Language Detection on the Edge

Sensors, Actuators and Embedded AI

by

Niels van der HEIJDEN
Data Science M.Sc. (EIT),

and

Dalim WAHBY
Autonomous Systems M.Sc. (EIT)

Polytech Nice-Sophia
Université Cote d'Azur

Supervisor: Benoit Miramond
Submitted: May 1, 2023

Contents

1 Introduction

AI on the edge, which refers to running artificial intelligence algorithms directly on devices such as smartphones, cameras, and sensors, has become increasingly significant in recent years. This approach to AI processing can improve efficiency, reduce latency, and enhance privacy by processing data locally rather than sending it to a remote server.

One important application of AI is the ability to identify spoken languages. This technology can benefit various industries such as customer service, law enforcement, and healthcare by providing real-time language translation services, improving communication, and reducing errors. Overall, AI on the edge and language identification have the potential to improve productivity, enhance user experience, and facilitate global communication.

By combining these two fields of research, we found that new question arise, such as:

1. How can we design a neural network, that classifies languages for use on the edge?
2. How does the neural network perform on the edge compared to on a computer with full computing power?
3. How much storage does the neural network require on the edge?

In this project, we develop a neural network (NN), that is able to classify spoken language. To this end, we pre-train our model on the FLEURS data set and then fine-tune it on our own generated data set. This is done, to increase the sensitivity of our network to noise and device specific patterns of the used microphone. Subsequently, we convert the trained model to a format, which enables the NN to run on the edge, on an STM32, equipped with a microphone. To this end, we use the MicroAI framework, which uses quantization to convert our model from a Tensorflow model to a model in C [1].

This paper will cover our approach to developing such a NN and implementing it on the edge. We commence, by explaining our methodology. Subsequently, we talk about the results of our NN running on a computer with full computing capacities and then talk about the results, that we get from running the program on the edge. The conclude, this paper with a short conclusion of our project and a short outlook on further research.

2 Methodology

2.1 Workflow

The workflow of this project is a multi-step process, including both technical and theoretical parts. Every technical application has a theoretical counterpart where the specifications or decision are being decided. An overview is show in Figure 1.

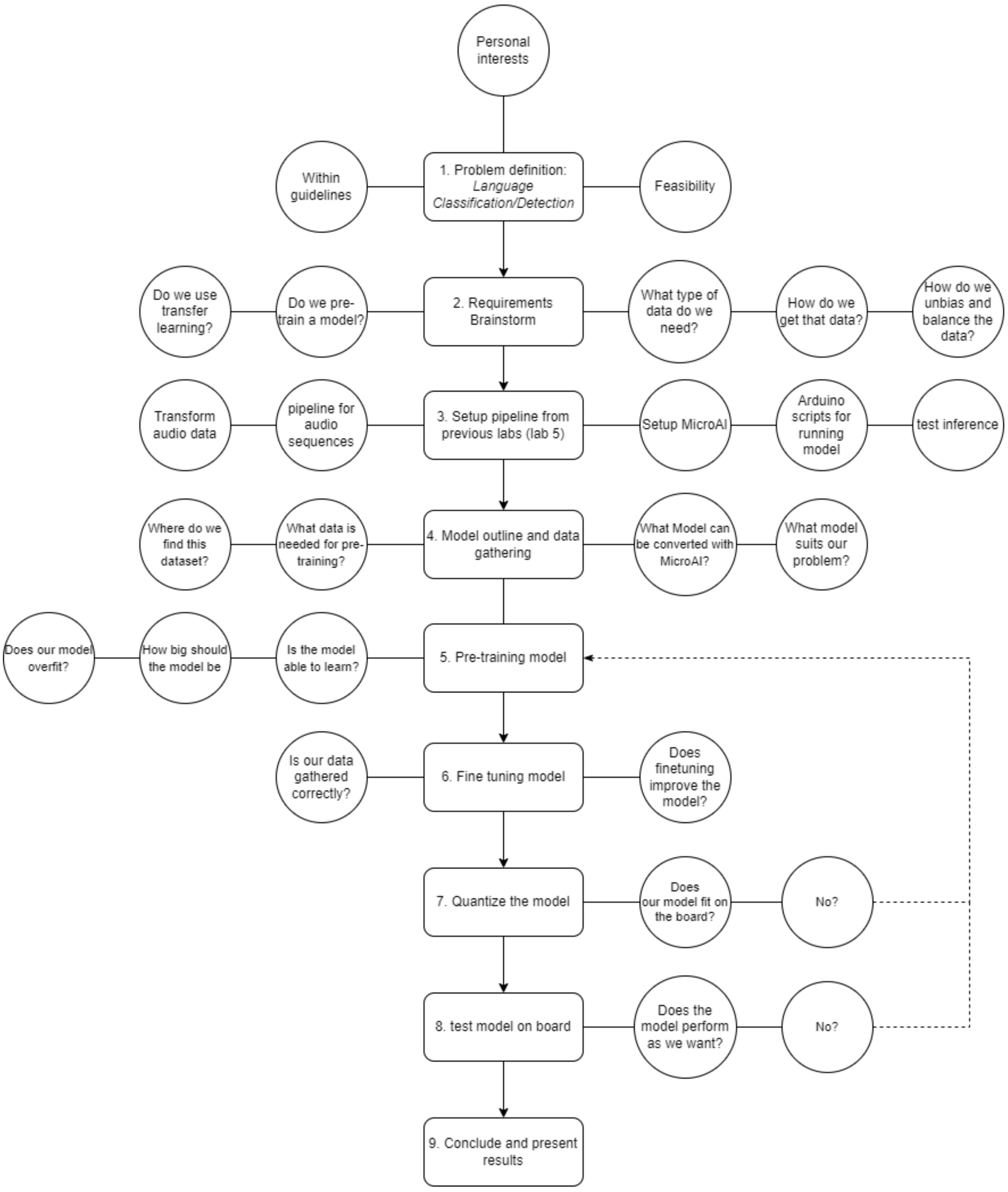

Personal interests
1. Problem definition: *Language Classification/Detection*
Within guidelines
Feasibility
2. Requirements Brainstorm
Do we use transfer learning?
Do we pre-train a model?
What type of data do we need?
How do we get that data?
How do we unbias and balance the data?
3. Setup pipeline from previous labs (lab 5)
Transform audio data
pipeline for audio sequences
Setup MicroAI
Arduino scripts for running model
test inference
4. Model outline and data gathering
Where do we find this dataset?
What data is needed for pre-training?
What Model can be converted with MicroAI?
What model suits our problem?
5. Pre-training model
Does our model overfit?
How big should the model be
Is the model able to learn?
6. Fine tuning model
Is our data gathered correctly?
Does finetuning improve the model?
7. Quantize the model
Does our model fit on the board?
No?
8. test model on board
Does the model perform as we want?
No?
9. Conclude and present results

Figure 1: Flow diagram of the project's process

In Figure 1 the squares describe each step in the process and the circles are some of the most important questions we asked ourselves at this stage. Furthermore, the solid arrows indicate the flow of the process, while the solid lines indicate connected questions. Finally, the dotted arrows indicate cases in which we come back to previous steps to change data pipelines, change the model architecture or model parameters.

Communication
The decision making within the group mainly took place by discussing the problem at hand and mutually agreeing on the best solution. E.g. we decided to pre-train due to the likelihood of overfitting on our self-collected data. Naturally, not every part of the project was worked on in pairs, hence some of the code and decision were made on an individual level. However, both team members know what decisions have been made and why that contributed to the final objective.

2.2 Convolutional Neural Network

Choosing a neural network
Since the research field of Artificial Intelligence and Deep Learning is ever changing and developing, there are plenty of methodologies to apply to each problem. However, there are a number of widely accepted architectures that consistently outperform other methods. For signal data, this is the Convolutional Neural Network, CNN in short, which is a deep learning architecture, designed to capture spatial correlation within an input frame (Image or Signal time window). Other networks such as Recurrent Neural Networks also attempt to mimic the memorization of a sequence, which is useful for speech recognition, but are harder to train and tend to lose a lot of information over time.

Next to its characteristics, the embedding of the neural network should be taken into account. The MircoAI framework, which allows us to convert a full neural network in python to C code, does not support the quantization of Recurrent Neural Networks.

With this information in mind, it is safe to assume that the Convolutional Neural Network is most likely to give a desirable results, given the objective of the project.

CNN Architecture
Now that the CNN architecture has been chosen as the method to classify the spoken language, we will briefly give an overview of its most important components. More specifically, we focus on the one dimensional (1D) CNN, which is the architecture that is usually applied for handling audio data.

Just like any other neural network architecture CNNs are made up of several layers, each with a specific function:

1. Input layer: This layer takes in the raw signal data, often these come in the form of

the input values of a pixel, but can be used for other kinds of signals such as audio.

2. Convolutional layer: This layer applies a set of filters to the input image in order to extract features such changes in amplitude or volume. Each filter is applied to a small region of the input, and the output of each filter is then passed to the next layer.

3. Pooling layer: This layer downsamples the output of the convolutional layer by taking the maximum or average value in small regions of the feature map. This helps to reduce the size of the feature map and hence the number of parameters while preserving the most important features. This is crucial for the embedding of the neural network, since it will significantly reduce the amount of parameters used for inference on the board.

4. Fully connected layer: This layer takes the output of all the feature maps that have been created and feeds it into a traditional neural network architecture, consisting of one or more layers of densely connected neurons. This allows the network to learn more complex features and make a final prediction based on the input image.

5. Output layer: This layer produces the final output of the network, which is typically a probability distribution over a set of target classes.

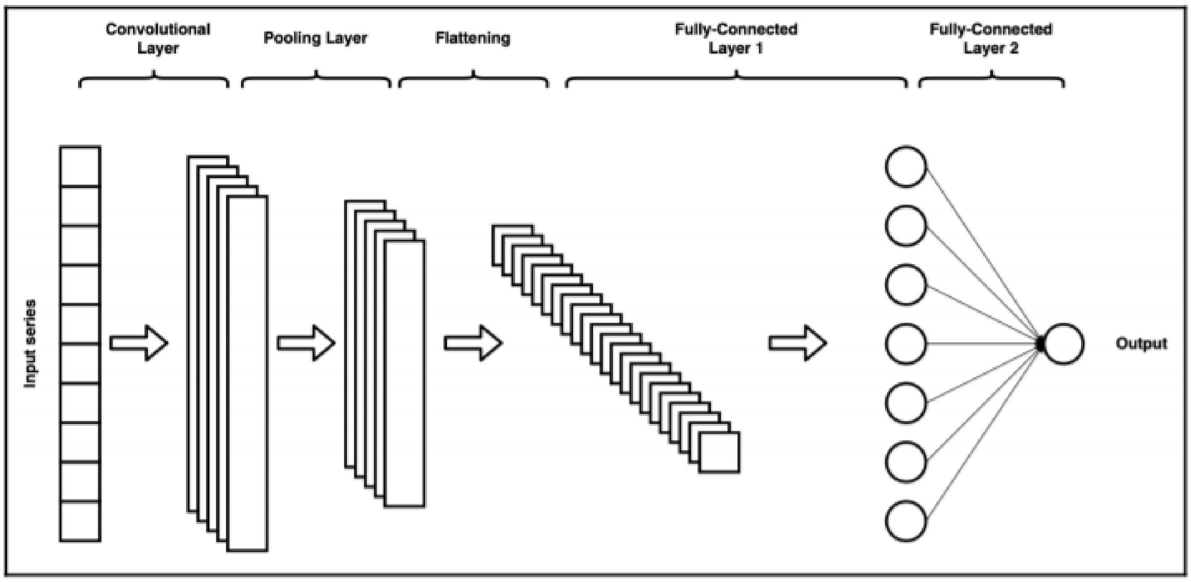


Figure 2: General architecture of a 1D CNN

In short, the convolutional layers try to capture the spatial correlation within a set input grid, where these values are then transformed into higher dimensions so they embed more information than just the raw input. Finally, all this information is passed to a standard fully connected neural network, where the non-linear behaviour of the input is learned and the classification process can be carried out.

2.2.1 Pre-Training on Common Voice

We decide to pre-train the model, because this will benefit the performance of the model in multiple ways. First, the model is less likely to overfit to our manually collected data. Second, neural networks generally require large amounts of data to be able to train, hence more data samples improve the performance of the model overall. Finally, there might be biases in our data sampling, due to the specific microphone and surroundings of our data gathering environment. These biased are reduced when pre-training on different data.

For pre-training we have selected the FLEURS dataset, which is short for Few-shot Learning Evaluation of Universal Representations of Speech. The FLEURS dataset is a benchmark for evaluating few-shot learning methods for speech recognition. It consists of 20 hours of speech data, with each speaker contributing 20 minutes of speech across 10 different languages. The dataset is designed to test the ability of models to recognize new languages with minimal training data, and it includes a set of evaluation metrics and a baseline model for comparison. FLEURS is intended to facilitate research in few-shot learning and cross-lingual speech recognition.

The data is processed where each sample consists of 1 second of data with a sampling rate of 16.000 Hz. Thus, one minute speech data would be transformed into 60 samples, containing 16.000 values each.

A subset has been selected and show in figure 3, the following languages are captured with their respective frequencies:

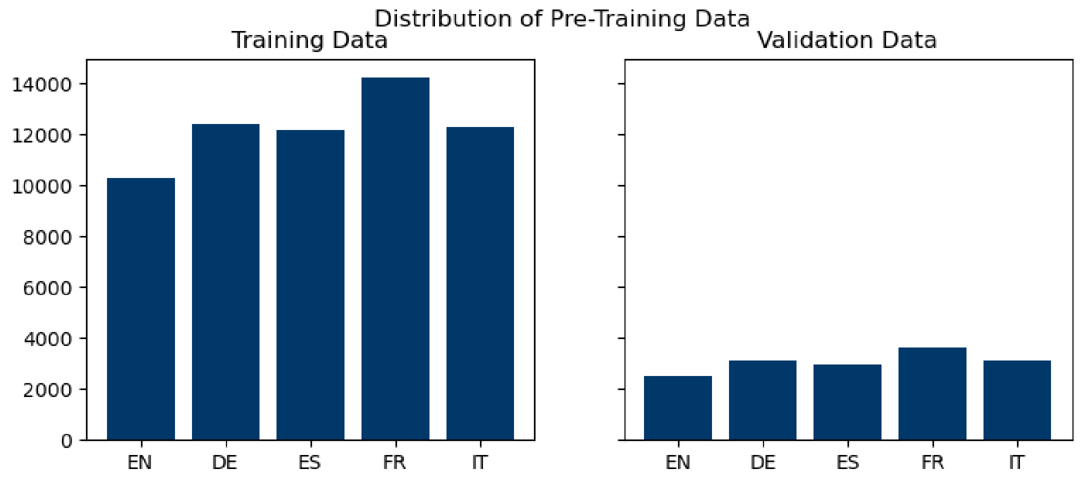


Figure 3: Distribution of pre-training data

We made this selection specifically, because of the amount of native speakers in our personal network. For each language we want to capture at least one male and one female sample.

Model Architecture for Pre-Training

Category	Hyper-Parameters	Value
Input	Input Size	(16.000 x 1)
	Sample rate	16.000
	Sample length	1 second
	Batch Size	128
	Epochs	50
Architecture	4 Convolutional layers 1D	filters = (8,16,32,64) respectively, kernel size = (20,8,4,2) respetively , ReLU
	3 Max Pooling Layer 1D	pool size = 2
	1 Average Pooling 1D	pool size = 4, activation = ReLU
	Flatten	None
	Fully Connected layer	neurons = 5, Softmax
Training Softmax	Activation Function	Activation = Relu
Learning	Optimizer	adam
	Loss Function	categorical cross entropy
	Learning Rate	0.0001
	Loss Rate	0.000001
Output	Output shape	(None, 4)

Table 1: Overview of input, parameters and training settings of the model

Layer (type)	Output Shape	Number of Parameters
Convolutional 1D	(None, 1559, 8)	168
MaxPooling1D	(None, 799, 8)	0
Convolutional 1D	(None, 198, 16)	1040
MaxPooling1D	(None, 99, 16)	0
Convolutional 1D	(None, 48, 32)	2080
MaxPooling1D	(None, 24, 32)	0
Convolutional 1D	(None, 23, 64)	4160
AvgPooling1D	(None, 5, 64)	0
Flatten	(None, 320)	0
Dense	(None, 5)	1605
Total Parameters		9053
Trainable Parameters		9053
Non-Trainable Parameters		0

Table 2: Summary of the architecture with number of trainable parameters.

Now the general outline of our model has been described, this section presents the final structure of our Convolutional Neural Network, used for pre-training and fine tuning on a larger device. The architecture, hyperparameters and layers are give in table 1 and 2. These have been selected in two ways, trial and error and by selecting hyperparameters that are likely to help our model fit to this specific problem. For instance, we expect the model to train on different sounds that are present in each languages. Therefore the window size cannot be to small, otherwise these different sounds are not captured as a whole. Furthermore, we experimented with different learning rates, optimizers and internal settings such as MaxPooling kernels. The results show in the tabel are the parameters that yielded the highest accuracy.

2.2.2 Fine-Tuning on Own Data Set

Now the first model has been pre-trained for our specific needs and with a much larger dataset than we could generate ourselves, we will fine-tune the model.

Data collection
The data that will be used for fine-tuning has been collected by ourselves, using the board and attached microphone. Here, as mentioned before, we selected the 5 languages for which we could collect at least one female and male voice. In total we collected 10 minutes of data. Roughly 1 minute per person and one male and female person per language. After permutation, the splitted training, validation and testing sets are described in Figure 4. Some imbalance can be seen in the data, due to the random permutations and due to the imbalance in sample length. However, this should not greatly impact the performance, because none of the samples is heavily over or under-represented.

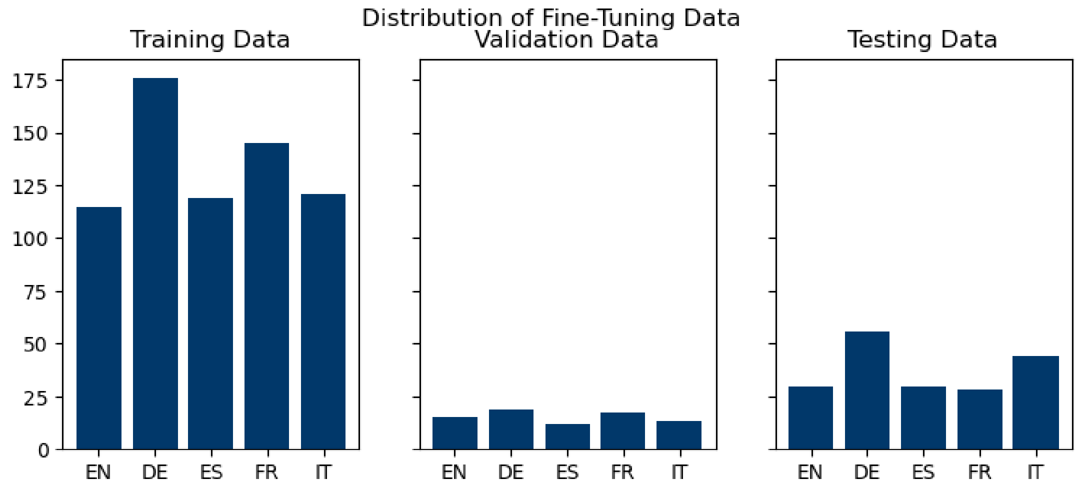


Figure 4: Distribution of fine-tuning self-collected data

Fine-tuning
After collectin the data, there are a couple of things to note when fine tuning a model, to

ensure a smooth transition and increased performance. First, the learning rate has to be much lower compared to regular training, since we do not want the model to completely change weights. We simply want to give it a nudge in the right direction. This should ensure the model can work properly with our microphone that is attached to the board. Second, we should be very careful of overfitting on the self-collected data, since the dataset is smaller then usual. Furthermore, the original model can be frozen when fine tuning it with new data. However, since our model is relatively small overall and there is not concrete evidence that freezing improves later performance, we refrain from doing this. The final parameters for fine tuning are given in table 3.

Category	**Parameter**	**Value**
Model	CNN	pre-trained CNN (Tab 2)
Input	shape	16.000 x 1
Output	shape	5 x 1
fine-tuning	Learning rate	10e-6
	Epochs	100
	Batch size	8

Table 3: Settings Fine-tuning

2.3 Processing on Sensor Data Setup

After fine-tuning the model, we quantize the model. This is done using the MicroAI library, that automatically translates our Tensorflow model to C code. We then use the Arduino scripts for running inference on the board and report its performance in terms of accuracy. To be able to test the performance of the model, we select a random subset of the self-collected data and label it as the test set. During training, the model will never observe this data and can therefore be used to test inference on the board. The distribution and size of this data can be seen in the left barplot of Figure 4.

2.4 Definitions DMA and ADC

DMA (Direct Memory Access) and ADC (Analog-to-Digital Converter) are two different components commonly used in embedded systems, and they serve different purposes.

DMA is a hardware mechanism that allows data to be transferred between peripherals and memory without the need for intervention by the CPU. It helps to reduce the CPU's workload and improve the system's performance by allowing peripherals to directly access memory. In other words, DMA is a method for transferring data without using the CPU.

On the other hand, ADC is a device that converts analog signals into digital signals. In an embedded system, an ADC is commonly used to read input signals from sensors and convert them into digital values that can be processed by the system. ADCs are essential for systems that need to measure analog signals accurately.

In summary, while DMA is a mechanism for transferring data between peripherals and memory without the CPU's involvement, ADC is a device that converts analog signals into digital signals. They are used for different purposes in an embedded system.

2.5 Power Consumption on the Edge

Computing the energy consumption of the board is based on the diagram of Figure 5. In the diagram L denotes the period of time, where the model is recording and predicting and T denotes the period of time, where the model only records.

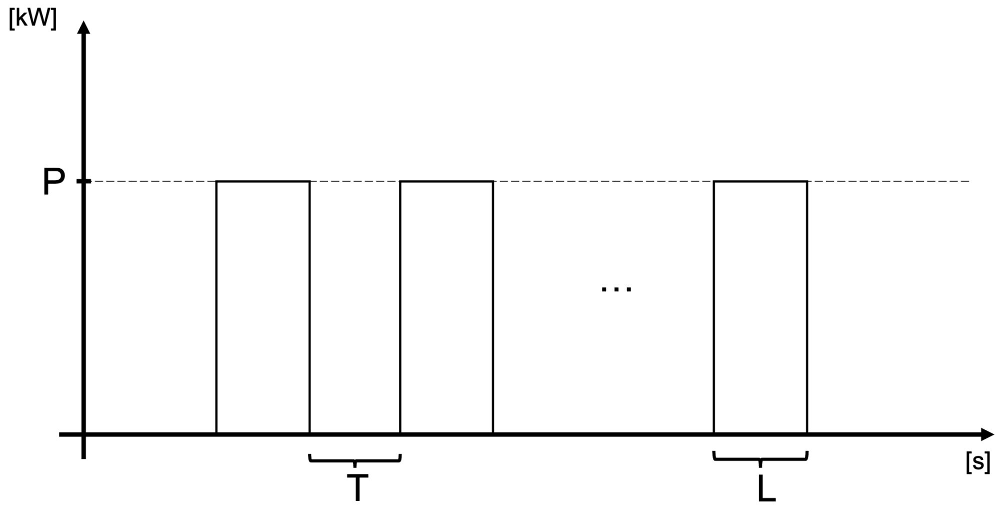


Figure 5: Diagram of energy consumption over time

Resulting from the diagram, we get the following equation to compute the power consumption per inference assuming that $P_T = 0$, as shown in Equation 1.

$$P_{avg} = \frac{P_L L}{T + L}. \tag{1}$$

However, Equation 1 has the limitation that it does not capture the base power consumption of the board in low-power mode. If we were to account for the low-power mode, the equation would look as in Equation 2. For this report, we assume that the low-power mode does not consume power, hence enabling us to use Equation 1. To compute the battery lifetime, we simply divide the capacity of the battery E_{tot} by P_{tot}.

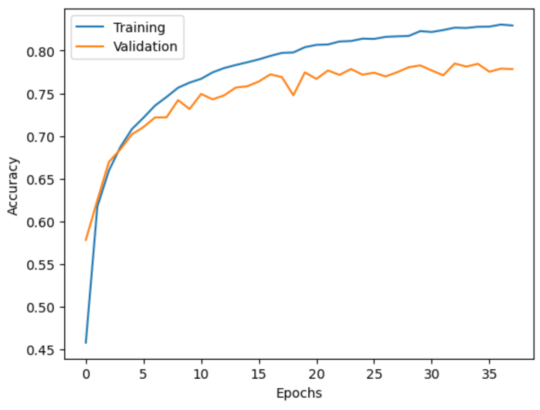
$$P_{avg} = \frac{P_L L + P_T T}{T + L} \tag{2}$$

3 Results

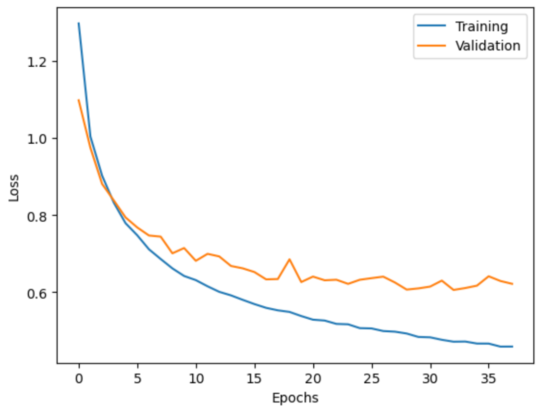
In this section we will describe the results after applying the methods described in the previous section. As a reminder, the aim of this report is to classify spoken languages on the edge to best of our ability. Hence, we analyse the performance and present the results by using the accuracy metric. This metric simply captures in which situation the most samples are correctly classified. Since, applications such as customer service or smart-device integration, mainly require a high level of accuracy for the most common classes, we will adopt this metric.

3.1 Results of Pre-Training

In Figure 6, the results for training model 2 is shown. As said before we trained for 50 epochs, however the early callback, which stops training after lack of improvement, halted the training after 38 epochs (x-axis). In the left image we see the accuracy plotted over time (epochs), here we see the common behaviour in machine learning. At some point (around 25 epochs in our case) the validation line stagnates and the training line keeps increasing. From this moment onwards we stop the training and capture the weights as our final model. The right visualization of Figure 6 displays the loss, with captures how wrong the prediction model is compared to the truth value. A similar behaviour as for the accuracy can be observed, since the the training loss will keep decreasing, but the validation loss will stagnate after hitting a local minima. Naturally the loss and accuracy stagnate around the same number of epochs. Finally, when feeding the self-collected data to this model, it achieves an accuracy of **20%** which is equal to complete randomness.



(a) Accuracy



(b) Loss

Figure 6: Training Results

Furthermore, we can visualize the decision making of the model in more detail, by showcasing it's predictions compared to it's truth values (on the test set) in a confusion matrix (shown in figure 7). Here we see the desired behaviour, where the highest values are shown on the diagonal (indicating right classification). We can observe that the accuracy results range between **74,8%** for English and **83,2%** for German. We believe this performance is quite good for a small model, given that some of the samples (reminder: 1 second of recording) are tricky to classify, even for humans. Think of moments of silence or sounds that are carried over between similar languages. With these results we feel confident to move on to the fine-tuning phase.

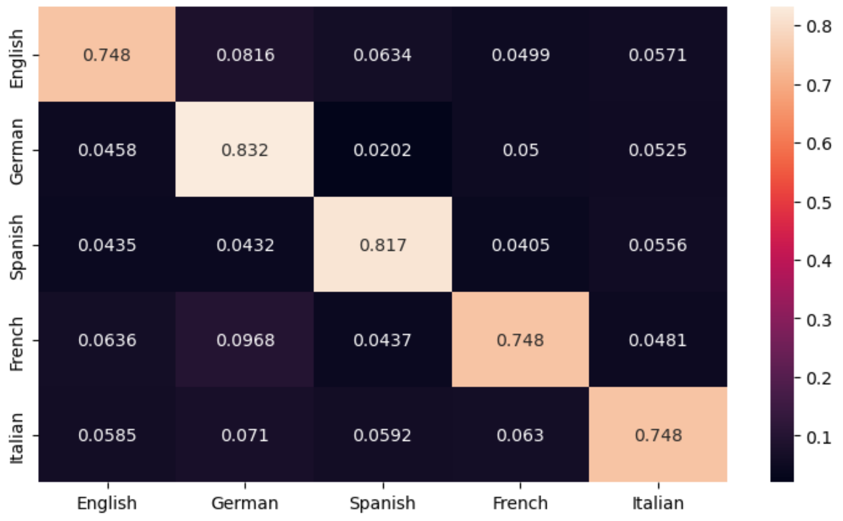


Figure 7: Confusion matrix for pre-training

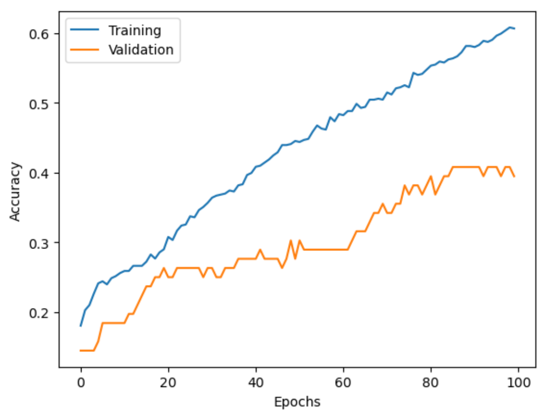
3.2 Results of Fine-Tuning

After the previous results, we attempt to finetune the model, due to the reasons given in chapter 2.2.2. The following figures describe the results after fine-tuning with the data collected by ourselves (reminder, see table 4). It can be observed in the left plot of Figure 8, that the validation accuracy stagnates around 42%. From this information we can draw the following two conclusions.

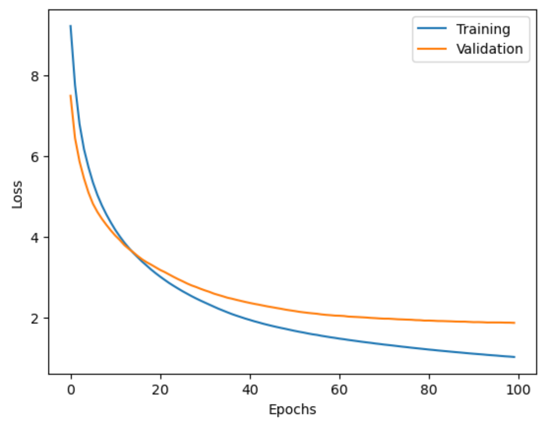
1. First, when only training on the self-collected (thus, no pre-training), we get an accuracy **37%**. Hence, pre-training increases the accuracy of our model (with self-collected data) with 5 percentage points. Hence, we confirm that pre-training our model has a positive impact on our model's ability to classify our self-collected data.

2. Second, without fine-tuning the model and hence getting a **42%** accuracy on the self-collected test set. We achieve an accuracy of **20%** on that test data. This means that without fine-tuning, the pre-trained model achieves a 22 percentage points drop when not being fine-tuned. Hence, we can conclude that fine-tuning our model has a positive impact on our model's ability to classify our self-collected data.

Finally, we can prove that our model does not overfit, by analyzing the accuracy and loss graphs. When the validation curve flattens, we stop the training and take the weights of the corresponding epoch.



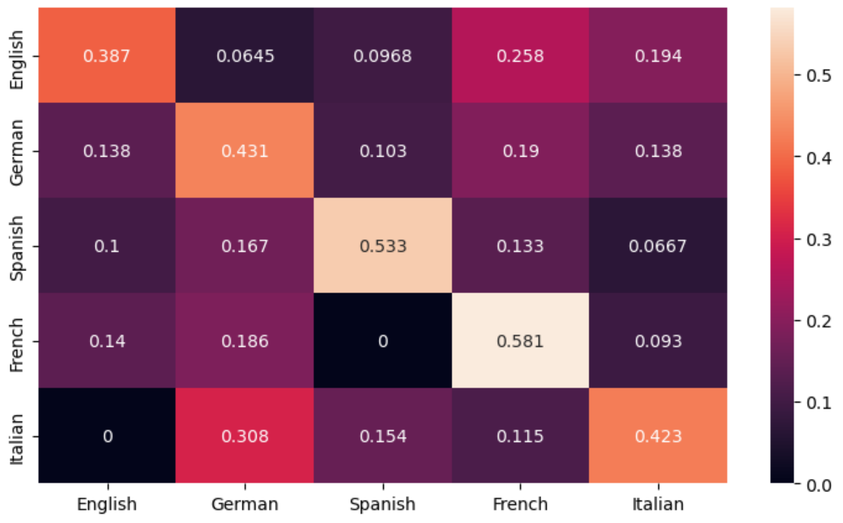
(a) Accuracy



(b) Loss

Figure 8: Fine-tuning results

In Figure 9, we see a similar behaviour as before, however the model seems to perform better on different languages. We expect that in our data, some language specific samples are harder to classify than others, due to noise or bad quality. However, the model still performs relatively well considering the noisiness of the data.



	English	German	Spanish	French	Italian
English	0.387	0.0645	0.0968	0.258	0.194
German	0.138	0.431	0.103	0.19	0.138
Spanish	0.1	0.167	0.533	0.133	0.0667
French	0.14	0.186	0	0.581	0.093
Italian	0	0.308	0.154	0.115	0.423

Figure 9: Confusion matrix for fine-tuning

3.3 Results on the Edge

Now we have proven that pre-training and fine-tuning are the best strategies for optimizing our model's performance, we quantize the following three models:

1. The model only pre-trained on the FLEURS dataset
2. The model only trained on our self-collected dataset
3. The model pre-trained on the FLEURS dataset fine-tuned on our self-collected dataset.

Tabel 4 displays the results after quantizing the model and running running inference on the board with the test set of self-collected data. We can observe that most models lose approximately 1 percentage point of accuracy after quantization. This is to be expected since information is being lost when reducing the memory footprints of the weights. After quantization, the best performing model is still the pre-trained and fine-tuned version.

Model type	**Accuracy**	**Accuracy after quantization**
pre-trained	20%	19%
Pre-trained & Fine-tuned	**42%**	**42%**
only trained own dataset	37%	36%

Table 4: Results models before and after quantization

The measured latency L of the board is $152ms$

To compute the the battery life time, we use the following values, where P denotes the power consumption of the board in active mode, meaning that it records data and predicts a language. $E_{battery}$ denotes the energy of the battery using two batteries.

$$P_L = 62mW \qquad E_{battery} = 2 * 3900mWh = 7800mWh$$

$$P_{avg} = \frac{116ms * 62mW}{116ms + 1s} = 6.44mW$$

$$\Rightarrow t_{Autonomy} = \frac{E_{battery}}{P_{avg}} = \frac{7800mWh}{6.44mW} = 1210.345h.$$

As shown in the calculations above, the board has an autonomy of 1210.345 hours. Furthermore, it has an average power consumption of $6.44mW$. The NN on the board uses 5% of the program storage space (58536 bytes out of 1032192 bytes).

4 Conclusion

This paper has attempted to create a model, capable of running on the edge, that can accurately classify the language being spoken in real time. The application of techniques such as pre-training on a larger FLEURS dataset and fine-tuning the model with the data collected on the edge-device, has proven to be effective. Ultimately reaching an accuracy of **44%** on the edge device. Overall, these results are satisfactory given the small size of the model and the noisy data that is gathered by the edge device. However, there are definitely factors that could be improved upon.

future work
To improve on the work in this paper, we suggest the following adjustments or adaptations:

1. Implement a Recurrent Neural Network (RNN) or Long Short Term Memory model (LSTM). These networks are proven to work well on sequential one dimensional data, such as ours. However, due to the lack of implementation in the MicroAI framework, this could not be explored in this project.

2. Aggregate the output in longer windows, for real life applications. Since it is not required for most real life applications to classify the spoken language in just one second. The final output could be aggregated over a longer time window. For instance, if English is being spoken for 10 seconds and it correctly classifies for 4/10 seconds, it might still be the most common output in that 10 second time window. Hence, if you would take the maximum argument over a longer window, the accuracy may increase.

3. The number of self-collected datapoints with the edge-device could be further increased, e.g. include more different speakers. Currently we did not have the resources and network to include more people, however this would likely further increase the performance after fine-tuning.

References

[1] Novac, P.-E., Boukli Hacene, G., Pegatoquet, A., Miramond, B., and Gripon, V. Quantization and deployment of deep neural networks on microcontrollers. *Sensors 21*, 9 (2021).