Sound Source Localization and Distance Estimation in Open Environment using Simulation and AI

Master Thesis

Denis Rosset

University of Applied Sciences and Arts Western Switzerland

July 2023

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Supervisors:

Michael Mäder: Professor in computer science
Beat Wolf: Professor in computer science

Principals

Marc-Antoine Fénart: Professor in civil engineering
Gabriel Python: Scientific associate at Rosas

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.[**?**]

Contents

1 Introduction

Within the framework of the research project "NPR Teleoperation," the engineers of the HEIA-FR have developed the first concept in Switzerland of a remote-controlled automated vehicle. However, teleoperation only makes sense if the vehicle is automated. There can be no teleoperation without automation (economic factors), just as there can be no automation without teleoperation (legal, technical, and social factors). ROSAS then created the Autovete (Automatisation de véhicules téléopérés) project. HEIA-FR finances them to build up vehicle automation expertise. Detecting other emergency vehicles is mandatory for a vehicle to be fully autonomous. V2V (Vehicle-to-Vehicle) communication is a solution but is not yet integrated into emergency vehicles. So, to detect such a vehicle, two signals need to be processed: the sound of the emergency siren and the blinking lights of the vehicle. The first use case of this project focuses only on sound source distance estimation and localization. Detecting excessively noisy vehicles on the street is a simpler use case created for this project to understand if sound source estimation and localization could work for vehicles in an open environment. The goal is to measure the sound level of the passing vehicles and compare it with the legal limits. If a vehicle exceeds the limit, the system can record its license plate and report it to the authorities. This way, the system can help reduce noise pollution and improve road safety. This system requires a microphone array, a camera, and a processing unit to achieve the needed detection. The microphone array captures the sound signals from different positions and sends them to the processing unit. The processing unit applies a sound source localization algorithm to estimate the direction and distance of the sound source. The camera captures the image of the vehicle and performs license plate recognition.

Big improvements in sound source localization with the help of machine learning are being made. They can be used to reliably localize the origin of a sound using one or more microphone arrays (multiple microphones operating in tandem). A non-negligible problem is the small number of real-world datasets with moving sources in an open environment. A solution is to create datasets in realistic sound propagation simulation and use them to augment the real-world datasets. A model can then be trained on the augmented dataset and tested for its performance in real-world data.

Using machine learning to solve the sound source localization problem can lead to a new attack vector for the system. The system can be attacked by modifying the sound source or the sound signal. Tests to understand how the system reacts to such attacks are necessary to understand the system's robustness.

1.1 Motivation

1.1.1 Objectives

Objective n°1 Definition of a baseline The first objective is to define a baseline for the project. The baseline should contain the problem statement, the project's scope, and objectives. The baseline will help us understand the project's context and the problem we are trying to solve.

Objective n°2 Dataset according to the baseline The first objective is constructing a coherent dataset with the project's baseline. The dataset contains the target variable, features, and necessary pre-processing steps. This dataset will help represent and understand the problem.

Objective n°3 Realising a model for better sound source localization and distance estimation The project uses a neural network model to detect the origin of a sound using a microphone array. The neural network uses the dataset created in objective 2 for training and can localize the sound source accurately. An evaluation of the trained neural network model in a real environment allows us to see how it performs. The model will also be evaluated to understand how it can be improved.

Objective n°4 Attacking the model to understand how it reacts The trained neural network model needs to be evaluated by testing it on data modified in some way, such as by adding or removing noise or modifying the sound source. Tests against various attacks, such as masking, time-warping, and frequency-shifting, will help us understand how it performs.

1.1.2 Challenges

Challenge n°1: Realistic datasets One of the main challenges in this project is to construct realistic datasets that accurately capture the data in a real-world scenario. The lack of open-source datasets that contain moving sound sources in open environments can make this difficult. The project should use realistic simulations of sound propagation to produce suitable datasets.

Challenge n°2: Robust models Another challenge is to create a neural network model that can accurately detect the origin of a sound using a microphone array. The trained model must be accurate and robust to resist adversarial attacks. The model should localize the sound source accurately on altered data.

Challenge n°3: Adequate evaluation metrics The last challenge is to create an evaluation metric that adequately reflects the model's performance. The evaluation metric should consider the accuracy of sound source localization in a real environment and its ability to resist adversarial attacks. The metric should also capture how well the model can provide reliable results in various environments.

1.2 Structure of the thesis

- **Chapter 2: Background and Litterature**: This chapter provides an overview of the background knowledge necessary to understand the project, such as the theory of machine learning and sound propagation, and a brief introduction to sound source localization.
- **Chapter 3: Design**: This chapter describes the design of the different parts of the project. It presents the description of the project's architecture and components.

- **Chapter 4: Realization**: This chapter describes the work done to achieve the project's objectives, such as creating a dataset, creating a simulation, building a neural network model, and evaluating it using an appropriate metric.
- **Chapter 5: Results and Analysis**: This chapter presents the project's results, such as the performance of the neural network model and the evaluation metrics. It also analyzes the results and discusses the findings.
- **Chapter 6: Conclusion and future work**: This chapter concludes this project by discussing the main results and summarizing the key findings.

2
Background and Litterature

This chapter introduces technical concepts and background used in the conceptualized solution of the thesis. It also explains the analysis of the needs of the thesis and finds relations with the current state of research in sound source localization systems.

2.1 Baseline analysis

During the first weeks of the thesis, we had the opportunity to place an installation of microphones on the HEIA-FR main building roof. We took that opportunity to design the baseline and analyze how to build a system around it.

After analyzing the road in front of the HEIA-FR main building, we decided to use the baseline to detect the position of vehicles driving on the road. The road is moderately busy, and the vehicles drive at a reasonable speed. The road is also straight, which makes it easier to detect the position of the vehicles. The baseline is shown in Figure 3.3. This analysis helps provide an intuitive understanding of the sound source localization system. The baseline comprises a vehicle as the sound source we want to record, multiple microphones recording the sound of the street, and an embedded system that manages the microphones.

2.1.1 Audio file format

The Waveform Audio File Format is a standard for storing computer audio data. The audio signal is continuous and is sampled to store it on a computer. The sampling rate represents the number of samples per second. The sampling rate is usually measured in Hertz (Hz). The numerical representation of the audio often used with wav is the PCM.

The PCM (pulse-code modulation) is a method used to represent an analog signal with a binary. It is often used to represent uncompressed digital audio. The PCM is a sequence of amplitude values. Depending on the quality, the amplitude values are often stored as 8-bit, 16-bit, or 24-bit integers. A representation of a sampled sinusoidal signal is shown in Figure 2.1.

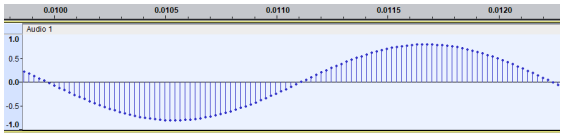


Figure 2.1: PCM representation of a sinusoidal signal

Each vertical line represents a sample. The horizontal axis represents the time, and the vertical axis represents the amplitude. The sampling rate is usually 44100 Hz, 48000 Hz, or 96000 Hz.

2.2 Sound Source Localization

Sound Source Localization (SSL) is the process of determining the position of a sound source. It usually uses a microphone array that captures the sound signals from multiple directions. Various applications use SSL [1], such as speech recognition [2], source separation [3], human-robot interaction [4] or room acoustic analysis [5]. In this thesis, SSL is used to estimate the distance and direction of a sound source to detect excessively noisy vehicles.

2.2.1 Spectrograms for sound visualization

Spectrograms are a visual representation of the frequency content of a sound signal. They are often used in sound source localization to identify the direction of a sound source. The spectrogram is a two-dimensional representation of the frequency content of a sound signal (figure 2.2).

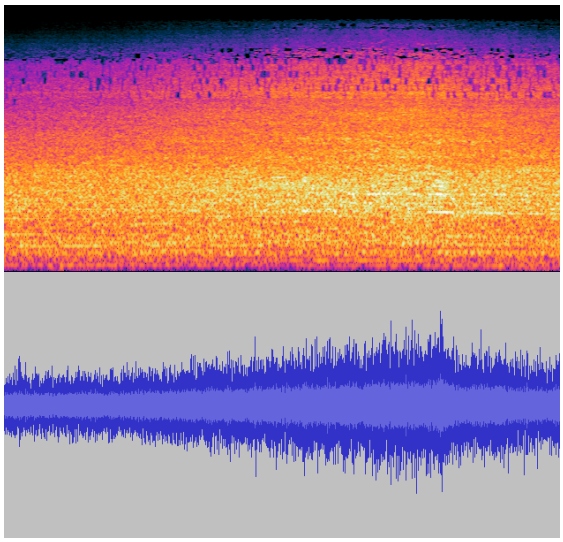


Figure 2.2: Spectrogram of a sound signal

The x-axis represents time, and the y-axis represents frequency. The intensity of the color at each point in the spectrogram represents the amplitude of the frequency component. A matrix of spectrograms allows the representation of multiple channels, such as the ones recorded by a microphone array. On that matrix, each spectrogram represents the frequency content of a single channel (figure 2.3).

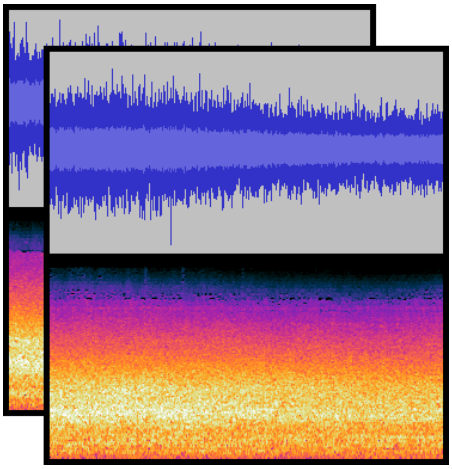


Figure 2.3: Dual channel spectrogram matrix of a sound signal

Looking at the frequency content of the sound signal allows us to identify the time delta of a recorded sound by using a multi-channel spectrogram. The bright spot on the spectrogram will indicate a jump in the amplitude and determine the start time of the recording of a loud sound. By comparing this time with the other channel, we can find the direction of the sound source by comparing the sound signal's time delta with the other channels' time delta (figure 2.4).

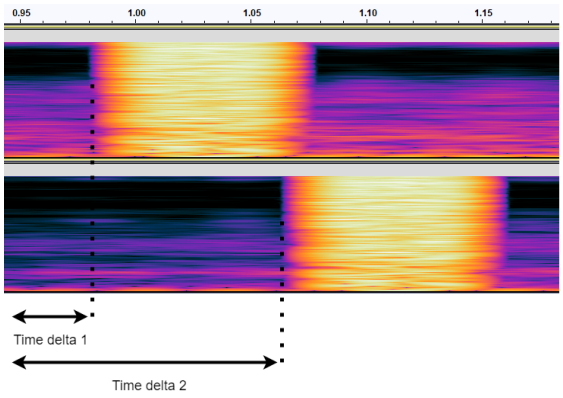


Figure 2.4: Spectrogram of two sound signals with time their delta

Since we know the distance between the microphones, we can determine the direction of the sound.

2.2.2 Origin of sound using two microphones

Admitting the following setup (figure 2.5), if the time delta 1 is greater than the time delta 2 of the other channels (setup 1), the sound source is closer to microphone 2. If the time delta 1 equals the time delta 2 (setup 3), the sound source is at the same distance to both microphones. If the time delta 2 is greater than the time delta 1 (setup 2), the sound source is closer to the microphone 1.

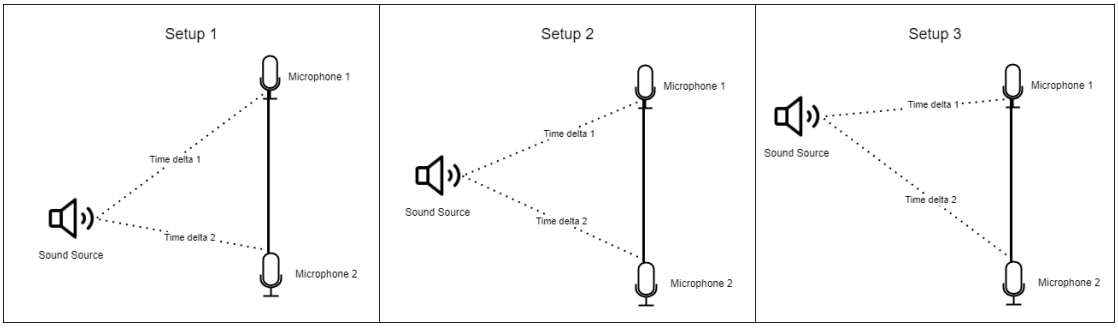


Figure 2.5: Sound source localization setup

This concept can be formalized and is better explained in [6]. Once the delay between the two microphones is known, the equation allows us to find the direction of the sound source by using trigonometric calculations. As in the figure 2.6, considering point M as the sound source and point A and B as microphones, the distance between the two microphones is d and the time delta between the two microphones is Δt, the angle α can be calculated.

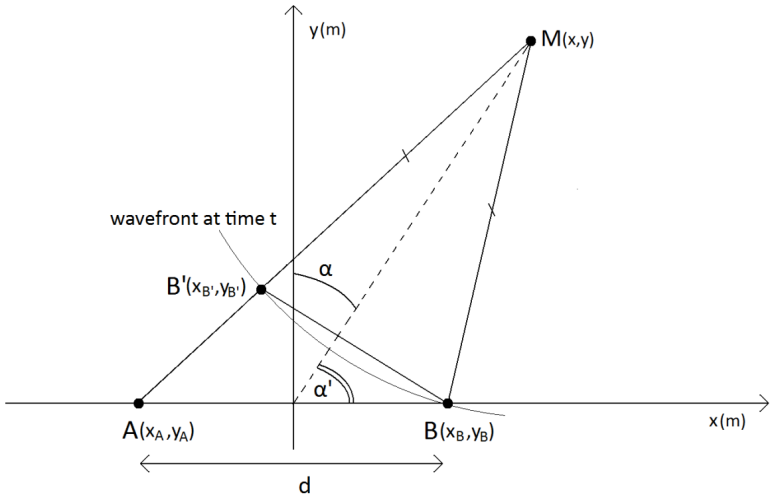


Figure 2.6: Equation formalization. Original image from [6]

Looking at the graphic allows us to find the following equation:

$$AB' = AM - B'M \tag{2.1}$$

With Pythagorean theorem:

$$AM = \sqrt{(X_a - X)^2 + (Y_a - Y)^2} \tag{2.2}$$

$$BM = \sqrt{(X_b - X)^2 + (Y_b - Y)^2} \tag{2.3}$$

The two microphones have the same Y coordinate, so $Y_a = Y_b = Y$ and $Y_a - Y_b = 0$ and $X_a = -X_B$ The equation becomes:

$$y = \pm\sqrt{\frac{AB'^2}{4} - x_B^2 + x^2(\frac{4 \cdot x_B^2}{AB'^2} - 1)} \tag{2.4}$$

This setup shows that two microphones are enough to determine the direction of a sound source.

2.3 Neural networks

Neural networks are machine learning algorithms based on biological neurons used to solve various problems, including image recognition, speech recognition, and natural language processing. Neural networks learn from provided data to solve a problem without explicitly programming the solution. Many domains, like self-driving cars, facial recognition, and medical imaging, achieve state-of-the-art results using neural network models.

A neural network needs to be trained to solve a problem. The training consists of providing the neural network with examples to recognize patterns in the data. Once we finish the training, the model can solve the problem.

A neural network (figure 2.7) is composed of multiple neurons (the circles) that are organized in layers and connected to the neurons in the previous and next layers.

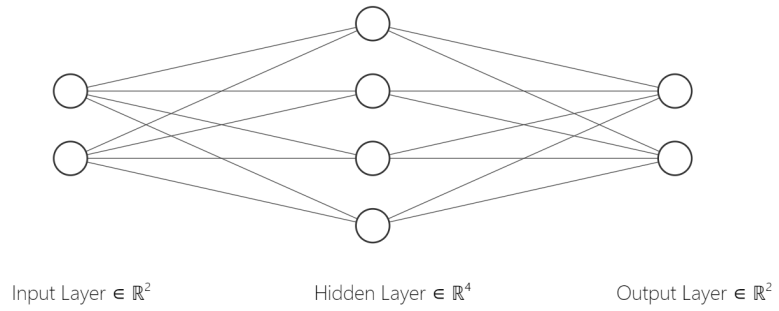


Figure 2.7: Neural network

There are two main phases in the life of a neural network: training and inference. During the training phase, the neural network is trained on a large dataset of images. The neural network learns to recognize patterns in the images and classify them. During the inference phase, the neural network is used to classify new images.

Neural networks comprise multiple neurons. Neurons are mathematical functions with activation functions and weights. These determine how the neurons respond to inputs and connect to other neurons. Neural networks train themselves by adjusting the weights to minimize the error between the predicted and desired outputs. They use an optimization algorithm to adjust the neurons' weights [7]. Multiple optimization algorithms exist, including gradient descent[8] and backpropagation[9]. These algorithms minimize the error between the predicted and desired outputs.

2.3.1 Neuron

A neuron is a mathematical function that takes multiple inputs and produces an output. The activation function and the weights of the neuron determine the output. The activation function determines how the neuron responds to inputs. The weights determine the importance of the inputs. The neuron's output is

calculated by multiplying the inputs by their weights and applying the activation function to the result. The neuron sends its output to the next layer of neurons.

Activation function Multiple activation functions exist for neural networks. The most common activation functions are the linear function, the sigmoid function, the tanh function, and the ReLU function.

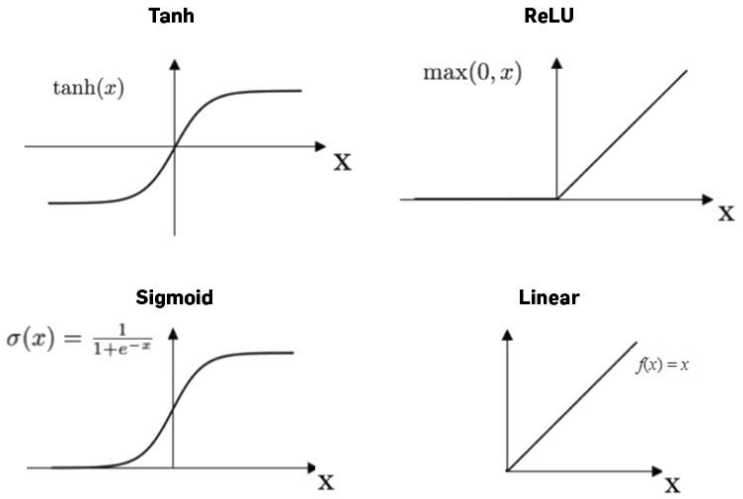


Figure 2.8: Activation functions

These functions will determine how the neurons respond to inputs. These functions have been surveyed in [Activation Functions in Deep Learning: A Comprehensive Survey and Benchmark][10]. It shows that the ReLU function is the most used activation function in deep learning. The ReLU function is defined as:

$$f(x) = max(0, x) \tag{2.5}$$

The ReLU function is used in most neural networks because it is fast to compute and provides good results.

Loss function Neural networks use a loss function to measure the error between the predicted and desired outputs. The loss function is a mathematical function that takes the predicted and desired outputs as inputs and outputs a value representing the error between the predicted and desired outputs. The optimization algorithms use the loss function to adjust the neurons' weights to minimize the error between the predicted and desired outputs. When the loss function's value is low, the neural network accurately predicts the desired outputs.

There is a wide variety of loss functions, including the mean squared error and the cross-entropy.

The mean squared error loss function is defined as follows:

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{2.6}$$

Where y is the desired output, $\hat{y}$ is the predicted output, and n is the number of classes.

The cross-entropy loss function is defined as follows:

$$L(y, \hat{y}) = -\sum_{i=1}^{n} y_i \cdot log(\hat{y}_i) \tag{2.7}$$

Where y is the desired output, $\hat{y}$ is the predicted output, and n is the number of classes.

We can add weights to the loss function to give more importance to some classes. It helps when a dataset is unbalanced by giving more importance to the underrepresented classes. The weighted cross-entropy loss function is defined as follows:

$$L(y, \hat{y}) = -\sum_{i=1}^{n} w_i \cdot y_i \cdot log(\hat{y}_i) \tag{2.8}$$

Where y is the desired output, $\hat{y}$ is the predicted output, n is the number of classes, and w is the weight of the class.

Gradient descent Gradient descent is an optimization algorithm that minimizes the error between the predicted and desired outputs. We use it to train neural networks. Gradient descent works by iteratively adjusting the neurons' weights to minimize the error between the predicted and desired outputs. It uses the gradient of the loss function to find the direction of the steepest descent. The weights are adjusted in the opposite direction of the gradient. The gradient descent algorithm is defined as follows:

$$\theta_{n+1} = \theta_n - \alpha \cdot \nabla f(\theta_n) \tag{2.9}$$

Where θ_n is the current weight, α is the learning rate, and $\nabla f(\theta_n)$ is the gradient of the loss function.

Backpropagation Backpropagation is another optimization algorithm used to train neural networks. Gradient descent forms the basis of the backpropagation algorithm. It uses the gradient of the loss function to find the direction of the steepest descent. The weights are adjusted in the opposite direction of the gradient. Backpropagation is an iterative algorithm that adjusts the neurons' weights to minimize the error between the predicted and desired outputs. The backpropagation algorithm is defined as follows:

$$\theta_{n+1} = \theta_n - \alpha \cdot \nabla f(\theta_n) \tag{2.10}$$

Where θ_n is the current weight, α is the learning rate, and $\nabla f(\theta_n)$ is the gradient of the loss function.

2.3.2 Neural network hyperparameters

Neural networks use different hyperparameters to control the training process. The most common hyperparameters are the activation function and the learning rate.

Learning rate To train efficiently a neural network, we use a learning rate. It determines a factor of how much the weights are adjusted during training. A high learning rate will adjust the weights by a large amount, hence training the neural network faster but less precisely. A low learning rate will adjust the weights by a small amount, hence training the neural network slower but more precisely. The learning rate is a hyperparameter that needs to be tuned to achieve the best results.

A solution to this problem is to use an adaptive learning rate. Adaptive learning rates are learning rates that change during training. They are used to train the neural network faster and more precisely. [Adaptive Learning Rate and Momentum for Training Deep Neural Networks][11] shows us how it can train neural networks efficiently without losing precision.

2.3.3 Deep Neural Networks

Deep neural networks are a type of neural network composed of multiple layers of neurons[12]. They are trained on a large dataset of images and then used to classify new images. There are countless architectures [13] and implementations of neural networks, but they all share the same basic principles. The most known architectures of neural networks include CNNs[14], transformers[15], and many others.

2.3.4 Convolutional Neural Networks for sound source localization

Convolutional Neural Networks (CNNs)[14] are deep neural networks specifically used for image recognition. They often comprise convolutional, subsampling, and fully connected layers (Figure 2.9).

- Convolutional layers are used to extract features from images. These features are then fed into fully connected layers to perform classification. Each convolutional layer comprises multiple filters convolved with the input image to produce a feature map. The model trains the filters to extract specific features from the input image.
- Subsampling layers are used to reduce the size of the feature maps. The most common subsampling layer is the max-pooling layer, which takes the maximum value of a specific region of the feature map.
- Fully connected layers are trained to classify the features extracted by the convolutional layers. The output of the fully connected layers is a probability distribution over the possible classes.

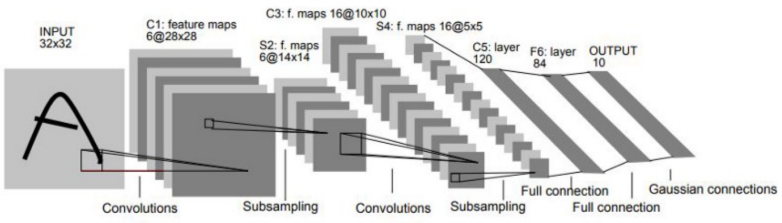


Figure 2.9: CNN architecture example with LeNet-5 [16] composed of two convolutional layers, two subsampling layers, and finishing with two fully connected layers.

Even if CNNs are mainly used to classify real-life photography, they can classify any image, including sounds. The report [A survey of sound source localization with deep learning methods][1] shows that deep neural networks achieve good scores in sound source localization. CNNs can use any image as input. Based on section 2.2.1, we can use the spectrograms as input in the network since we convert the sound into images during the spectrogram process. An approach for sound source localization is to use zones from which the sound can come as classes. The CNN will output a probability distribution over the possible classes. The class with the highest probability is the predicted class. The predicted class can then refer to a zone. The CNN then outputs a probability distribution over the possible classes. The possible classes need to be defined before training the CNN. In[17], they approach the problem with 15 classes, using angles 0, 30, and 60 degrees and distances 1, 2, and 3 meters (Figure 2.10).

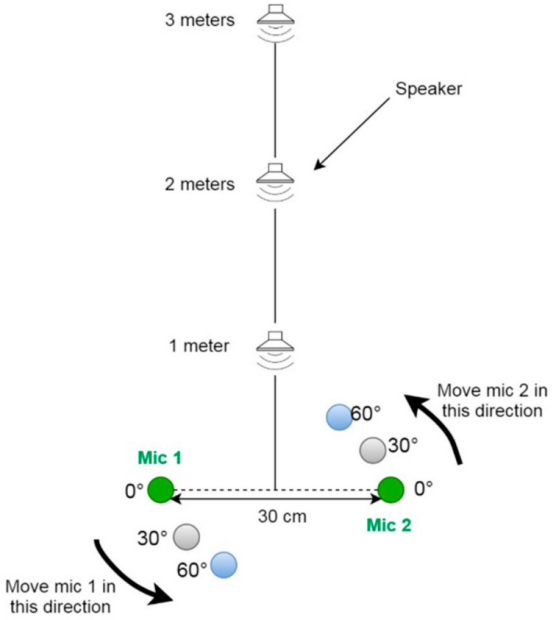


Figure 2.10: CNN for source localization

This setup gives nine classes representing a different zone from where the sound can come from. The CNN will output a probability distribution over the nine classes, which allows determining the zone from where the sound comes from.

2.3.5 Dropout layers

Dropout layers are used to prevent overfitting in neural networks.

2.3.6 Transfer learning

Transfer learning is a machine learning technique where a model trained on a specific task is reused as the starting point for a model on a different task. The model is trained on a large dataset and then used as a starting point for a model on a different dataset. The model is then trained on the new dataset, and the weights are adjusted to minimize the error between the predicted and desired outputs. Transfer learning is used to train models on smaller datasets and achieve better results than training the model from scratch.

2.4 Datasets for machine learning

Datasets are needed to train and test neural networks. They are composed of data and labels. The data is the neural network input, and the label is the expected output. Each entry in the dataset is composed of the data and the corresponding label. An entry in a dataset can also be called a sample.

A balanced dataset contains nearly the same number of samples for each class. An unbalanced dataset contains a considerably different number of samples for each class.

2.4.1 Datasets for sound source localization

In the case of sound source localization, the data is audio, and the labels are the zones of the sound source.

Multiple datasets exist in sound source localization for neural networks. The most common are the DCASE 2019 task 3 dataset[18] and the DCASE 2020 task 3 dataset[19]. These datasets are composed of audio files and the corresponding labels. The labels are the zones of the sound source. The audio files are recorded in a room with a microphone array and a sound source. The sound source is moved around the room, and the audio is recorded. The audio files are then annotated with the zones of the sound source. The annotations are done manually by listening to the audio files and annotating the zones. Multiple annotators then verify the annotations to ensure the quality of the annotations.

Although these datasets are good baselines for sound source localization, they do not suit the needs of this project. The datasets are recorded in a closed environment and do not reflect the baseline defined in this project. Still, these datasets are good baselines for sound source localization and help to understand how to create a dataset.

2.4.2 Dataset augmentation for audio classification

Since recording many audio files is time-consuming and costly, and since the dataset needs to be large to train a neural network, we decided to use dataset augmentation techniques.

Dataset augmentation is a technique used to increase the size of a dataset. It is used to improve the performance of a neural network by training on more data, thus becoming better at generalizing. The most common techniques are flipping, rotating, and cropping images, but since the classification in this project is realized on audio, other techniques are needed to augment the dataset.

Some techniques that work well on audio are adding noise, changing the pitch, or simulating new data.

2.5 Dataset generation using simulation

Simulating a dataset is a technique used to create a dataset without recording data from real life. It helps to create a dataset with many samples. Since the goal is to generate sounds in an open environment, a 3D-capable engine is necessary. Game engines are increasingly popular for simulation tasks since they are optimized for real-time rendering and can simulate complex 3D scenes. The game engine must simulate sound propagation for a realistic audio simulation.

TODO

2.5.1 Metrics review for neural networks

Accuracy The most common metric for neural networks is the accuracy. The accuracy is the number of correct predictions divided by the total number of predictions. Accuracy is a good metric for classification problems since it tells us how the model performs. The accuracy is defined as follows:

$$\text{Accuracy} = \frac{\text{Number of correct predictions}}{\text{Total number of predictions}} \tag{2.11}$$

The accuracy is a useful metric for classification problems as it provides insight into how well the model performs. However, it may not be the best metric for unbalanced datasets. For instance, if a dataset contains 90% of class A and 10% of class B, a model that always predicts class A will have an accuracy of 90%. Although this model has high accuracy, it is ineffective since it always predicts the same class.

Recall The recall is another metric that can be used. The recall is the number of true positives divided by the number of true positives and false negatives. The recall is defined as follows:

$$\text{Recall} = \frac{\text{True positives}}{\text{True positives} + \text{False negatives}} \tag{2.12}$$

The recall is a good metric for unbalanced datasets since it considers the number of false negatives.

F1 score Another metric that can be used is the F1 score. The F1 score is the harmonic mean of the precision and recall. The F1 score is defined as follows:

$$\text{F1 score} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.13}$$

The F1 score is a good metric for unbalanced datasets since it considers precision and recall.

Confusion matrix To have a visual understanding of how a model performs, a confusion matrix can be used. A confusion matrix is a matrix that shows the number of correct and incorrect predictions for each class. The confusion matrix can be used with any number of classes and is a good way to visualize the performance of a model. The main aim of the visualization of a confusion matrix is to see which classes are misclassified and which classes are correctly classified. For example, in a binary classification problem, the confusion matrix is a 2x2 matrix. The confusion matrix is defined as follows:

$$\begin{bmatrix} \text{True positive} & \text{False negative} \\ \text{False positive} & \text{True negative} \end{bmatrix} \tag{2.14}$$

And the matrix can be visualized as follows:

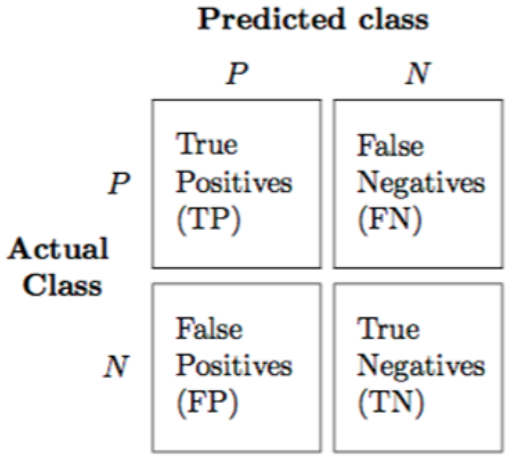


Figure 2.11: Confusion matrix visualization

The confusion matrix allows us to understand which classes are misclassified and to better understand if there are difficulties for the model to predict certain classes.

2.6 Sound propagation

Sound propagation is the physical process by which sound waves propagate in a given environment. Multiple factors affect the propagation of sound waves, including reverberation, occlusion, doppler effect, and obstruction.

The strength of the sound wave depends on various factors, including the frequency, environment, and distance from the sound source. These factors make an accurate identification and localization of a sound source difficult; thus, we need a more accurate and robust sound source localization system.

2.6.1 Microsoft Project Acoustics

Microsoft Project Acoustics is a sound propagation engine that simulates the propagation of sound waves in a given environment. Various applications, including video games, virtual reality, and physics simulation, use this engine. It simulates wave effects like obstruction, reverberation, and occlusion in complex 3D scenes without requiring zone markup or raytracing. It works similarly to a raytracing engine but is precomputed and optimized for real-time performance.

2.6.1.1 Sound Propagation in game-engine

2.7 Adversarial Attacks

Adversarial attacks are a manipulation technique that aims to fool a neural network by modifying the input data. The goal is to make the neural network misclassify. Adversarial attacks allow us to test the robustness of neural networks and understand how neural networks work and how we can improve them.

Adversarial attacks can be categorized into white-box attacks and black-box attacks. White-box attacks are attacks where the attacker has access to the neural network's parameters and architecture. Conversely, black-box attacks are attacks where the attacker cannot access the neural network's parameters and architecture.

One of the most common adversarial attacks is the Fast Gradient Sign Method (FGSM)[20]. It is a white-box attack that uses the gradient of the loss function to find the adversarial example. The adversarial example is calculated using the following equation:

$$X_{adv} = X + \epsilon \cdot sign(\nabla_X J(\theta, X, y)) \tag{2.15}$$

X is the input, y is the target class, ϵ is the magnitude of the perturbation, and $J(\theta, X, y)$ is the loss function. The loss function is the function that the neural network normally tries to minimize, but here is used to maximize the loss. The gradient of the loss function is calculated for the input X. The sign of the gradient is then calculated and multiplied by the magnitude of the perturbation ϵ. The result is added to the input to create the adversarial example X_{adv}. The adversarial example is then fed into the neural network, which outputs the adversarial class (Figure 2.12).

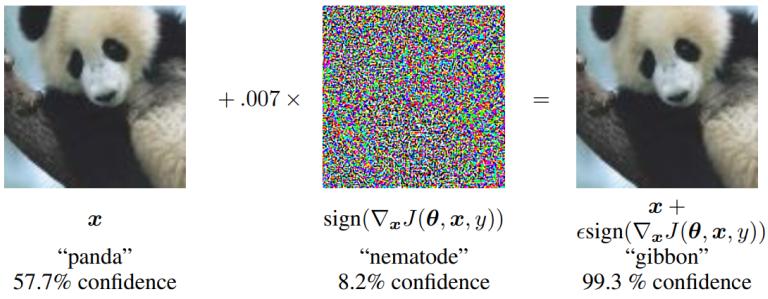


Figure 2.12: FGSM example in [20] with a neural network classifying a panda as a gibbon because of the attack.

2.7.1 Signal estimation from Short Time Fourier Transform

The Griffin-Lim algorithm[21] is an iterative algorithm that uses a spectrogram to estimate the phase of an audio signal. The algorithm starts with a random phase and iteratively updates the phase until the spectrogram converges to the original spectrogram. The algorithm is defined as follows:

3 Design

This chapter presents the design of the project. It is the description of the project's architecture and the project's components.

3.1 Baseline design

We must design a baseline since we start the project without previous work. The baseline is the starting point of the project. It is the simplest system that we can create to solve the problem. The baseline can then compare the results and improve the system.

The baseline system comprises three main parts: the vehicle recordings, the dataset creation, the model training, and the model testing. The system design is shown in Figure 3.1.

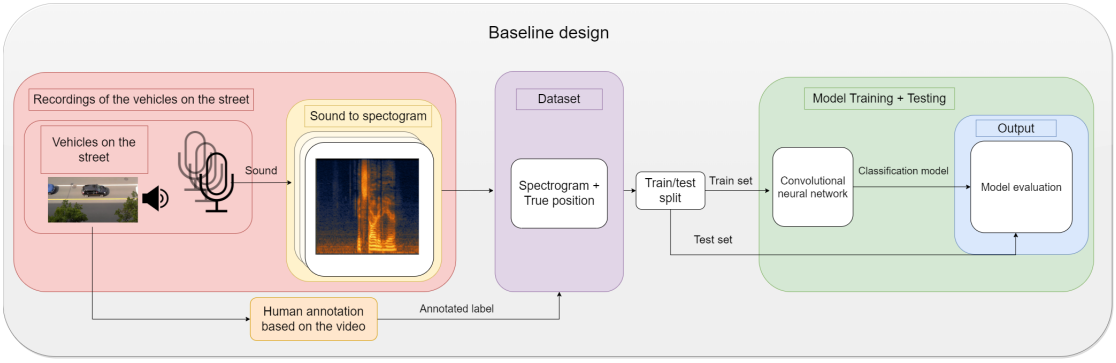


Figure 3.1: Baseline system design

In Figure 3.1, the red zone represents the vehicle recordings with multiple microphones and the transformation of the sound into spectrograms. We also record a video to have a ground truth to annotate the dataset. The purple zone represents the dataset creation with the spectrograms as data and the annotations from humans watching the videos as labels. We then split the dataset into a train and a test set.

In the green zone, we feed the train set into a neural network to train it to predict the position of the sound source based on the spectrograms. We then test the model on the test set to evaluate its performance with unseen data.

Once we train and evaluate the model, we can use it to predict the position of a sound source based on a new recording. The model can be used in inference to detect the position of a sound source. The inference is shown in Figure 3.2.

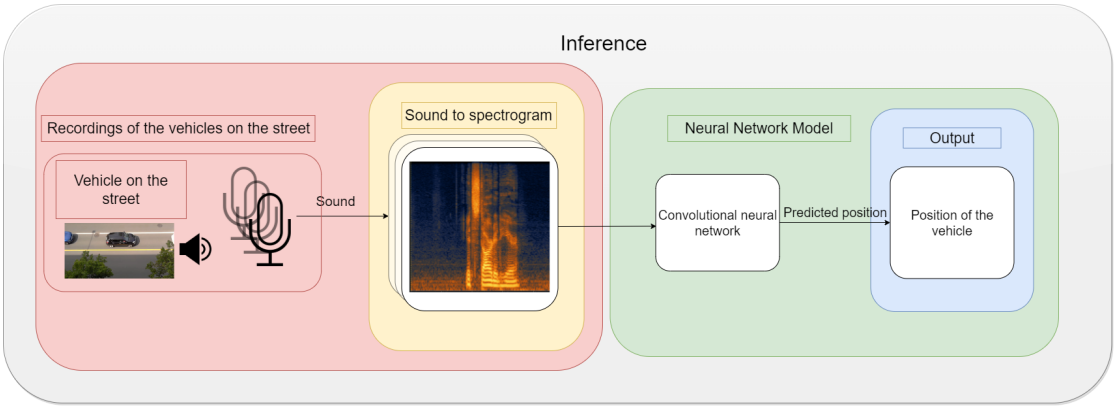


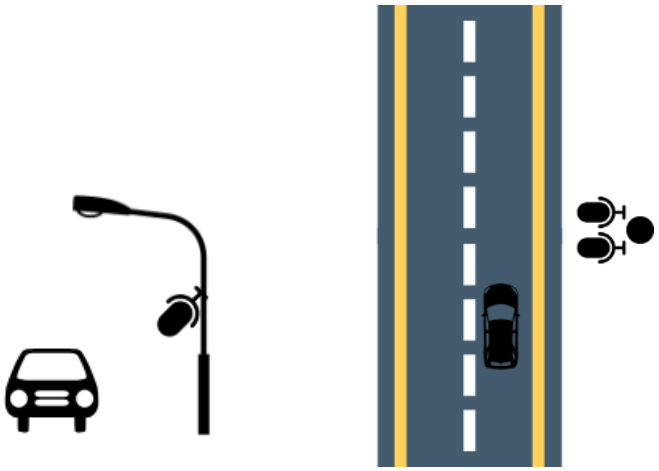
Figure 3.2: Baseline inference

The inference is similar to the training and testing, except that we do not have the ground truth since our model makes the prediction. We only have to do the spectrograms from the recording, and we can feed the spectrograms into the model. The model will then predict the position of the sound source.

We can further develop this idea to incorporate other sound sources for movement tracking in a generalized environment, such as emergency vehicle detection. The baseline is a valuable starting point to develop and test a system that can accurately identify and track sound sources.

3.1.1 Vehicle recordings

To create the dataset, we must have vehicle recordings with multiple microphones. We place de microphones on the side of the street as shown in Figure 3.3.



(a) Side view (b) Top down view

Figure 3.3: Baseline's microphone setup

We also need to record a video of the vehicle to have a ground truth to annotate the dataset. Vehicle recordings are the most crucial part of the baseline. We need to design a system that will allow us to record vehicles from the street and save the data. We designed the system managing the data recording and storage with two microphones, a camera, an embedded system, and a server to store the recordings. This system is shown in Figure 3.4.

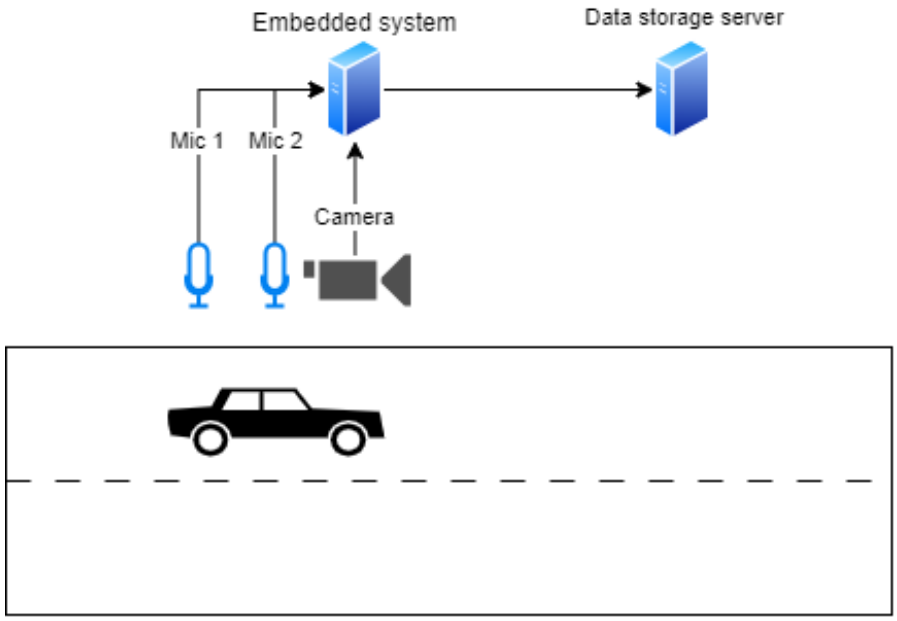


Figure 3.4: Recording system design

We defined two seconds of recording as our sample length. We chose this length arbitrarily, mainly

because it is long enough to have a vehicle passing by the microphones and short enough to have a reasonable size for the dataset.

Overall, the baseline provides a context to develop further a concept of an accurate sound source localization system for outdoor space.

3.1.2 Dataset conception

The dataset is the most crucial part of the baseline. We can determine the dataset's characteristics based on the analysis of the section 2.4.1. The dataset needs to contain the sound recorded by the microphone and the position of the sound source. To simplify the problem, we will use four classes as the main classification challenge in the project. The classes are the following:

- *left_to_right*: The vehicle goes from the left to the right of the microphone.
- *right_to_left*: The vehicle goes from the right to the left of the microphone.
- *no_cars*: No vehicles pass by the microphone.
- *multiple_cars*: Multiple vehicles pass by the microphone.

By adding a camera to the system in section 3.1.1, we can use the image captured by the camera to determine the ground truth of the sound source's position. The camera's position is the same as the microphone's position, and the camera is facing the road. These classes allow the creation of a dataset without precisely recording the vehicle's position. The *no_cars* and *multiple_cars* are here to ensure we will have a complete dataset, as with these four classes, we can cover every possible scenario recorded by the microphones and don't need to cherry-pick only the recordings that match our classification system.

We also used only two classes at the beginning of the project to ensure the concept's functionality when installing the system. These classes are the following:

- *left_to_right*: The vehicle goes from the left to the right of the microphone.
- *right_to_left*: The vehicle goes from the right to the left of the microphone.

3.1.2.1 Recorded data design

The input data needs to be an audio signal. Based on the analysis in section 2.1.1, we use the Waveform Audio File Format with pulse-code modulation to represent our audio signal. With this representation, we obtain a vector of floating point numbers representing the audio signal. Since we record multiple channels simultaneously, we can consider the channels as another vector dimension. We can then represent the audio signal as a matrix of floating point numbers.

3.2 Convolutional Neural Network design for Sound Source Localization

For our baseline, we use a convolutional neural network to predict the position of the sound source. We use a convolutional neural network because, based on the analysis in section 2.3.4, it is the most common neural network architecture for image classification and hence for sound source localization. We can use the spectrograms as image input and the convolutional neural network to classify the spectrograms.

The network design is composed of a feature extraction part and a classifier part. The feature extraction part is composed of convolutional layers, ReLU, and pooling layers. The classifier part is composed of

fully connected layers. The feature extraction part is used to extract the features from the spectrograms, and the classifier part is used to classify the features extracted. The full architecture is shown in Figure 3.5.

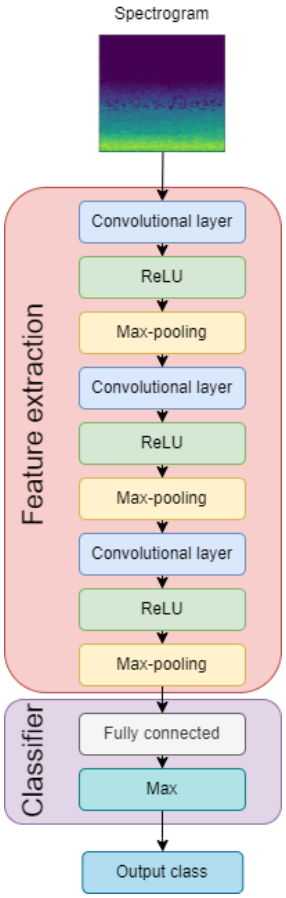


Figure 3.5: Baseline feature extraction

The feature extraction part comprises three blocks of one convolution, one ReLU, and one Max-pooling. The convolutional layers extract the features from the spectrograms. The ReLU layers introduce non-linearity in the network. The pooling layers reduce the dimensionality of the network. The classifier part is composed of one fully connected layer. The fully connected layer classifies the features extracted by the feature extraction part.

3.3 Simulation concept design

To improve the classification score of the baseline, we need to have more data. Multiple possibilities are available to achieve this goal. We can record more data, but it is time-consuming and expensive. We can also use a simulation to generate new data. In this project, we use a simulation to generate new recordings to add to the training dataset to achieve a better classification score on the baseline. The simulation comprises the same elements in the recording system in section 3.1.1 except we simulate them. The simulation design is shown in Figure 3.6.

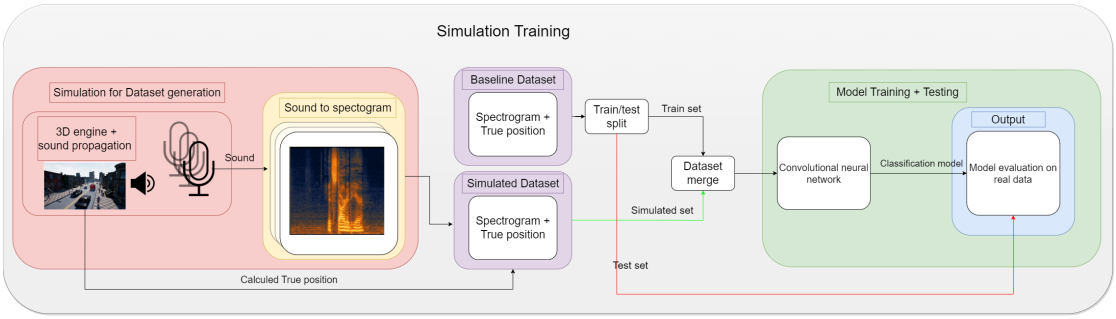


Figure 3.6: Simulation system design for training

There are differences with the baseline system. The main one is that we generate the data in a simulation. The second is that there is no need to annotate the dataset since we know the position of the sound source in the simulation, and we can deduce the position of the sound directly from the simulation. The last one is that we add the dataset generated by the simulation to the trainset of the baseline but not to the test set. This allows us to understand better the simulation's impact on the real data classification score.

The main advantages of the simulation are that we can generate as much data as we want. We can generate data for any position of the sound source. The simulation is composed of a vehicle, a microphone, and a camera. Based on the section 3.1.2, we want to generate data for the classes *left_to_right* and *right_to_left*. We can generate data for the class *left_to_right* by placing the vehicle on the microphone's left and moving it to the right. We can generate data for the class *right_to_left* by placing the vehicle on the right of the microphone and moving it to the left. For *no_cars* class, we can generate data by not placing the vehicle in the simulation. For the *multiple_cars* class, we can generate data by placing multiple vehicles in the simulation and moving them in the same direction.

3.3.1 Generalization aspect of the simulation

For the simulation to best generalize and better represent real-life data, we need to add randomness to the simulation in multiple ways.

Random speed The vehicle's speed is not constant in real life, and we need to add randomness to the vehicle's speed in the simulation. We can add randomness to the vehicle's speed by varying the speed assigned at the beginning of the simulation.

Random path At the beginning of the simulation, we define multiple points as possible start and end points for the vehicle journey. The vehicle's path is generated by randomly choosing a start and end point. We can then generate the vehicle's path by drawing a straight from the start to the end. This method matches the real-life scenario where the straight road in front of the HEIA-FR building constrains the vehicle's path.

Random starting time The vehicle's arrival time is not constant in real life. We add randomness to the vehicle's starting time in the simulation to match the real-life cases. We achieve it by varying the vehicle's waiting time at the simulation's beginning.

Random engine noise We need to assign an engine sound to the vehicle during the simulation for the vehicle to be recorded by the microphone. There are many vehicles in real life and many different engine

noises. We reproduce this by randomly choosing an engine noise at the beginning of the simulation and playing it during the simulation.

Random background noise We add randomness to the background noise by randomly choosing a noise track at the beginning of the simulation and playing it during the simulation.

3.3.2 Simulation software design

The simulation should generate audio data by playing scenarios and recording the sound generated inside it. The comportment should represent the ones analyzed in the section 2.1. Based on the observation, the scenarios chosen were the following:

- **No cars**: No cars are present in the simulation. The background noise is played during the simulation.
- **Left to right**: A car is present in the simulation. The car starts on the left of the microphone and moves to the right. The background noise is played during the simulation.
- **Right to left**: A car is present in the simulation. The car starts on the right of the microphone and moves to the left. The background noise is played during the simulation.
- **Multiple cars**: Multiple cars are present in the simulation. The cars start on the left of the microphone and move to the right. The background noise is played during the simulation.

3.4 Adversarial Attack design

Based on the analysis in section 2.7, we can use an adversarial attack to fool the model designed in section 3.2. This project uses the Fast Gradient Sign Method (FGSM) to generate adversarial inputs. The FGSM is a white-box attack meaning that we need the model's parameters to generate the adversarial inputs. Once we finish training the model, we can calculate the loss function's gradient concerning the input to find the direction that maximizes the loss function. Once we find the direction, we can add a perturbation multiplied by a value of epsilon to the input to generate the adversarial input. Once we generate adversarial inputs, we test them on the model to see how it reacts. We realize the attack on a specific value of epsilon. We can try to attack the model with different epsilon values to see how much noise is needed for the model to fail. We analyzed the results by comparing them with the baseline model's results.

3.4.1 Audio reconstruction design

Realizing the adversarial attack by modifying the spectrogram input is not enough to have a negative impact if we use the model for inference in a real-life scenario. Since the system designed in section 3.1 uses microphones as input, we need to transform the adversarial spectrogram back to audio. We can then play the audio on a speaker in front of the baseline system to see if the model still fails after the reconstruction of the audio signal.

3.4.2 Adversarial attack protection design

Once we generate the adversarial audio signal, we must again play it through a speaker and the microphone. We can analyze the adversarial audio signal before the classification and try detecting an attempted attack. We can use a classic CNN to classify the type of sound recorded. We can then use the classification score to detect an attempted attack.

4 Realization

This chapter explains how we realized every part of our project. We present the results for each part of this chapter in the chapter 5.

4.1 Realization of the data recording system

In this section, we explain how we realized the data recording system. We first explain the hardware we use and how we install it. Then, we explain how we record the data and process it to get the needed data. Finally, we explain how we use the data to train our model.

4.1.1 Hardware for the recordings

To record real data that suits our baseline, we must design a system to record and save lots of data. As we had the opportunity to place it on the HEIA-FR, we decided to design a system containing an embedded system, two microphones, a camera, and an embedded system. For the hardware, we chose to ensure that the system is easily replicable and that the system is not too expensive. We use hardware available at the HEIA and Rosas. The global architecture of the system is shown in Figure 4.1. This diagram helps us understand how each system component is connected and how to access them.

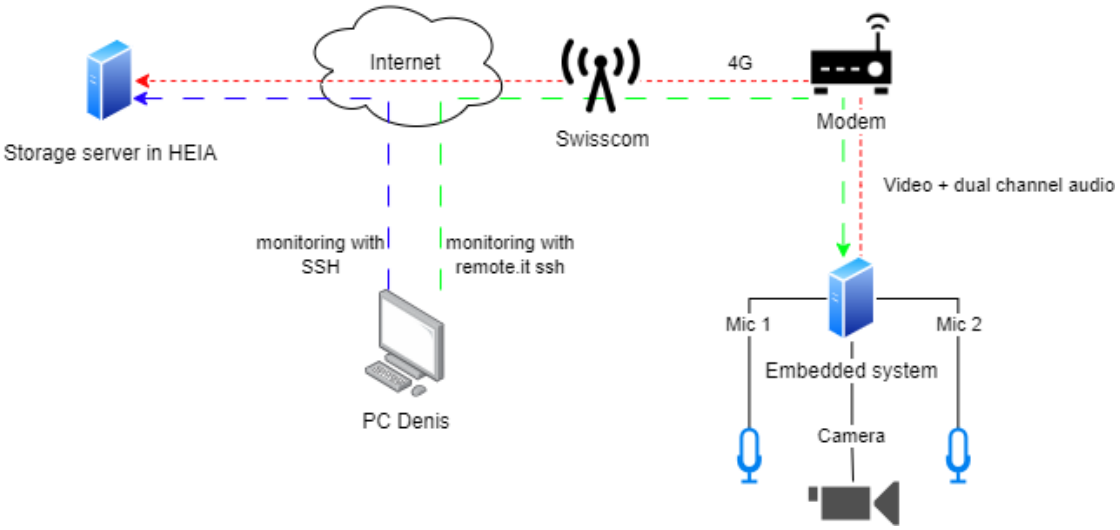


Figure 4.1: Global architecture

The HEIA-FR has a small balcony on the 4th floor with hardware installed for other projects. We use this balcony to install our system. The balcony is located on the side of a street and has a barrier on which we can attach hardware to have a good view of the street.

To record real data that suits our baseline, we must get hardware to record and save the data. The hardware we use is the following:

Microphones We use two microphones to record the sound. We use the *nsrt mk3 dev kit* from *convergenceinstruments* [1] with the USB audio interface (Figure 4.2). Prof. Marc-Antoine Fénart chose the microphones himself. We use two microphones to have a stereo sound recording.

Figure 4.2: Microphone used for the recordings

Camera The camera used is a webcam from Rosas that was available at the moment of the installation. We use the *C310 webcam* from *logitech* [2] since it meets our needs (Figure 4.3).

[1] https://convergenceinstruments.com/
[2] https://www.logitech.fr/fr-fr/product/hd-pro-webcam-c920

Figure 4.3: Webcam used for the recordings

Embedded system We use the *Raspberry Pi 4* from *raspberrypi* [3] as an embedded system (Figure 4.4). We use this embedded system because it is powerful enough to run the recordings and was available at the HEIA-FR. Since our microphones and our camera have USB connectors, we needed an embedded system with at least 3 USB connectors.

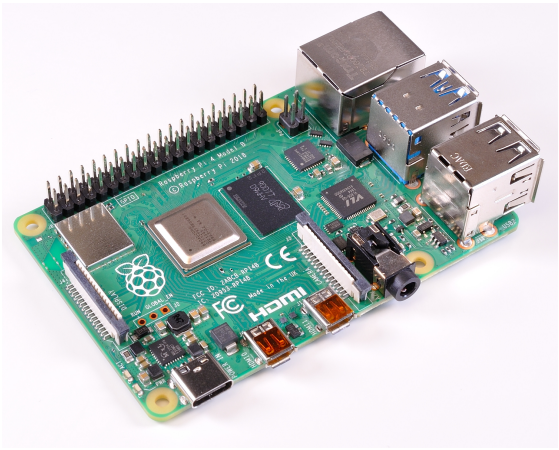


Figure 4.4: Raspberry Pi 4 used for the recordings

Storage We asked the HEIA-FR for a storage server in the school to upload the data. They lend us a server with two terabyte of storage accessible from the school network and from the internet.

[3] `https://www.raspberrypi.org/products/raspberry-pi-4-model-b/`

Data transmission Since there is no network cable on the HEIA-FR's balcony, we use a 4G modem lent by the HEIA-FR to transmit the data to the storage server. We use a 4G LTE N300 router from D-Link [4] (Figure 4.5) to transmit the data. We use a SIM card from Swisscom, also lent by the school to transmit the recordings via the Internet.

Figure 4.5: D-Link router used for the recordings

3D support To attach the microphones, we design 3D pieces with CAD software to 3D print them. We use *tinkercad* to design the pieces. After the design, we give the 3D models to the mechanics at ROSAS to 3D print them (Figure 4.6).

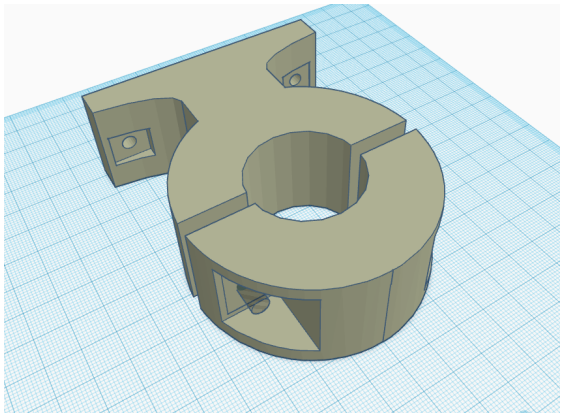


Figure 4.6: 3D pieces

We can see the installed hardware in Figure TODO Ajouter image ! + mesure de la distance entre les micros

[4] https://eu.dlink.com/

4.1.2 Software for the recordings

To record the data, we must have software to control the hardware. We must also have software to transmit the recorded data to the storage server. This subsection describes the software used to make the system work.

Operating system We used *Raspberry Pi OS* [5] as the operating system for the Raspberry Pi because it is the official operating system for the Raspberry Pi, and it also is the most used operating system for the Raspberry Pi. It is based on *Debian* [6] and is optimized for the Raspberry Pi. It is also easy to install and use.

Audio recording Since each microphone has its sound card, we must treat them as one sound card per microphone on the operating system. This can cause some issues if we start the recording at a slightly different time. This problem can, for example, be the case when we execute two consecutive commands, one after the other, in a script to start the microphone recording. We can use *Advanced Linux Sound Architecture (ALSA)* [7] to manage each sound interface and combine them into one virtual sound interface to be able to start the record from both sound cards at the same time. This can be done in a configuration file for ALSA. We used the following configuration file to combine the two sound cards into one virtual sound card:

```
pcm.mic1 {
  type hw
  card NSRTmk3Dev
  device 0
}

pcm.mic2 {
  type hw
  card NSRTmk3Dev_1
  device 0
}

pcm.mic12 {
  type multi
  slaves.a.pcm mic1
  slaves.a.channels 1
  slaves.b.pcm mic2
  slaves.b.channels 1
  bindings.0.slave a
  bindings.0.channel 0
  bindings.1.slave b
  bindings.1.channel 0
}
```

This configuration file will create a virtual sound card called mic12 that will combine the two sound cards: *mic1* and *mic2*. We can then use this virtual sound card to start the recording on both sound cards simultaneously.

Video recording and synchronization We use *ffmpeg* [8] to record the video and the audio synchronously. The command used to record the video and the audio is the following:

[5] https://www.raspberrypi.org/software/operating-systems/
[6] https://www.debian.org/
[7] https://www.alsa-project.org/
[8] https://www.ffmpeg.org/

```
ffmpeg -f alsa -thread_queue_size 2048 -i plug:mic12 -f v4l2 -thread_queue_size
↪ 2048 -input_format mjpeg -video_size 600x400 -i /dev/video0 -c:a aac -map 0:a -
↪ map 1:v -segment_time 00:10:00 -f segment /mnt/videos/$current_date/output%05d.
↪ mp4
```

This command combines the video from the device */dev/video0* and the virtual sound card *mic12* in a single file containing audio and video.

Before installing the system on the HEIA-FR balcony, we calculated the delay between the video and the audio by clapping in front of the webcam and the microphones. Since the delay found was inferior to 100 milliseconds and we only recorded two seconds of video to provide ground truth, we admitted it was negligible for the dataset annotation task.

Raspberry Pi access for administration We used a server from the HEIA-FR to store the data. We can access the server through OpenSSH on the local network of the HEIA-FR.

Since we don't want to go to the HEIA-FR every time we want to access the Raspberry Pi, and we don't want to have ports open on the internet, we use *remote.it* [9] to access the Raspberry Pi remotely. *Remote.it* is a service that allows us to access the Raspberry Pi remotely without opening ports on the internet. It creates a VPN between the Raspberry Pi and the *remote.it* server and gives us the VPN address on the *remote.it* web application. We can then access the Raspberry Pi through the VPN.

Data transmission Since the data we transmit could be sensitive, we transferred it using a secure file transfer protocol. SFTP is a network protocol that provides file access, file transfer, and file management functionalities over a secure channel. We used *OpenSSH* [10] to transfer the data. OpenSSH is a suite of secure networking utilities based on the Secure Shell (SSH) protocol. OpenSSH encrypts all traffic (including passwords) to eliminate eavesdropping, connection hijacking, and other attacks. To transfer the data, we mounted the storage server as a local drive on the Raspberry Pi by using the following command:

```
sshfs drosset@proxy51.rt3.io:/home/drosset/workspace/videos /mnt/videos -p 33838
```

This command will ask for a password to connect to the server and provide us with a local drive on the Raspberry Pi that is connected to the server. We can then use this local drive to store the data directly on the storage server.

4.2 Dataset creation

For most of the dataset management, we used Python. Python is a programming language that gives us many dataset management tools. We used Python to split the recordings into smaller files, annotate the dataset, and manage the folders.

Once we set up the hardware and the software allows us to record the vehicles on the street, we can build a dataset. The recordings of the vehicles contain audio and video in mp4 files of ten minutes each. We must split the recordings into smaller files to get to the two seconds of length defined in section 3.1.1. We split the recordings into two seconds files. Since splitting a video can be time-consuming, we launch it in a subprocess to execute it concurrently on multiple files at the same time. We used the following script:

```
for file in files:
    subprocess.call(['ffmpeg', '-i', directory + '/' + file, '-c:v', 'libx264', '-crf',
    ↪  '22', '-map', '0', '-segment_time', time, '-reset_timestamps', '1', '-g', '30',
    ↪  '-sc_threshold', '0', '-force_key_frames', 'expr:gte(t,n_forced*'+str(time)+')'
    ↪ , '-f', 'segment', directory[:-1]+'-2sec/' + file + '%05d.mp4'])
```

[9] https://remote.it/
[10] https://www.openssh.com/

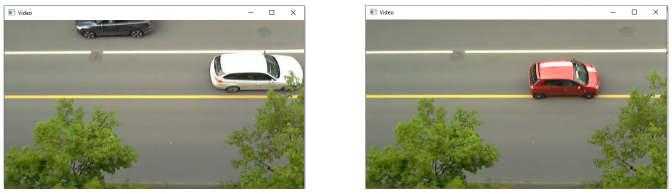
This script allows to have two seconds of video clips of the vehicles. We can then annotate the dataset.

4.2.1 Dataset annotation

A good practice when creating a dataset to classify it is to put the files in folders. Each folder represents a class. In our case, we use the classes defined in section 3.1.2:

- *left_to_right*: Key: **D** The vehicle goes from the left to the right of the microphone.
- *right_to_left*: Key: **A** The vehicle goes from the right to the left of the microphone.
- *no_cars*: Key: **S** No vehicles pass by the microphone.
- *multiple_cars*: Key: **W** Multiple vehicles pass by the microphone.

Each class has its folder. We can then annotate the dataset by moving the files to the correct folder. We developed a tool to annotate the dataset. The tool is an application that shows a 2-second video from the recordings. The user can then press a key in the application to automatically move the video to the class folder. The user can also press a ”cancel” key to remove the last annotated video if the user makes a mistake. Figure 4.7 (a) shows an example of the tool. We can see multiple vehicles in this video. The user can press the **W** key. In Figure 4.7 (b), we see only one vehicle going from right to left. The user can then press the **D** key to save the file in the correct folder.



(a) Multiple car (b) Left to right

Figure 4.7: Dataset annotation tool

Since the videos only show vehicles moving on a road, we don’t need to play the video at real speed. The tool plays the video accelerated to gain time when annotating the dataset.

4.2.2 Dataset

The annotation was done for 2037 videos and defined our baseline dataset for the project. The dataset contains 2037 videos of two seconds each. We split the dataset into four classes. The statistics for the classes are the following:

```
classes:  ['left_to_right', 'multiple_cars', 'no_car', 'right_to_left']
total files:  2037
total files for class left_to_right:  513
total files for class multiple_cars:  234
total files for class no_car:  582
total files for class right_to_left:  708
```

We represent the statistics as a table in Table 4.1.

Class	Number of files
left_to_right	513
multiple_cars	234
no_car	582
right_to_left	708

Table 4.1: Dataset statistics

We made a script to shuffle the dataset and split it into training and test sets. The script's parameter defines the proportion of data in the train and test set. We use the training set to train the neural network and the test set to evaluate the neural network. The script output the number of files in each set. For example, when we run the script with a 70% train and a 30% test, the script gives us the following output:

```
classes:  ['left_to_right', 'multiple_cars', 'no_car', 'right_to_left']
files in train:  1426
files in test:  611
total files:  2037
ratio:  0.29995090819833087
```

The script calculates the ratio to ensure that the proportion of data in the train and test set is correct.

4.2.3 Dataset annotation from audio

Since the project aims to use audio to annotate the dataset, we also create a dataset from the audio. We can use the same tool to annotate the dataset. The tool plays the audio of the video instead of the video. The user can then press the same keys to annotate the dataset. Since we use two microphones, we can play the audio in a stereo headset to have all audio channels listenable during the annotation. We can then compare the results of the dataset annotated from the audio only with the results of the neural network trained on the dataset annotated from the video.

4.3 Neural Network for Sound Source Localization

For the neural network implementation, we use Python. Python is a popular programming language for machine learning. It is easy to use and has a lot of libraries for machine learning. The machine learning library we use is *PyTorch* [11]. We use it because it is popular, open-source, and free. It also has a lot of documentation. Pytorch allows us to create neural networks in Python. It helps us create the neural network architecture, load the data, train the neural network, and test it. It uses the concept of tensor to represent the data. A tensor is a multidimensional array. To manage the arrays on the Python side, we use the *numpy* [12] library. We also use the *matplotlib* [13] library to plot the results.

The shape of a tensor is a representation of the size of each of their dimension and their sizes.

4.3.1 Data loading

Once we have annotated the dataset, we can load the train set in as a matrix. The matrix contains a dimension for each sample, a dimension for each channel, and a dimension for each time step. In parallel, we keep an array with every label for each sample. We then convert the matrix to a tensor with shape ([1422, 2, 88200]). The first dimension is the number of samples in the train set (here, 1418). The second

[11] https://pytorch.org/
[12] https://numpy.org/
[13] https://matplotlib.org/

dimension is the number of channels (here, two since we recorded with two microphones). The third dimension is the number of samples (here 88200, which is two times the used sampling rate since we recorded for two seconds).

4.3.2 Data preparation

Since we want to input spectrograms into our network, we need to convert the audio samples to spectrograms. We use the *torchaudio* [14] spectrogram implementation to convert each sample to a spectrogram. The spectrogram function has the following parameters:

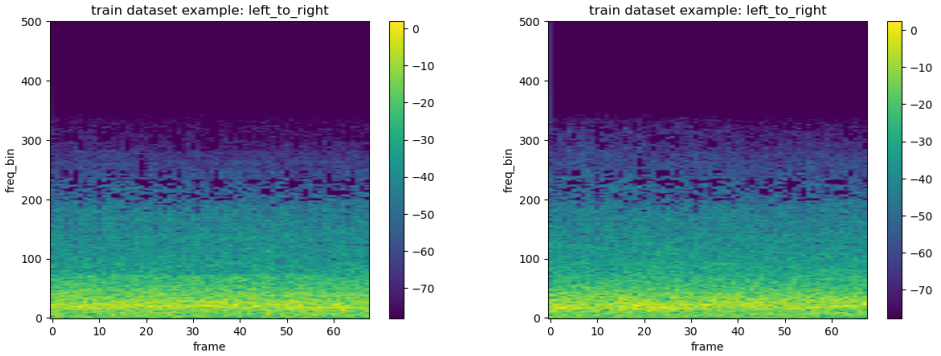
- **n_fft**: the number of Fourier bins. We use 1000 bins.
- **win_length**: the length of the window to use to compute the spectrogram. We use a window of 1000 samples.
- **hop_length**: the length of the hop to use to compute the spectrogram. We use a hop of 1280 samples.
- **window**: the window to compute the spectrogram. We use a Hann window of 1000 samples.
- **normalized**: whether to normalize the spectrogram. We use a normalized spectrogram.

The spectrogram function returns a tensor with the shape [1422, 2, 501, 69]. The first two dimensions are the number of samples and channels. The third dimension is the number of frequency bins (here, 501). The fourth dimension is the number of time steps (here, 68). These parameters are the result of the spectrogram function parameters. The number of frequency bins is the number of Fourier bins divided by two plus one. The number of time steps is the number of samples minus the window length divided by the hop length plus one.

$$\text{number of frequency bins} = \frac{\text{number of Fourier bins}}{2} + 1 = \frac{1000}{2} + 1 = 501 \tag{4.1}$$

$$\text{number of time steps} = \frac{\text{number of samples} - \text{window length}}{\text{hop length}} + 1 = \frac{88200 - 1000}{1280} + 1 = 69 \tag{4.2}$$

After the spectrogram process, we can visualize some spectrograms. Figure 4.8 shows some spectrograms from the train set.



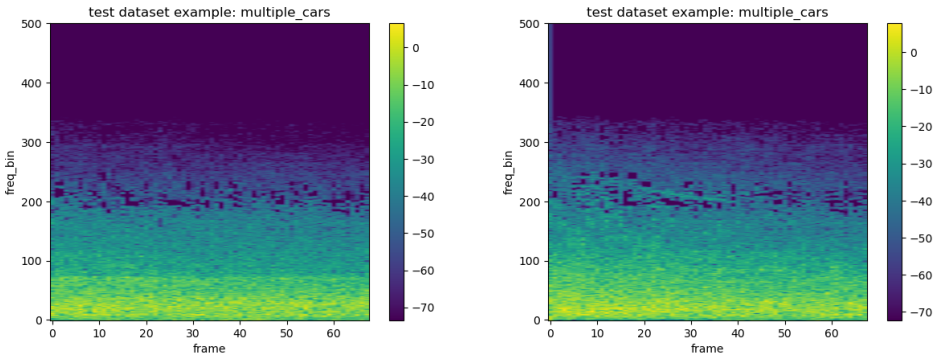
(a) Channel 0

(b) Channel 1

Figure 4.8: Spectrograms for each channel from the train set

[14] https://pytorch.org/audio/stable/generated/torchaudio.transforms.Spectrogram.html

And Figure 4.9 shows some spectrograms from the test set.



(a) Channel 0 (b) Channel 1

Figure 4.9: Spectrograms for each channel from the test set

It's hard to tell the difference between the spectrograms even when they are in two different classes. That's why we use a neural network to classify them.

4.3.3 Convolutional Neural Network architecture

We use the architecture defined in section 3.2. Based on the input dimension, it gives the architecture in Figure 4.11. We generate this by using the *sumarry* function of library *torchsumarry*[15]. The *sumarry* function takes a neural network as input and returns the architecture of the network.

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
├─Sequential: 1-1                        [-1, 8, 100, 13]          --
│    └─Conv2d: 2-1                       [-1, 8, 501, 68]          408
│    └─ReLU: 2-2                         [-1, 8, 501, 68]          --
│    └─MaxPool2d: 2-3                    [-1, 8, 100, 13]          --
├─Sequential: 1-2                        [-1, 16, 33, 4]           --
│    └─Conv2d: 2-4                       [-1, 16, 100, 13]         3,216
│    └─ReLU: 2-5                         [-1, 16, 100, 13]         --
│    └─MaxPool2d: 2-6                    [-1, 16, 33, 4]           --
├─Sequential: 1-3                        [-1, 32, 16, 2]           --
│    └─Conv2d: 2-7                       [-1, 32, 33, 4]           12,832
│    └─ReLU: 2-8                         [-1, 32, 33, 4]           --
│    └─MaxPool2d: 2-9                    [-1, 32, 16, 2]           --
├─Linear: 1-4                            [-1, 4]                   4,100
==========================================================================================
Total params: 20,556
Trainable params: 20,556
Non-trainable params: 0
Total mult-adds (M): 19.50
==========================================================================================
Input size (MB): 0.26
Forward/backward pass size (MB): 2.27
Params size (MB): 0.08
Estimated Total Size (MB): 2.61
==========================================================================================
```

Figure 4.10: Convolutional Neural Network architecture from Torchsumarry

We can better visualize the network architecture with Figure 4.11. It shows us the different layers that we apply to the network.

[15] https://pypi.org/project/torchsummary/

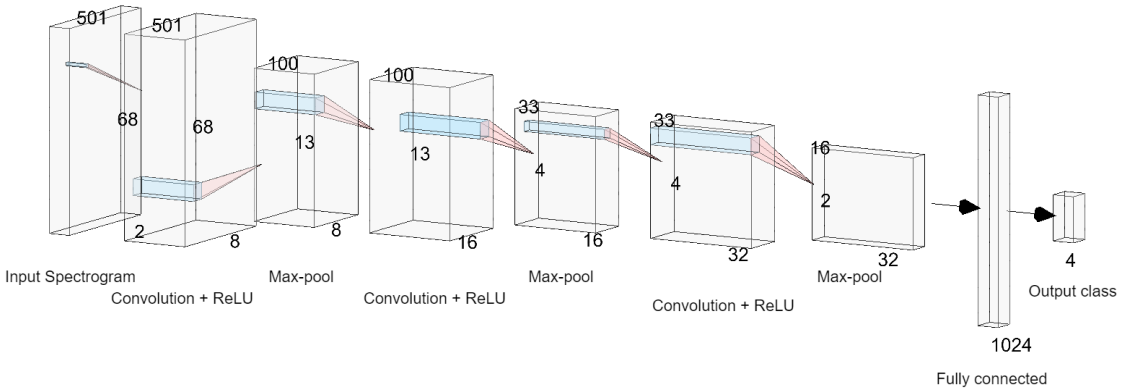


Figure 4.11: Convolutional Neural Network architecture

We added a dropout layer between the last layer and the output. It will help to prevent overfitting.

Our dataset is unbalanced, so we use the weighted cross-entropy loss function. We use the pytorch implementation of the weighted cross-entropy loss function[16].

We use the Adam optimizer with a learning rate of 0.001. We use the pytorch implementation of the Adam optimizer[17].

We also use a scheduler to reduce the learning rate when the loss function stops decreasing. We use the pytorch implementation of the ReduceLROnPlateau scheduler[18]. This will change the rate at which the optimizer will change the weights of the neural network.

4.3.4 Training

When training, we have to define some more parameters for the training. We use the following parameters:

- **batch_size**: the number of samples to use in one batch. We use a batch size of 32.
- **patience**: the number of epochs to wait before reducing the learning rate. We use a patience of 3 epochs.
- **weight_decay**: the weight decay. We use a weight decay of 1e-3.

We don't use the epochs count to stop the training. We use the ReduceLROnPlateau scheduler to stop the training when the learning rate is less than 1e-7. Which means that the model is nearly not learning anymore.

4.3.5 Training hardware

We used a computer with the following specifications to train the neural network:

- **CPU**: Intel Core i7-10700K
- **GPU**: NVIDIA GeForce RTX 4070 Ti

[16] `https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html`
[17] `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`
[18] `https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html`

- **RAM**: 32 GB

The important part of the hardware is the GPU. Since machine learning comprises many parallelizable operations, it's much faster on GPU. We use CUDA[19] to use the GPU. CUDA is a parallel computing platform and application programming interface model created by Nvidia. It allows us to use the GPU to train the neural network. The Pytorch library does the CUDA integration. To use the GPU, we have to tell Pytorch to move the model and the data on the GPU by using the *.to('cuda')* function.

This hardware helps us to train the neural network faster. We can train the neural network in 2 minutes on this hardware. If we don't use the GPU, the training time is multiplied by 40. The short training time allows us to train the neural network multiple times and to try different parameters.

Once we train the network, we can use it to classify the data by giving new input data to the network. After each training, we save the model to be able to use it later.

4.3.6 Training visualization

We use *Tensorboard*[20] to visualize the training. Tensorboard is a visualization toolkit for machine learning experimentation. Figure 4.12 shows the Tensorboard interface.

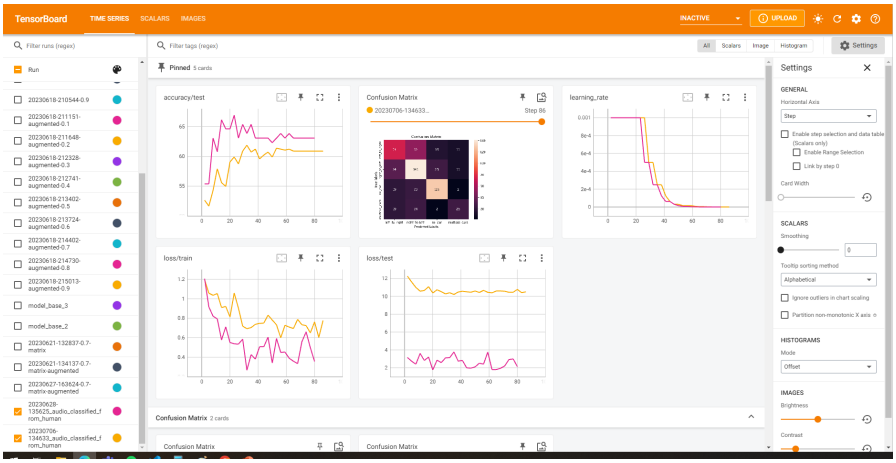


Figure 4.12: Tensorboard interface

It allows us to visualize the training loss and accuracy during the learning process. It helps to understand if we choose good hyperparameters for the training or if the neural network is learning. It also helps to compare the results between training.

4.4 Simulation model creation

To create a dataset in simulation, we use Unity combined with the *Microsoft Project Acoustic*[21] plugin. The combination of the two allows us to create a simulation of a street with vehicles passing by and to record the sound of the vehicles. We can then use the recorded sound to create a dataset. We can then use the dataset to train a neural network.

[19] https://developer.nvidia.com/cuda-toolkit
[20] `https://www.tensorflow.org/tensorboard`
[21] https://learn.microsoft.com/en-us/gaming/acoustics/what-is-acoustics

The simulation is also faster than recording the data from real life. We can generate a lot of data in a short time. The simulation is also cheaper than data recording since we only need a computer. The simulation is also more flexible than recording the data. We can change the position and comportment of every object in the simulation at any time.

4.4.1 Unity

We reproduced a street in Unity. We used the Unity Asset Store[22] to get the assets to create the street. Figure 4.13 shows the street in Unity.

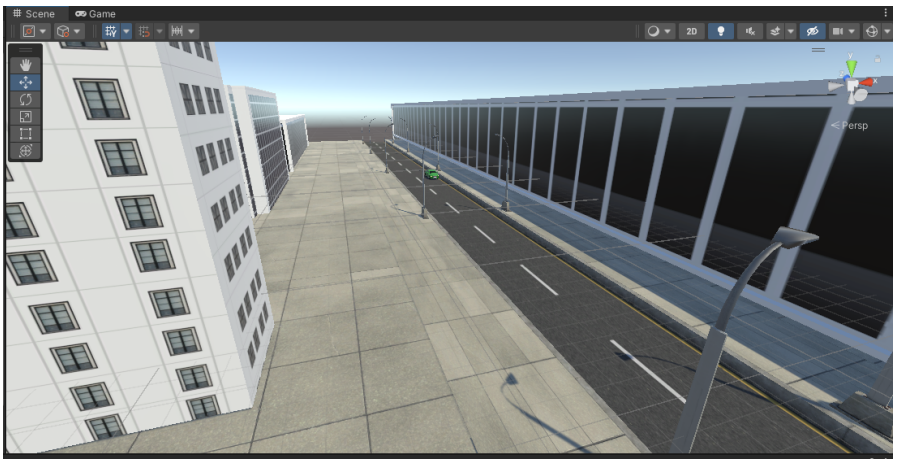


Figure 4.13: Street in Unity

4.4.2 Microsoft Project Acoustic plugin

The Microsoft Project Acoustic plugin allows us to simulate the sound propagation in the scene. We can add it in the Unity's plugin tab by following Microsoft's guide[23]. Once it's added, we need to voxelize the scene to define which 3D surface the sound will bounce. The plugin does the voxelization and takes a short time to complete. Figure 4.14 shows the voxelization of the scene. Each green square represents a voxel.

[22] `https://assetstore.unity.com/`
[23] https://learn.microsoft.com/en-us/gaming/acoustics/unity-integration

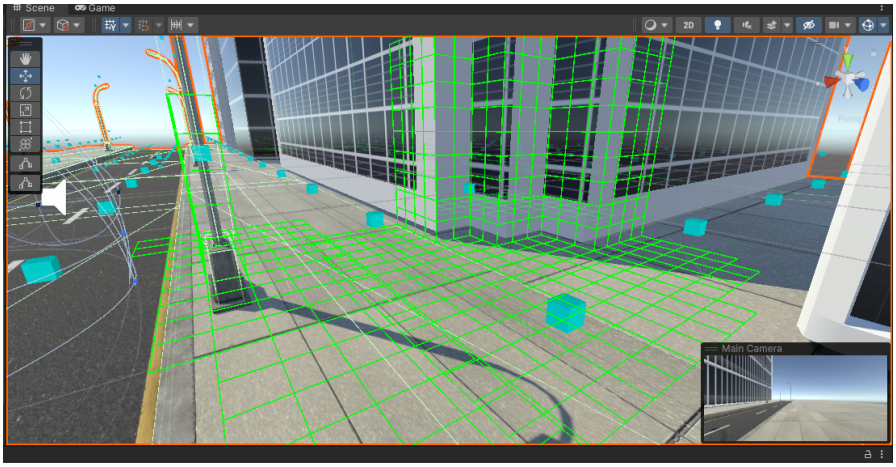


Figure 4.14: Voxelization of the scene

Once we finish the voxelization, we can bake the scene. Although Microsoft proposes using their Azure cloud service[24] to bake the scene, we can also bake it locally. The baking process uses docker[25] to set up the environment. The baking process takes a lot of time to complete. It took us around 2 hours to bake the scene. If we had to bake the scene multiple times, we could use the Azure cloud service to bake the scene faster.

Multiple channel recording in Unity In unity, by design, only one audio listener can be active at a time [26]. This design means that we can only record one channel at a time. To record multiple channels, we have to record each channel separately. We solve this issue by creating a script that will record each channel separately by moving the audio listener to the microphone's predetermined positions and recording the sound. We then merge the channels to create a multi-channel audio file.

4.4.3 Managing sound in game engine

In game engines, sounds travel instantly. This concept means that if we play a sound, we will hear it instantly. In real life, sound travels at 343 m/s, so we will hear it after the time it takes for the sound to travel the distance separating us from the sound origin. We have to take this into account when we create the simulation. We have to play the sound at the right time. We have to calculate the time it takes for the sound to reach the microphone. We can calculate the time with the following formula:

$$t = \frac{d}{v} \tag{4.3}$$

Where t is the time, d is the distance between the sound source and the microphone, and v is the speed of sound. The speed of sound is 343 m/s. We can then play the sound after the calculated time.

Since we record each channel separately, the time difference between each channel depends on the microphone's position. We show an example of this effect in figure 4.15 with a large distance between each microphone.

[24] https://azure.microsoft.com/fr-fr
[25] https://www.docker.com/
[26] https://docs.unity3d.com/Manual/class-AudioListener.html

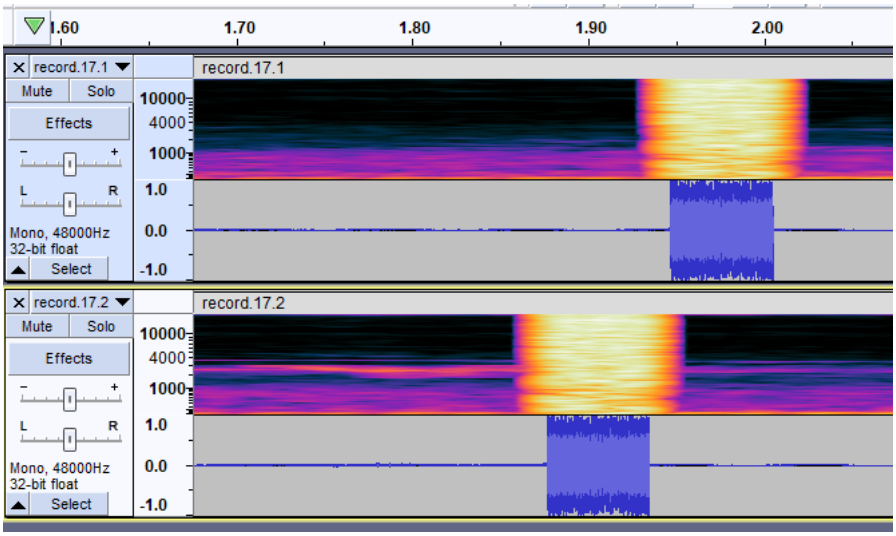


Figure 4.15: Time difference between channels

Recording the sound To record the sound, we use the *AudioListener* component of Unity and gather each sample of the sound. We then save the samples in a wav file. We use *SavWav.cs* script created by *darktable* [27] to create the file and save data in it.

4.4.4 Creating the dataset

Based on each class defined in section 3.1.2, we can create a folder correspondingto the class. When the simulation runs, it saves the audio file in the correct folder. Since the simulation always knows the state of the objects, we can use this information to define in which folder the audio file is saved. This process allows us to create a dataset based on the same design as the one created with real-life data.

Once we create the dataset, we can use it to extend the dataset created with real-life data to improve the neural network's performance.

4.5 Adversarial Attack

To realize the adversarial attack, we needed to have a trained model. We used the model trained on the dataset created with real-life data. We used the *Fast Gradient Signed Method* (FGSM) defined in section 3.4 as the adversarial attack design. Since we saved the model at the end of the training, we can load it and use it to create the adversarial attack.

4.5.1 Adversarial example generation

To generate adversarial examples, we need to do the equivalent of one pass of the training. The goal is to find the gradient direction that maximizes the loss. To find the sign, we can do as follow:

```
loss = criterion(model(input_tensor), label)
loss.backward()
perturbed_input_tensor = input_tensor + epsilon * input_tensor.grad.sign()
```

[27] https://gist.github.com/darktable/2317063

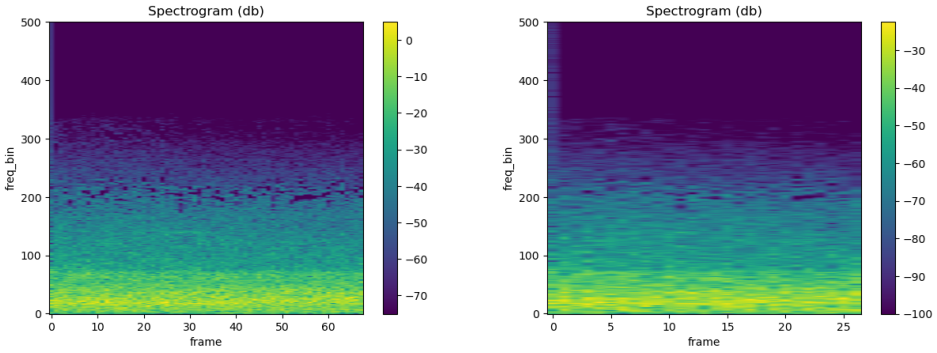
The $model(input_tensor)$ gives us the model prediction. The $label$ is the true class of the $input_tensor$. The $criterion$ is the loss function. We calculate the loss value when we call $backward()$. Once we find the loss, we can find the gradient direction by taking its sign. Once we have the direction of the gradient, we can multiply our $epsilon$ by the direction to find an image modified to maximize the loss, hence maximizing the miss-classification of the image.

The point of the FGSM on images is to get a modified image that is undetectable by the human eye. To achieve this, the $epsilon$ must be the smallest possible. To find the smallest possible $epsilon$, we can run the FGSM multiple times with an increasing $epsilon$ until we find the most efficient value of $epsilon$. This value will show the level of resistance of the model to adversarial attacks.

Once we have the $perturbed_input_tensor$, we can save it as an adversarial example.

4.5.2 Audio signal reconstruction

Since we use spectrograms, but our system records audio directly from a microphone, we need to convert the adversarial example to an audio file to attack our system. We use the *friggin-lim* algorithm from *pytorch* to reconstruct the audio signal from the spectrogram. Since going through a spectrogram and back to an audio signal uses multiple approximations, the reconstructed audio signal will differ from the original audio signal. We can see that the signal loses some of its information and has a smaller definition in figure 4.16.



(a) Audio signal before reconstruction

(b) Audio signal after reconstruction

Figure 4.16: Spectrograms before and after the reconstruction

5
Results and Analysis

5.1 Sound Propagation Simulation

5.1.1 Dataset Augmentation

5.2 Neural Network model results

5.3 Adversarial Attack results

5.3.1 Adversarial Attack mitigation

6 Conclusions and Future Work

6.1 Conclusions

6.1.1 Specification Fulfillment

6.2 Future Work

6.2.1 Sound Propagation Simulation

6.2.2 Sound Propagation Simulation bachelor's thesis

A thesis named *SimSound3D is*

6.2.3 Advanced adversarial attack

6.2.3.1 Patch attack

6.2.3.2 Targeted attack

6.2.4 Dataset publication

6.2.4.1 Dataset annotation

6.2.5 Loxo Ears model

A
Appendix

A.1

List of Tables

List of Figures

Bibliography

[1] Pierre-Amaury Grumiaux, Srdjan Kitic, Laurent Girin, and Alexandre Guerin. A survey of sound source localization with deep learning methods. The Journal of the Acoustical Society of America*, 152(1):107–151, jul 2022.*

[2] Jort F. Gemmeke, Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. Audio set: An ontology and human-labeled dataset for audio events. In 2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 776–780, 2017.*

[3] Shlomo E. Chazan, Hodaya Hammer, Gershon Hazan, Jacob Goldberger, and Sharon Gannot. Multi-microphone speaker separation based on deep doa estimation. In 2019 27th European Signal Processing Conference (EUSIPCO)*, pages 1–5, 2019.*

[4] Xiaofei Li, Laurent Girin, Fabien Badeig, and Radu Horaud. Reverberant sound localization with a robot head based on direct-path relative transfer function. In 2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, oct 2016.*

[5] Amengual Garamp. Spatial analysis and auralization of room acoustics using a tetrahedral microphone, Apr 2017.

[6] Carlos Fernández Scola and María Dolores Bolaños Ortega. Direction of arrival estimation : A two microphones approach. In 2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2010.*

[7] Ruoyu Sun. Optimization for deep learning: theory and algorithms, 2019.

[8] Jiawei Zhang. Gradient descent based optimization algorithms for deep learning models training, 2019.

[9] Ch Sekhar and P Meghana. A study on backpropagation in artificial neural networks. Asia-Pacific Journal of Neural Networks and Its Applications*, 4:21–28, 08 2020.*

[10] Shiv Ram Dubey, Satish Kumar Singh, and Bidyut Baran Chaudhuri. Activation functions in deep learning: A comprehensive survey and benchmark, 2022.

[11] Zhiyong Hao, Yixuan Jiang, Huihua Yu, and Hsiao-Dong Chiang. Adaptive learning rate and momentum for training deep neural networks, 2021.

[12] Jürgen Schmidhuber. Deep learning in neural networks: An overview. Neural Networks*, 61:85–117, jan 2015.*

[13] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A survey of deep neural network architectures and their applications. Neurocomputing*, 234:11–26, 2017.*

[14] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[15] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[16] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. Proceedings of the IEEE*, 86(11):2278–2324, 1998.*

[17] Mariam Yiwere and Eun Joo Rhee. Sound source distance estimation using deep learning: An image classification approach. Sensors*, 20(1), 2020.*

[18] Sharath Adavanne, Archontis Politis, and Tuomas Virtanen. A multi-room reverberant dataset for sound event localization and detection. In Proceedings of the Detection and Classification of Acoustic Scenes and Events 2019 Workshop (DCASE2019)*, pages 10–14, New York University, NY, USA, October 2019.*

[19] Archontis Politis, Sharath Adavanne, and Tuomas Virtanen. A dataset of reverberant spatial sound scenes with moving sources for sound event localization and detection. In Proceedings of the Detection and Classification of Acoustic Scenes and Events 2020 Workshop (DCASE2020)*, pages 165–169, Tokyo, Japan, November 2020.*

[20] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples, 2015.

[21] Nathanaël Perraudin, Peter Balazs, and Peter L. Søndergaard. A fast griffin-lim algorithm. In 2013 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics*, pages 1–4, 2013.*