AI RMF)

The NIST AI Risk Management Framework (AI RMF) provides guidelines for designing, developing, and evaluating artificial intelligence systems in a safe, reliable, and trustworthy manner [11]. The framework was developed by the National Institute of Standards and Technology (NIST) to help organizations identify and reduce risks associated with AI systems. The AI RMF outlines several characteristics that trustworthy AI systems should demonstrate, including reliability, safety, transparency, explainability, and accuracy. The framework emphasizes that AI systems should be tested under realistic conditions and continuously monitored to ensure consistent performance. Additionally, the framework encourages developers to understand the limitations of their models and communicate uncertainty or risk clearly to users.

This project relates strongly to the goals of the NIST AI RMF because the system was designed to make automated decisions about machine health using machine learning inferencing. Several aspects of the project align with the framework's recommendations. First, the model was tested both using validation data in Edge Impulse and through live inferencing on the physical testbench[3]. Second, the system communicated prediction confidence values alongside predicted machine states in order to provide additional context for model decisions. Finally, the project included evaluation of confusion matrices to better understand where the model performed reliably and where classification errors occurred.

Although the project followed several recommendations outlined in the NIST AI RMF, the final system still had significant limitations in reliability and consistency. Confusion matrix results showed that certain machine states were misclassified more frequently than others, indicating that the model did not generalize equally across all operating conditions. Additionally, the system was only tested on a controlled experimental testbench rather than under diverse real-world industrial conditions. Another limitation is that the model functioned largely as a black-box classifier, meaning that individual inferencing decisions could not always be fully explained or interpreted. These limitations reduce the trustworthiness of the system and would

likely prevent deployment in safety-critical industrial environments without additional testing and refinement.

Engineering Practices and Methodologies

Data Collection and Processing

Audio samples were collected directly from the custom machine testbench using the onboard microphone of the Arduino Nano BLE Sense Rev2. Data was collected for each machine state, including Healthy, Gear_Fail, No_Load, and Machine_Off. Using the same hardware for both training and deployment helped maintain consistency between recorded training data and live inferencing conditions. Long recordings were later segmented into smaller windows within Edge Impulse to increase the number of usable training samples and improve model training efficiency[3].

Prior to feature extraction, RMS normalization was applied to standardize the amplitude of audio recordings across the dataset [12]. This preprocessing step reduced variation caused by inconsistent recording volume and allowed the model to focus more heavily on machine characteristics rather than signal loudness. However, excessive normalization can potentially reduce important amplitude-related fault information and cannot fully remove environmental noise from recordings.

Windowing and frame stride settings were also used during preprocessing in order to divide long audio recordings into smaller overlapping samples suitable for machine learning processing [8]. Proper window selection is important because smaller windows may lose long-term machine behavior information, while larger windows increase computational requirements and processing time.

After preprocessing, Edge Impulse's Mel-frequency feature extractor was used to convert raw audio signals into spectral representations prior to classification [3][13]. Mel-frequency features

reduce raw audio dimensionality while emphasizing meaningful frequency-domain information commonly used in audio classification systems. This approach helped the model focus on spectral characteristics associated with different machine states. However, transforming raw audio into spectral features may remove temporal information that could otherwise be useful for fault detection.

To further improve robustness and reduce overfitting, random masking augmentation techniques were applied during training [14]. This process randomly masked portions of the input spectrogram across both time and frequency domains. Random masking encourages the model to learn more generalized patterns rather than memorizing specific spectral regions from the training dataset. However, excessive augmentation may remove important fault information and create unrealistic training samples.

Machine Learning Architecture

The final model used a one-dimensional convolutional neural network (1D CNN) architecture to classify machine operating states based on extracted audio features [11]. The model consisted of two convolutional layers followed by pooling and dropout layers before the final classification output. CNN architectures are well suited for audio classification tasks because they can identify local feature patterns while maintaining relatively low computational complexity compared to fully connected neural networks.

Pooling layers were included after each convolutional layer to reduce feature dimensionality and computational requirements. This helped improve embedded deployment efficiency while also reducing overfitting [16]. However, pooling can potentially remove fine-detail information that may be important for distinguishing similar machine states.

Dropout regularization was also incorporated into the architecture in order to reduce overfitting during training [17]. By randomly disabling portions of the network during training, dropout encourages the model to learn more generalized feature representations rather than memorizing

the training dataset. Although dropout improves generalization, excessive dropout may reduce overall learning performance and increase training time.

Model Validation and Deployment

Model performance was evaluated using confusion matrices and additional classification metrics such as accuracy, precision, recall, and F1-score [18]. These metrics provided insight into how reliably the model classified each machine state and helped identify patterns of misclassification between classes. Both Edge Impulse validation results and live inferencing tests performed on the physical machine testbench were analyzed during evaluation.

The completed model was deployed to the Arduino Nano BLE Sense Rev2 using Edge Impulse's generated Arduino library[3]. During live operation, the system continuously recorded approximately two seconds of audio, processed the sample locally on the embedded processor, and produced predicted machine states with confidence values. This allowed the system to demonstrate practical embedded Edge AI inferencing without relying on external computing systems.

Despite successful deployment, several hardware limitations affected overall system performance. Limited memory and processing power restricted model complexity and contributed to increased inference latency. Additionally, inferencing delays prevented fully continuous monitoring, and environmental noise during live testing occasionally reduced classification reliability.

Discussion

Although the final model did not achieve deployment-level reliability, the project successfully demonstrated the feasibility of embedded audio-based machine condition monitoring using Edge Impulse and the Arduino Nano 33 BLE Sense Rev2. The system was capable of consistently detecting abnormal machine behavior and never classified a faulty machine state as healthy

during live testing. This suggests that the model learned several meaningful fault-related audio characteristics despite overall classification limitations.

The primary limitation observed during testing was the model's inability to consistently distinguish between Healthy and Gear_Fail operating conditions. The large number of false-positive Gear_Fail classifications suggests that the extracted audio features for these two states shared similar spectral characteristics. As a result, the model struggled to develop clean decision boundaries between healthy and faulty operation.

Hardware limitations also played a major role in overall system performance. The Arduino Nano 33 BLE Sense Rev2 imposed strict memory and computational constraints that limited both model complexity and real-time inferencing capability. These constraints prevented deployment of larger architectures and restricted experimentation with more advanced multi-model implementations.

Another important observation made during testing was that model behavior changed depending on microphone placement relative to the machine. This suggests that acoustic machine monitoring systems may benefit significantly from multi-perspective inferencing using multiple microphones positioned at different locations around the machine. Incorporating multiple acoustic viewpoints could potentially improve classification reliability by capturing a more complete representation of machine behavior.

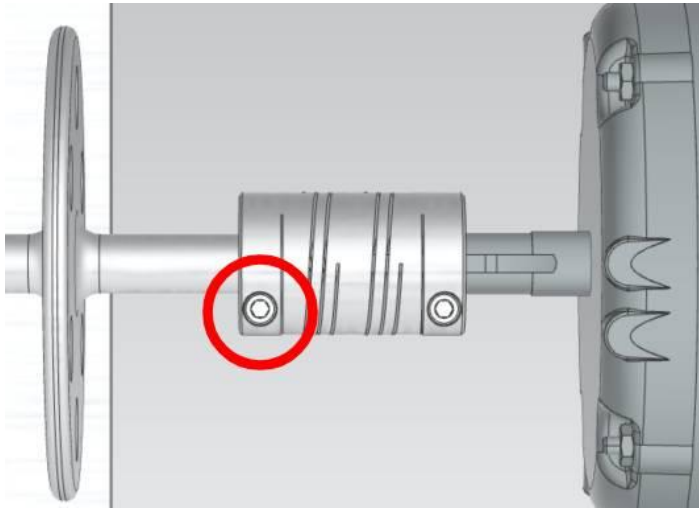
Future work could also include combining traditional classification models with anomaly detection systems operating in parallel. A combined architecture would allow the system not only to classify known machine faults but also detect when the machine enters previously unseen operating conditions. Although hardware limitations prevented full deployment of this approach during the project, preliminary training results suggest that this methodology may improve the robustness of future machine monitoring systems.

INSTRUCTION MANUAL

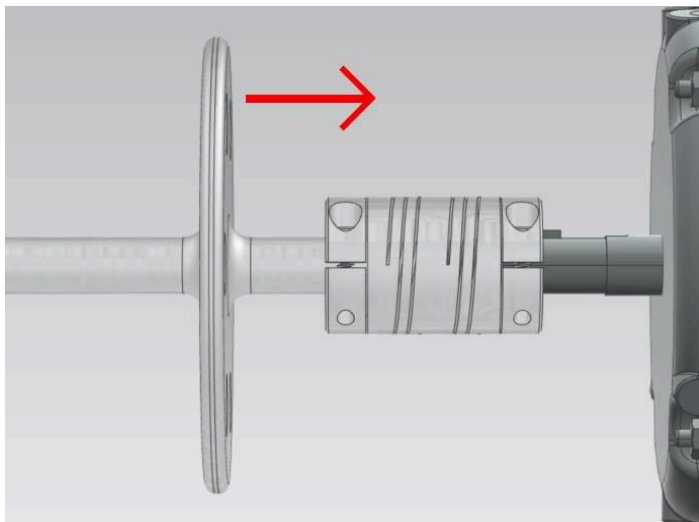
Basic Operation of the Test Bench

Swapping Gears

Step 1: Using an allen wrench, loosen the end of the shaft coupling connected to the drive axle.



Step 2: Slide the drive axle backward, towards the motor.



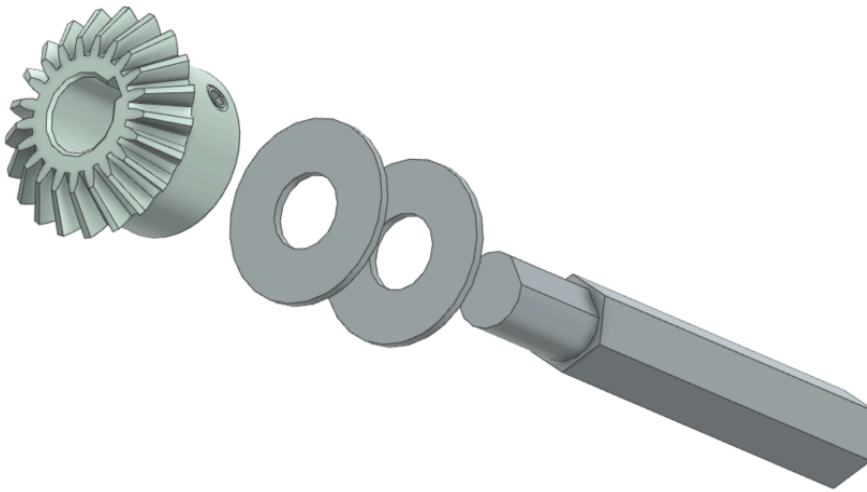
Step 3: Select a HEALTHY gear, and slide it onto the drive axle with the teeth facing away from the motor.

Step 4: Ensure that the flat spot on the edge of the axle is aligned with the gear's set screw, and then tighten the screw with an Allen wrench until firm.

Step 5: Select your receiving gear type and the short, hexagonal receiving axle.

Step 6: Slide the included spacer washers onto the round end of the axle.

Step 7: Slide the gear onto the round end of the axle, ensuring the flat spot is aligned with the set screw, and that the teeth face away from the hexagonal end. It should sit directly touching the spacers.



Step 8: Tighten the set screw until firm.

Step 9: Drop the assembled receiving axle into the bearing assembly. Ensure that the axle is seated through both bearings. If the axle isn't seated, the spacers will not make contact with the upper bearing.

Step 10: Push the drive axle forward until its gear engages the receiving axle gear. Ensure that the teeth between the two gears have meshed correctly.

Step 11: Tighten the clamp on the shaft coupling.

Step 12: Manually turn the drive axle to verify everything is secured.

Angling the Jackplate

Step 1: Tighten the screws anchoring the jackplate at the corners closest to the motor.

Step 2: Turn the screws at the far corners clockwise to lower the plate, and counterclockwise to raise the angle.

Step 3: Turn the center screw on the far edge from the motor in the opposite direction of the corner screws.

Step 4: Manually rotate the drive shaft to verify everything is secured.

Adding or Removing Weights

Step 1: Insert an x/x" bolt through one of the holes of the balanced wheel towards the motor. The bolt head should face the bearing assembly.

Step 2: Add as much weight as desired to the bolt using washers.

Step 3: Apply Loctite Blue to a nut, then screw it onto the bolt, ensuring the washers have no space to move. The Loctite will prevent the nut from unscrewing itself during operation.

Step 4: Manually rotate the drive shaft to verify everything is secured.

Starting Up the Motor

Step 1: Manually rotate the drive shaft to verify everything is secured.

Step 2: Check the front of the VFD, ensure that the dial is set to zero, and that the power switch is in the OFF position.

Step 3: Plug the VFD into a wall outlet.

Step 4: Move the power switch to the ON position.

Step 5: Slowly increase the dial to the desired speed.

Capturing Data with the Nano 33 BLE

Step 1: Set the Machine to the desired state

Step 2: Deploy Recording Arduino Script to Nano BLE and close the Arduino IDE completely after uploading.

Step 3: Go to correct directory of the Python script in Terminal to prepare for recording

Step 4: Attach the Arduino Board to a decoupled stand and point the microphone at the machine from the desired position

Step 5: Turn the machine on

Step 6: Run the Python script to begin recording

Step 7: Stop the Python Script to complete recording

Inferencing with Edge Impulse Models

Step 1: Set the Machine to the desired state

Step 3: Attach the Arduino Board to a decoupled stand and point the microphone at the machine from the desired position

Step 4: Deploy the desired model onto the Arduino Board

Step 5: Turn the machine on if necessary

Step 6: Read Serial Monitor Outputs of the Arduino Script

REFERENCES

- [1] F. Ferraz Júnior, R. A. F. Romero, and S.-J. Hsieh, “Machine Learning for the Detection and Diagnosis of Anomalies in Applications Driven by Electric Motors,” *Sensors*, vol. 23, no. 24, p. 9725, Jan. 2023, doi: <https://doi.org/10.3390/s23249725>.
- [2] Arduino, “Nano 33 BLE Sense Rev2 Datasheet,” ABX00069. [Online]. Available: <https://docs.arduino.cc/resources/datasheets/ABX00069-datasheet.pdf>
- [3] C. Banbury et al., “Edge Impulse: An MLOps Platform for Tiny Machine Learning,” arXiv:2212.03332, 2023. [Online]. Available: <https://arxiv.org/abs/2212.03332>
- [4] “Ni.com/instrument-fundamentals next understanding ffts and windowing overview,” Understanding FFTs and Windowing , <https://download.ni.com/evaluation/pxi/Understanding%20FFTs%20and%20Windowing.pdf>
- [5] M. Müller, “Content-based audio retrieval,” SpringerLink, https://link.springer.com/chapter/10.1007/978-3-319-21945-5_7
- [6] M. Hasegawa-Johnso, Review semitones Mel Erb summary lecture 6: Frequency scales: Semitones, Mels, <https://courses.grainger.illinois.edu/ece417/fa2020/slides/lec06.pdf>
- [7] Filter banks and log-Mel Spectrograms, <https://apxml.com/courses/applied-speech-recognition/chapter-2-feature-extraction-for-speech/filter-banks-log-mel-spectrograms>
- [8] J. B. MacQueen, Some methods for classification and analysis of Multivariate observations (1967) , <https://scispace.com/papers/some-methods-for-classification-and-analysis-of-multivariate-4pswti19oz>
- [9] International Organization for Standardization, ISO 13374-1:2003 Condition Monitoring and Diagnostics of Machines — Data Processing, Communication and Presentation — Part 1: General Guidelines, Geneva, Switzerland: ISO, 2003.
- [10] International Organization for Standardization, ISO 10816-1:1995 Mechanical Vibration — Evaluation of Machine Vibration by Measurements on Non-Rotating Parts — Part 1: General Guidelines, Geneva, Switzerland: ISO, 1995.

- [11] National Institute of Standards and Technology, Artificial Intelligence Risk Management Framework (AI RMF 1.0), Gaithersburg, MD, USA: NIST, Jan. 2023.
- [12] D. O’Shaughnessy, Speech Communications: Human and Machine, 2nd ed. Piscataway, NJ, USA: IEEE Press, 2000.
- [13] A. V. Oppenheim and R. W. Schaffer, Discrete-Time Signal Processing, 3rd ed. Upper Saddle River, NJ, USA: Pearson, 2010.
- [14] S. Davis and P. Mermelstein, “Comparison of parametric representations for monosyllabic word recognition in continuously spoken sentences,” IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. 28, no. 4, pp. 357–366, Aug. 1980.
- [15] D. S. Park et al., “SpecAugment: A simple data augmentation method for automatic speech recognition,” in Proc. Interspeech 2019, Graz, Austria, Sep. 2019, pp. 2613–2617.
- [16] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” Nature, vol. 521, no. 7553, pp. 436–444, May 2015.
- [17] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA, USA: MIT Press, 2016.

APPENDIX A: Source Code

A.1.1 Raw data recording script

```
#include <PDM.h>
// buffer
short sampleBuffer[512];
volatile int samplesRead;
void setup() {
  Serial.begin(1000000); // High speed for audio data
  while (!Serial); // Wait for Python script to open the port
  // Configure the data receive callback
  PDM.onReceive(onPDMdata);
  // Initialize PDM with 1 channel (mono) at 16 kHz
  if (!PDM.begin(1, 16000)) {
    while (1); // Halt if microphone fails to start
  }
}
void loop() {
  // Wait for samples to be read
  if (samplesRead > 0) {
    // Write the raw binary data to the serial port
    Serial.write((uint8_t*)sampleBuffer, samplesRead * 2);
    samplesRead = 0;
  }
}
void onPDMdata() {
  // Query the number of bytes available
  int bytesAvailable = PDM.available();

  // Read into the sample buffer
  PDM.read(sampleBuffer, bytesAvailable);

  // 16-bit samples, so samples = bytes / 2
  samplesRead = bytesAvailable / 2;}
```

A.1.2 Python recording script

```
import os
import time
import wave
import serial
import serial.tools.list_ports
import datetime
import uuid
PORT = "/dev/cu.usbmodem1401" # replace with the corresponding Arduino port of the target operating system
BAUD = 1000000 # must match Serial.begin(1000000) in IDE
OUTPUT_DIR = "./recordings" # where WAVs will be saved
LABEL = "audio" # prefix for the file name
SAMPLE_RATE = 16000 # matches PDM frequency
CHANNELS = 1
SAMPLE_WIDTH = 2 # 16-bit audio
CHUNK_SIZE = 4096 # bytes per read
# Open serial port
print(f"\nOpening {PORT} at {BAUD} baud...")
ser = serial.Serial(PORT, BAUD, timeout=1)
time.sleep(2)
ser.reset_input_buffer()
# create output directory
os.makedirs(OUTPUT_DIR, exist_ok=True)
# create one WAV file
timestamp = datetime.datetime.now().strftime('%Y-%m-%d_%H-%M-%S')
uid = str(uuid.uuid4())[-12:]
filename = f"{LABEL}_{uid}_{timestamp}.wav"
filepath = os.path.join(OUTPUT_DIR, filename)
with wave.open(filepath, 'wb') as wav_file: # open path and initialize recording parameters
    wav_file.setnchannels(CHANNELS)
    wav_file.setsampwidth(SAMPLE_WIDTH)
    wav_file.setframerate(SAMPLE_RATE)
    print(f"\nRecording to {filepath}")
    print("Press Ctrl+C to stop.\n")
    try:
        while True:
            raw = ser.read(CHUNK_SIZE)
            if raw:
                wav_file.writeframes(raw) # write data
    except KeyboardInterrupt:
        print("\nStopped recording.")
    finally:
        ser.close()
```

A.2 Data Normalization Matlab Script

```
inputFolder = 'normalized_Data';    % Folder with original WAV files
outputFolder = 'normalized_audio';  % Folder to save normalized files
targetRMS = 0.05;                  % Desired RMS amplitude

cd C:\Users\raffa\Desktop\dataLocation % Move Directories
audioFiles = dir

if ~exist(outputFolder, 'dir') % make a save folder
    mkdir(outputFolder);
end

for k = 10:15
    fileName = strcat(audioFiles(k).name)
    filePath = fullfile(inputFolder, fileName);

    [y, fs] = audioread(fileName);          % Read audio file

    currentRMS = sqrt(mean(y.^2))           % Compute current RMS
    currentRMS = currentRMS(1,1)

    % Avoid division by zero for silent files
    if currentRMS == 0
        yNorm = y;
        warning('File %s is silent, skipped RMS normalization.', fileName);
    else
        % Scale waveform to target RMS
        yNorm = y * (targetRMS / currentRMS);
    end

    % Save normalized audio
    outputFilePath = fullfile(outputFolder, fileName);
    audiowrite(outputFilePath, yNorm, fs);
    fprintf('Normalized and saved: %s\n', fileName);
end

disp('All files normalized successfully');
```

A.3 Data Slicing Matlab Script

```
[y, fs] = audioread('Bearing_Fail_Rerecorded_Raw.wav');

clipDur = 3; % how long you want the clips to be
samplesPerClip = int32(clipDur * fs);

% used to save the last 25 clips for testing
saveClips = int32(3.125 * 25 * fs);

tempClip = zeros([1,saveClips]);

sourceIndex = 1 + length(y) - saveClips;
clipNumber = 0;

label = "Bearing_Fail_Raw";

% as long as there is enough information in the source file
% to make another clip
while length(y) - sourceIndex - samplesPerClip > 0

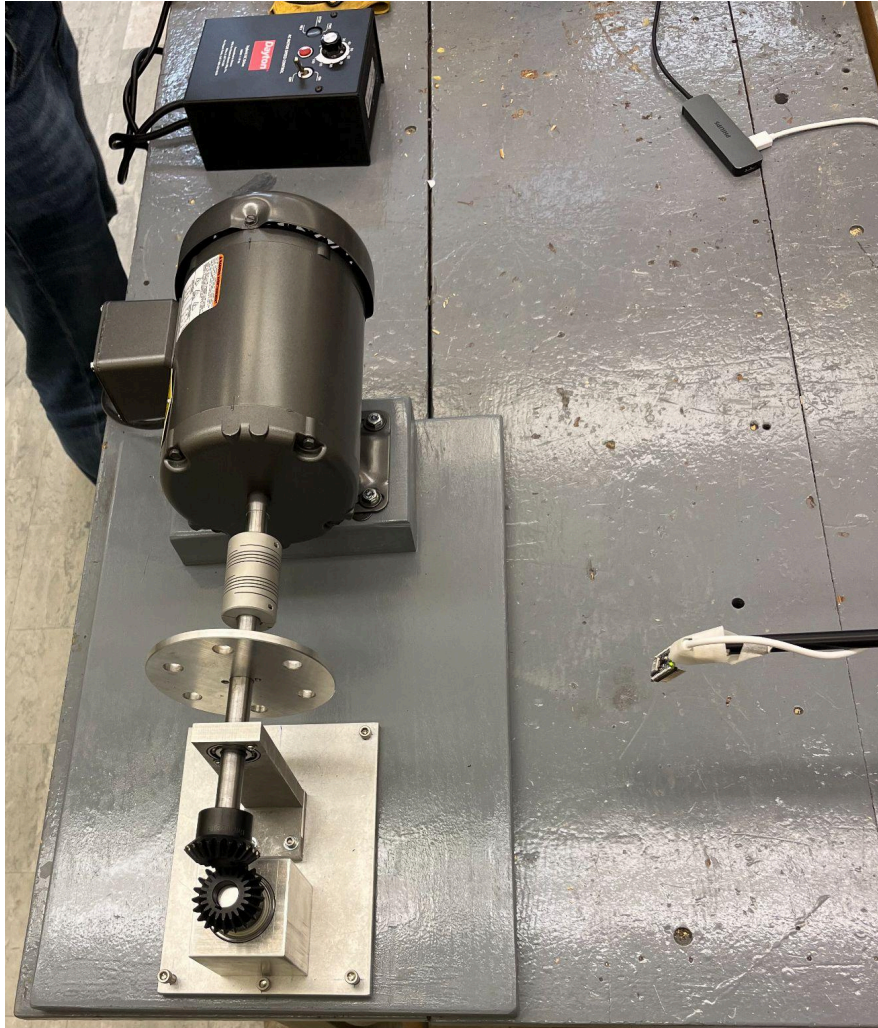
    % copy a clips worth of information to tempClip
    for i = 1:length(tempClip)
        tempClip(i) = y(sourceIndex);
        sourceIndex = sourceIndex + 1;
    end

    % generate file name
    writeFileName = sprintf('%s_%d.wav', label, clipNumber);
    clipNumber = clipNumber + 1;

    % write to that file
    audiowrite(writeFileName, tempClip, fs);

end
```

APPENDIX B: Visual Documentation



Healthy Machine State, Recording Position 1



Healthy Machine State, Recording Position 3

APPENDIX C: Test Bench Parts List

Part #	Part Name	Quantity
M3542 Baldor	General Purpose Motor	1
6512K251 McMaster-Carr	Low Carbon Steel Hex Bar, 11/16" Wide	1 foot of material
8920K175 McMaster-Carr	Low Carbon Steel Rod 5/8" Diameter	2 feet of material
6843K15 McMaster-Carr	1144 Carbon Steel Miter Gear with Set Screw, Keyed Bore	4
60355K178 McMaster-Carr	Steel Ball Bearing for 5/8" Shaft Diameter	2
1291N1 McMaster-Carr	Hex-Bore Ball Bearing for 11/16" Shaft Width	3
6208K639 McMaster-Carr	Clamping Precision Flexible Shaft Coupling	1