

Deep Learning on Microcontrollers

Learn how to develop embedded AI applications using TinyML



Atul Krishna Gupta
Dr. Siva Prasad Nandyala

bpb



Deep Learning

—on—

Microcontrollers

Learn how to develop embedded AI applications using TinyML



Atul Krishna Gupta
Dr. Siva Prasad Nandyala

bpb

Deep Learning on Microcontrollers

*Learn how to develop embedded
AI applications using TinyML*

Atul Krishna Gupta
Dr. Siva Prasad Nandyala



www.bpbonline.com

Copyright © 2023 BPB Online

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor BPB Online or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

BPB Online has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, BPB Online cannot guarantee the accuracy of this information.

First published: 2023

Published by BPB Online

WeWork

119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-5551-800-2

www.bpbonline.com

Dedicated to

*My Parents Mr. Keshava Kumar Gupta and
Late. Smt. Lakshmi Devi Gupta and to
my wife Richa Gupta and to
my children Ananya Gupta and Avi Gupta*

— Atul Krishna Gupta

*My Parents Late Mr. Koti Nagaiah Nandyala and
Smt. Durgamba Nandyala and to
my brother Sambasiva Rao Nandyala and to
my wife Sandya Kemisetti*

— Dr. Sivaprasad Nandyala

About the Authors

Atul Krishna Gupta has held many positions as Research & Development Executive in companies such as Syntiant, Macom, Inphi (now Marvell) and Genum (now Semtech). He has over 25 years of experience in delivering all aspects of systems from IC design to software support. He has made contributions to various forums such as IEEE, SMPTE and OIF. Two technical Emmy Awards were granted to two companies for the technical work he led in the past. He was awarded with the Employee of the year award and Excellence in R&D award at Genum. Atul holds over 20 patents. Currently, his research interests are in the field of Battery Management Systems (BMS) where he is finding ways to use AI to make Electrical Vehicles (EV) safer and last longer. He has received his B.Tech degree in Electrical Engineering from Indian Institute of Technology, Kanpur, India and MS degree in Electrical and Computer Engineering from University of Calgary, Canada.

Dr. Sivaprasad Nandyala worked in Eaton Research Labs as Lead Engineer (Data Science) at Eaton India Innovation Center, Pune, India. Prior to Eaton, he worked in companies like Tata Elxsi, Wipro Technologies, Analog Devices & Ikanos Communications in multiple technology areas. Dr. Nandyala has over 35+ research publications, 1 patent grant and 6 patents under review. He obtained his Ph.D. in Speech Processing from NIT Warangal, India. He was an ERASMUS MUNDUS scholarship holder from the European government for his Postdoctoral Research at Politecnico di Milano (POLIMI), Italy.

About the Reviewer

Dr. Sanjay Boddhu is an experienced Research and Engineering leader with expertise in leading and mentoring geographically distributed teams, in the domains of Computer Vision, Image Processing, Natural Language Processing, Predictive Analytics, and Modelling. He is skilled in using various Machine Learning Ops approaches to design and develop real-world applications in Cloud and at Edge.

Acknowledgements

Atul Krishna Gupta: I am incredibly grateful to Kurt Busch for providing me the opportunity to learn and contribute to the emerging field of Artificial Intelligence (AI). I would like to thank Dr. Jeremy Holleman for their insightful conversation over various topics related to neural networks. I would also like to thank Dr. Stephen Bailey for help in the firmware of the TinyML board. I would not have been able to showcase the TinyML board without guidance from Mallik Maturi and Poupak Khodabandeh.

I would like to offer a special thanks to Zack Shelby and Aurelien Lequertier for making available their Machine Learning (ML) platform to the developer community free of cost. The platform enables zero code deployment of production grade AI deployment.

I would like to thank my wife and children for their patience and support to finish the book.

Dr. Sivaprasad Nandyala: The writing of a book is never a solo effort, and this “**Deep Learning on Microcontrollers**” book is no exception. Before anything else, I want to express my deepest gratitude to Mr. Atul Krishna Gupta who had faith in me from the beginning in writing this book. In addition, I would like to thank Dr. Sanjay Boddu for his feedback and suggestions in reviewing the book. I want to thank the TinyML community for all the great things they have done for the field. My heartfelt thanks go to my family and friends for their unwavering support and understanding during this hard journey.

Preface

As the title of the book suggests, this book is intended to enable readers from different backgrounds to make a tangible AI application, which can be deployed on the edge on off-the-shelf platforms such as Arduino or TinyML board. The focus of this book is on the practical aspects of AI deployment. The journey of AI deployment from demo quality to production grade is not easy. We have taken a realistic example to show the pitfalls and given ideas on how to overcome the roadblocks.

While the focus of the book is on the practical side, the book also provides a good academic background as well. The field of AI is evolving and it is not practical to have one comprehensive book on all the topics, but we have given insight into some of the advanced topics of the AI field.

Deployment of AI on the edge will require some hardware. For cost effective deployment, it is expected that companies will develop their unique hardware. However, for getting started, there are several hardware boards available from websites such as Digikey or Amazon. Readers can buy this type of hardware in the range of \$35-\$100.

This book is divided into **9 chapters**. Each chapter description is listed as follows.

Chapter 1: Introduction to AI – will show a continuum of traditional code-based solution and Artificial Intelligence based solution. It will show where an AI based solution will be suitable and how to approach the solution.

Chapter 2: Traditional ML Lifecycle – will cover how machine learning is different from classical methods, introduction to traditional ML life cycle, performance metrics, and the basics of deep learning (DL) and different DL algorithms. It also covers transfer learning. We will discuss several tools, libraries, and frameworks for developing and deploying ML models on various embedded devices and microcontrollers. We also cover the differences between learning and inference, ML model deployment and inferencing on different hardware platforms and their comparison at various deployment levels.

Chapter 3: TinyML Hardware and Software Platforms – will cover CPUs, GPUs, Raspberry Pi boards, TPUs, and Data Center Servers. We will also look at TinyML compatible microcontrollers and Raspberry Pi boards. We then focus on TinyML's hardware boards and software platforms for machine learning. We will discuss important software platforms, data engineering, and model compression frameworks.

Chapter 4: End-to-End TinyML Deployment Phases – will discuss embedded machine learning's (EML) basics, characteristics, and examples. Next, we will also explore EML's building blocks, pros and cons, and how to run an ML model on microcontrollers. We will discuss Edge Impulse and Arduino IDE platforms, their pros and cons, and how to use different hardware boards with them. Data collection from sensors and the different platforms will be covered. We will cover data engineering, model training with Edge Impulse, optimization, and inferencing for model deployment on TinyML hardware platforms.

Chapter 5: Real World Use Cases – will cover various use cases of the TinyML deployment. The chapter categorizes these deployments in seven categories. However, many applications overlap multiple categories. These applications just show the tip of the iceberg because we just got started. Over the next few decades, we are expecting an explosion of TinyML deployment. These examples are provided just to ignite the creativity of the reader, so that they can lead innovation and deploy AI solutions which do not exist today.

Chapter 6: Practical Experiments with TinyML – will utilize Arduino IDE for TinyML hardware experimentation. We will collect sensor data using the TinyML board, clean the data for the practical experiment (Air Gesture Digit Recognition), upload it to the Edge Impulse platform, train and test the model with Nano RP2040 board sensor data. Finally, we will download the Edge Impulse inference model and test it on the RP2040 using Arduino IDE to evaluate performance.

Chapter 7: Advance Implementation with TinyML Board – will deep dive on specific hardware accelerator chips, which provide AI specific computation at a fraction of cost and power relative to microcontroller-based architecture. The development boards are readily available on these hardware accelerator chips where readers can deploy an AI solution. The power of the entire solution can run from batteries months to years. The

chapter describes the entire flow of deployment in a few easy steps on the readily available Edge Impulse software platform.

Chapter 8: Continuous Improvement – will cover topics in improving the accuracy of the AI solution. AI is a data driven flow where the accuracy depends on the data. This chapter takes a deeper dive into a keyword detection application to demonstrate how to curate the data and improve the performance to take the solution from demo to production quality.

Chapter 9: Conclusion – will provide the conclusion of various aspects learned in the earlier chapters. This is an introduction book on AI and there are many topics which will require many more books. Some of those topics are mentioned in this chapter to ensure that the reader knows there is more to AI than what is covered in the book.

Code Bundle and Coloured Images

Please follow the link to download the *Code Bundle* and the *Coloured Images* of the book:

<https://rebrand.ly/yt0v6ae>

The code bundle for the book is also hosted on GitHub at <https://github.com/bpbpublications/Deep-Learning-on-Microcontrollers>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at <https://github.com/bpbpublications>. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePUB files available? You can upgrade to

the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at: business@bpbonline.com for more details.

At www.bpbonline.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit www.bpbonline.com. We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit www.bpbonline.com.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Table of Contents

1. Introduction to AI

Introduction

Structure

Objectives

Artificial Intelligence

Continuum of code writing and artificial intelligence

Exercise

Changing the paradigm

Neural Network

Machine Learning

Intelligent IoT System vs. Cloud based IoT system

Arduino Nano 33 BLE Sense board

Limited compute resources

Battery power limits

TinyML and Nicla Voice board

>10x parameters

>200x Power advantage

>20x Throughput

TinyML Ecosystem

Key applications for Intelligent IoT systems

Smart agriculture

Smart appliances

Smart cities

Smart health

Smart homes

Smart industry

Conclusion

Key facts

Questions

References

2. Traditional ML Lifecycle

Introduction
Structure
Objectives
Traditional methods
Machine learning landscape
 Supervised learning
 Unsupervised learning
 Reinforcement Learning (RL)
ML Performance Metrics
 Confusion matrix
Basics of DL and different DL algorithms
Transfer Learning
Tools and Different ML, DL frameworks
 Python
 Jupyter Notebooks
 Google Colaboratory
 TensorFlow (TF), TFLite and TensorFlow Lite Micro
 TensorFlow Lite
 TensorFlow Lite Micro
 AI Model Efficiency Toolkit (AIMET)
 Convolutional Architecture for Fast Feature Embedding (Caffe)
 CoreML
 Open Neural Network Exchange (ONNX)
 Open Visual Inference and Neural network Optimization (OpenVINO)
 Pytorch and PyTorch Mobile
Embedded Machine Learning (EML)
Difference between Learning and Inference
ML model deployment and inferencing on different platforms
Conclusion
Key facts
Questions
References

3. TinyML Hardware and Software Platforms

Introduction
Structure

Objectives

Servers at Data Centers: CPUs, GPUs and TPUs

Mobile CPU, Raspberry Pi board and its types

Microcontrollers and Microcontroller with AI accelerator

TinyML Hardware Boards

Arduino and Arduino Nano 33 BLE

Arduino Nicla Sense ME

Adafruit Feather

SparkFun Edge

NVIDIA Jetson Nano

Google Coral Edge TPU

Qualcomm QCS605

NXP i.MX 8M

STMicroelectronics STM32L4

Intel Curie

Syntiant TinyML

TinyML Software Suites

TensorFlow Lite Micro (Google)

uTensor (ARM)

Arduino Create

EloquentML

EdgeML (Microsoft)

EON Compiler (Edge Impulse)

STM32Cube.AI and NanoEdge AI Studio (STMicroelectronics)

PYNQ

OpenMV

SensiML

Neuton TinyML

Metavision Intelligence Suite 3.0 (Vision applications)

Data Engineering Frameworks

Edge Impulse

SensiML

Qeexo AutoML

TinyML Model Compression Frameworks

Quantization

Pruning

Low ranked approximation

Knowledge distillation

TensorFlow Lite

STM32 X-CUBE-AI

QKeras

Qualcomm AIMET

Microsoft NNI

CMix-NN

OmniML

Conclusion

Key facts

Questions

References

4. End-to-End TinyML Deployment Phases

Introduction

Structure

Objectives

Understanding Embedded ML

Introduction to Edge-impulse and Arduino IDE

Edge-impulse

Arduino Integrated Development Environment (IDE)

Arduino Driver Installation

Data collection from multiple sensors

Data collection from an Arduino board

Data collection from Syntiant board

Data engineering steps for TinyML

Cleaning

Organizing

Transformation

Model Training in TinyML software platforms

EON Compiler (Edge Impulse)

Model Compression

Pruning

Knowledge distillation

Model conversion

Quantization

Inferencing/Prediction of results with test data

[Model Deployment in TinyML Hardware board](#)

[Conclusion](#)

[Key facts](#)

[Questions](#)

[References](#)

5. Real World Use Cases

[Introduction](#)

[Structure](#)

[Objectives](#)

[Smart agriculture](#)

[Agriculture video analytics](#)

[Crop intruder detection](#)

[Crop yield prediction and improvement](#)

[Agribots](#)

[Insect detection and pesticide reduction](#)

[Weedicides elimination](#)

[Acoustic insect detection](#)

[Animal husbandry](#)

[Smart appliances](#)

[Vision AI for appliances](#)

[Audio AI for appliances](#)

[Sensors based AI for appliances](#)

[Smart cities](#)

[Safe and secure city](#)

[City maintenance](#)

[Parking enforcement systems](#)

[Traffic management](#)

[Maintaining bridges](#)

[Non-Smoking enforcement](#)

[Smart health](#)

[Cataract detection](#)

[Fall detection](#)

[Cough detection](#)

[Boxing Moves Detector](#)

[Mosquito detection](#)

[Snoring and sleep apnea detection](#)

Smart home

Person detection at the door

Glassbreak detection

Smart baby monitoring

Voice recognition for home automation

Smart industry

Railway track defect detection

Telecom towers defect detection

Defect detection in components

Smart automotive

Drowsy driver alert

Advance collision detection

Conclusion

Key facts

Questions

References

6. Practical Experiments with TinyML

Introduction

Structure

Objectives

Introduction to Nano RP2040 TinyML board

Setting up Arduino IDE and testing the Nano RP2040 Board

High level steps involved in the air gesture digit recognition in Edge

Impulse platform

Data collection for the air gesture digit recognition

Loading the dataset in Edge Impulse Platform

Setting up the development framework and design of neural network classifier

Model training in Edge Impulse platform

Model testing with the collected data

Model deployment in Nano RP2040 board

Inferencing/Prediction of results with RP2040

Conclusion

Key facts

Questions

References

7. Advance Implementation with TinyML Board

Introduction

Structure

Objectives

NDP101 Architecture

NDP120 Architecture

Practical implementation and deployment

Creating a project

Uploading Data

Impulse Design

Epochs Setting

Learning rate setting

Validation data set setting

Auto balance setting

Data augmentation

Neural network architecture

Neural network training

Model testing

Deployment

Conclusion

Key facts

Questions

References

8. Continuous Improvement

Introduction

Structure

Objectives

Expectation gap

Unique issues about audio application

Raw neural network output and softmax transformation

Handling anomalous behavior during target classifier testing

Method 1: Running window averaging

Method 2: Enriching target classifier

Method 3: Enriching open set classifier

False Acceptance Rate testing

Optimization of window size in running window averaging

[Phrase recognition constraints to improve system level performance](#)
[FRR testing under noisy conditions](#)
[Improving FRR performance under noisy conditions](#)
[Data collection for continuous improvement](#)
[Conclusion](#)
[Key facts](#)
[Questions](#)
[References](#)

9. Conclusion

[Introduction](#)
[Structure](#)
[Objectives](#)
[Review of material covered in this book](#)

[Chapter 1](#)
[Chapter 2](#)
[Chapter 3](#)
[Chapter 4](#)
[Chapter 5](#)
[Chapter 6](#)
[Chapter 7](#)
[Chapter 8](#)

[Advanced topics](#)

[Different types of neural networks](#)
[Neural network optimization](#)
[Zero-shot, One-shot or Few-shot learning](#)
[Federated learning](#)

[Transfer learning](#)

[Tuning pretrained networks](#)

[MLOps](#)

[Key facts](#)

[Questions](#)

[References](#)

Index

CHAPTER 1

Introduction to AI

Introduction

Artificial Intelligence (AI) today has touched all our lives without us realizing it. You may have used **Siri**, **Alexa** or **Google Assistant**. Did you ever wonder how it understands speech? During international trips, one often sees facial recognition in action, in airports. It used to take a lot of time for airline agents to check passports, but now, it is as easy as simply walking through a door. This door opens only when facial recognition is done and matches the passport. When people are sick and cannot type on their phone, all they need to do is use the voice assistant that comes along with all smartphones nowadays.

AI has become an integral part of many organizational ecosystems, not just at the consumer levels, which has brought forth many benefits such as increasing efficiency, and automating multiple tasks, while reducing installation and setup costs. For example, most machines which were installed in the last several decades, have an analog display. To replace all the monitors with digital meters is a nebulous task. Now, an image detector is placed on top of these displays, which can recognize the position of the needle and interpret the measurement in digital form. The information can be sent to the master control room via wireless protocol, such as Wi-Fi, Bluetooth, **Long Range Radio (LoRa)** or even **Narrow Band IOT (NB-IOT)**. Refer to [Figure 1.1](#) for an illustration of the same:

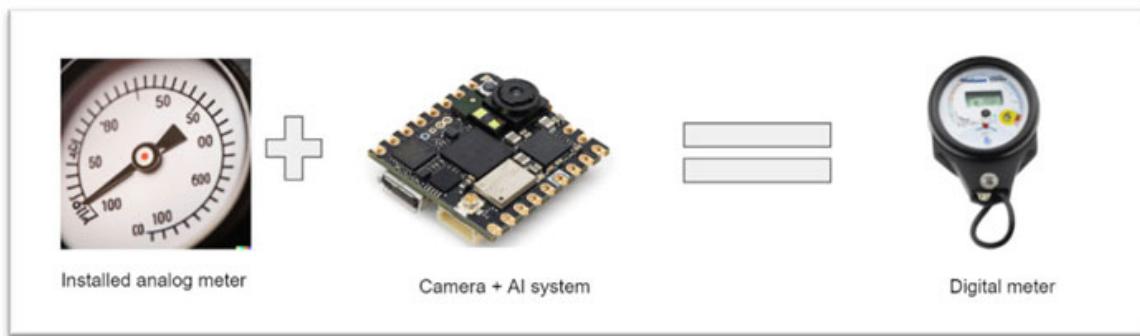


Figure 1.1: Transforming analog to digital with AI

Structure

In this chapter, the following topics will be covered:

- Artificial Intelligence
- Continuum of code writing and artificial intelligence
- Changing the paradigm
- Neural Network
- Machine Learning
- Intelligent IoT System vs. Cloud based IoT system
 - Arduino Nano 33 BLE Sense board
 - TinyML and Nicla Voice board
- TinyML Ecosystem
- Key applications for intelligent IoT systems

Objectives

By the end of this chapter, the reader will be able to relate what Artificial Intelligence has to offer. They will be familiar with the common lingo needed to bring an idea utilizing AI to a real system. Readers who are already doing firmware and software programming for the IoT devices can relate how their work will change when they plan to apply AI in their system. A concrete example is presented which shows where traditional methods will reach their limitations and AI deployment will be an easier path.

Artificial Intelligence

The intelligence demonstrated by computer systems is termed as artificial intelligence (Reference AI), as compared to natural intelligence demonstrated by living beings. The term **intelligence** could be controversial because as of today, the demonstrated capability of machines is still nowhere close to human intelligence. For this book, we will use the word Artificial Intelligence in the context of computers solving unique problems, which otherwise is not practical to solve with traditional code writing.

Continuum of code writing and artificial intelligence

It is expected that the reader is familiar with writing computer code. It can be argued that the problem which artificial intelligence solves, can also be solved with traditional computer code writing. However, the purpose of this book is to

show that sometimes, seemingly simple problems could be very difficult to solve by traditional code writing. To appreciate the value of artificial intelligence, a hypothetical problem is posed here. Let us say an image of 200x200 pixels could contain a single line or could contain a circle. To simplify, let us assume the image is only of white or black color, as shown in [Figure 1.2](#):

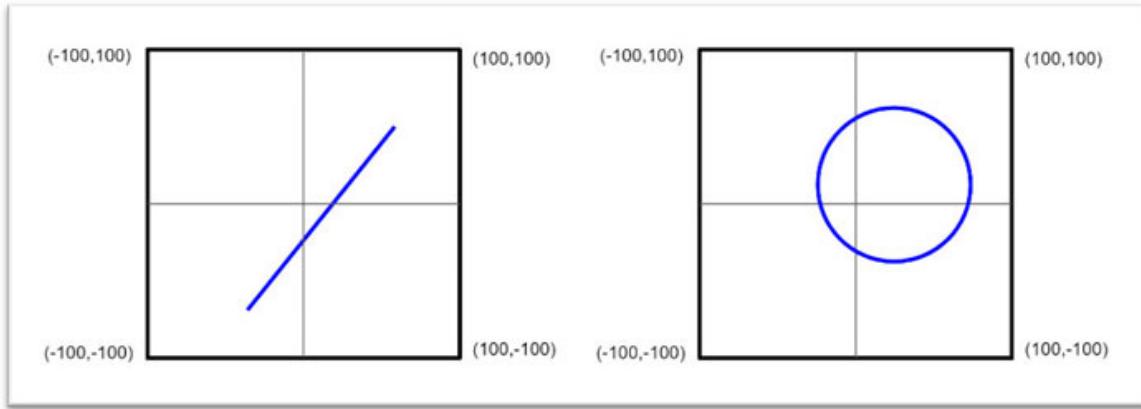


Figure 1.2: Image containing single line or circle

Exercise

Follow the given steps to perform the exercise:

1. Generate a 200x200 matrix with (0,0) points as origin, with value 0 (representing white space) or 1 (representing a dot in the curve).
2. Make multiple instances with lines of slope, ranging between +/-1 and y axis intercept of +/-50, as shown in [Figure 1.3](#):

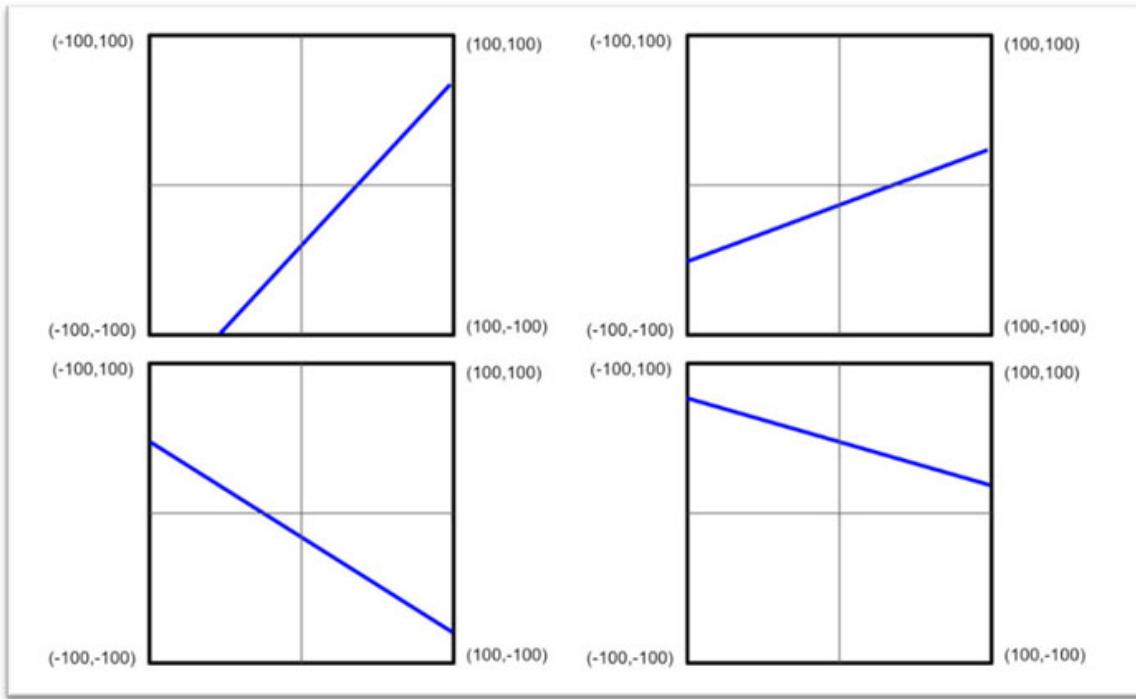


Figure 1.3: Instances with lines of different slopes and y intercepts

3. Similarly, make multiple instances of circles which fit completely within the image. Choose circles with radius of 10 to 100 and center within $+/-50$ units of $(0,0)$ coordinate, as shown in [Figure 1.4](#):

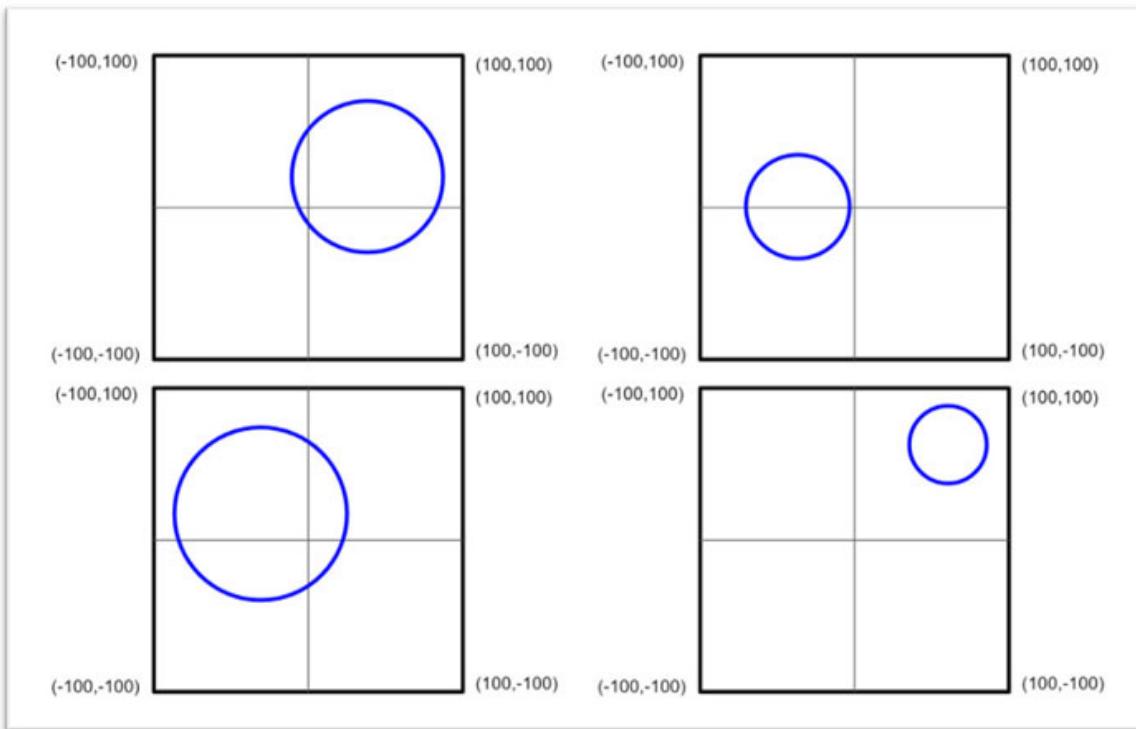


Figure 1.4: Instances with circles

4. Then, write a code with traditional logic and classify if this is line or circle.
5. Now extend the code to determine 0-9 digits in 28x28 pixels, using MNIST data Reference MNIST, as shown in [Figure 1.5](#). If it takes more than a month to write a code to successfully recognize over 90% accuracy, then the user will appreciate the advances in artificial intelligence. The artificial intelligence methodology can find a solution with over 97% accuracy in much shorter coder's time. Please refer to the following figure:

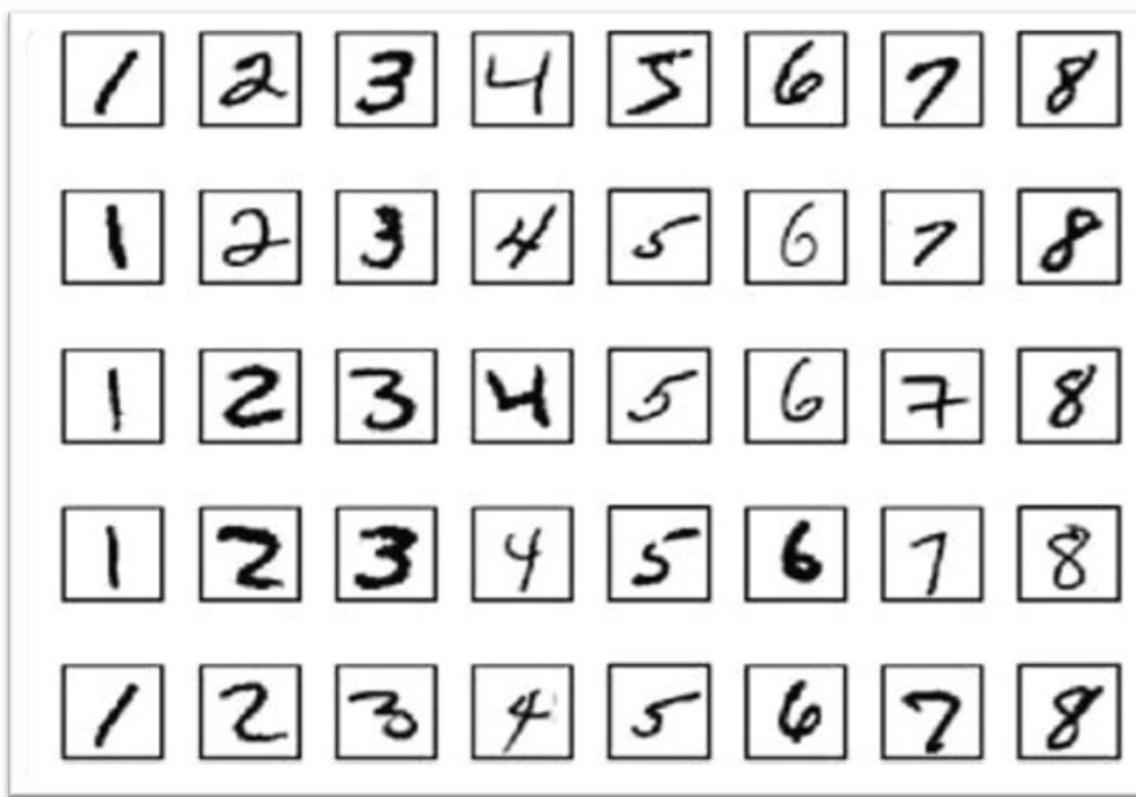


Figure 1.5: Sample images of MNIST dataset

In an artificial intelligence flow, the code is written once without analyzing a particular problem. The code uses already compiled libraries. Tensor flow library which is developed by Google is one such library. The user can scale the model with thousands to billions of variables which are also known as parameters. These variables are optimized during a process which is termed as a training aspect of machine learning. Thousands of data sets are required to train the system. Once the parameters are optimized, the test patterns are fed, and the classifications are checked. The test patterns are not part of the training set.

Changing the paradigm

As you may have noticed, code writing is automated at the expense of needing a lot of data for training. As the problem becomes more convoluted, it is not easy to write a traditional code even by a seasoned engineer. Writing a software which determines a face may be trivial to a seasoned engineer. However, writing a software which determines the age of the person without obvious clues, such as facial hair, is not trivial.

If the data is governed with simple laws or rules, then it is hard to justify use of artificial intelligence flow. For complex and obscure problems, such as age recognition, artificial intelligence is well suited.

Readers may be curious to know how AI programs are written. Let us consider the program which can distinguish between lines and circles, and then make it a more realistic problem where not all the points are strictly following one line or a circle. Let us consider the input image as shown in [Figure 1.6](#):

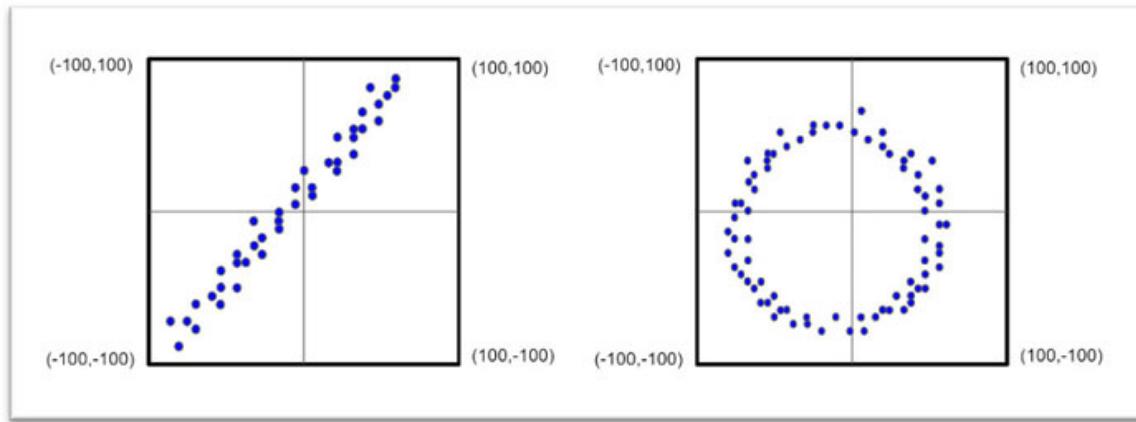


Figure 1.6: Input image where points do not follow one line or circle

As we know, it takes two points to define a line and three points to define a circle. Thus, we could use the same logic to define a line or circle, using specific chosen points. However, that will not be utilizing all the information, and results will also be different if different points are chosen, as shown in [Figure 1.7](#):

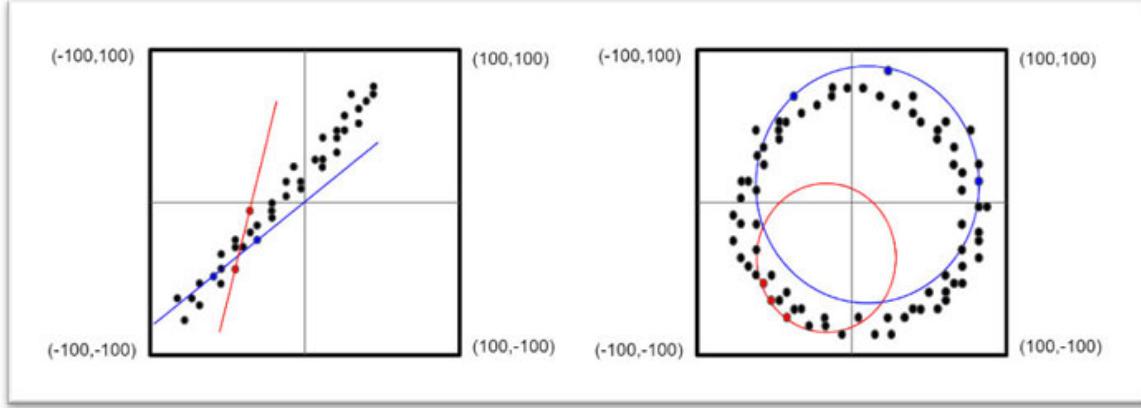


Figure 1.7: Multiple lines or circles can be estimated if only subset of the points used

For a robust solution, statistical regression methods should be used, which minimizes root mean square distance of all the points. All the data provided will be used, thus providing a robust solution. [Figure 1.8](#) illustrates the best fitting curve which is not dependent on few points but all the points:

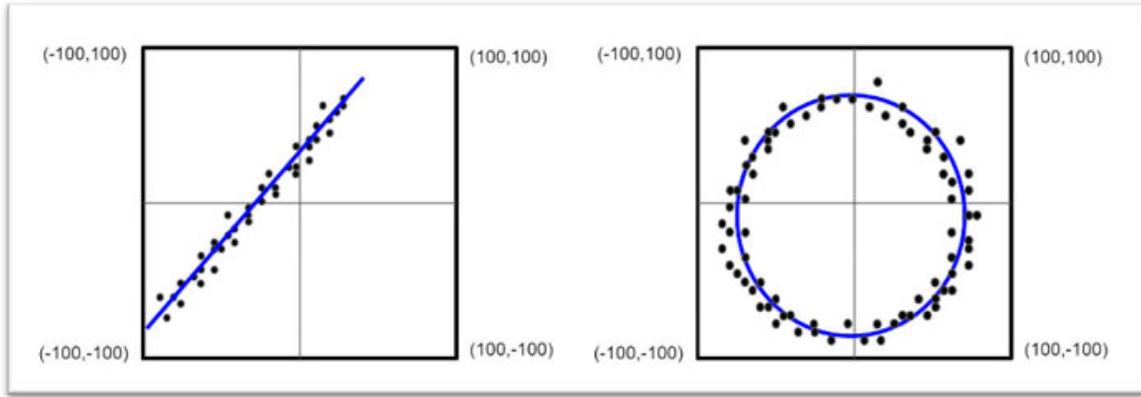


Figure 1.8: Using regression method to find best fitted line and circle

As you know, a line is specified as

$$Y = mX + c$$

where parameters, m and c are slope, and y intercept of the line respectively. Similarly, a circle can be written as

$$(X-X_0)^2 + (Y-Y_0)^2 = r^2$$

where three parameters, X_0, Y_0 and r define the circle. (X_0, Y_0) is the origin of the circle and r is the radius.

We can extrapolate to have several parameters to define a complex shape. The function can be defined with a set of multivariable linear equations which go

through a non-linear function. We can cascade such linear and nonlinear functions to form a complex function.

To solve a generalized problem pattern recognition, the study of the biological brain has inspired a new type of processor. A biological brain contains many neuron cells which are connected to each other, making a network. It is believed that electrical signals pass through the neurons and eventually interpret a pattern. This processor is named as a neural network which indicates its origin. Let us look at the neural network and how it resembles the biological brain.

Neural Network

Most of us can guess people's age at a glance within reasonable accuracy. It comes effortlessly because this is how our brain works. Scientists got inspired from the anatomy of the biological brain, which is made of neurons. Neurons relate to multiple other neurons, which may seem a random connection. However, as the signal passes through these neurons, living beings can make rational decisions. Refer to [Figure 1.9](#) for an illustration of the biological neuron:

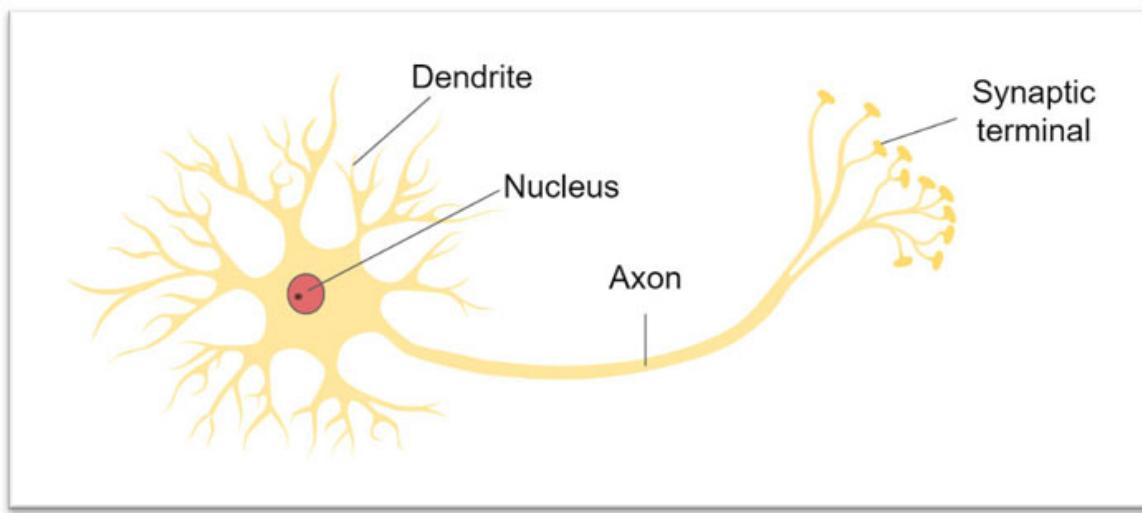


Figure 1.9: Illustration of a biological neuron

[Figure 1.10](#) shows how multiple neurons are connected, thus forming a neural network. The bond between two neurons is called synaptic bond. It is believed that these synaptic bonds are being made over time. The synaptic bonds could have different connection strengths which would pass proportionate information. However, it may not be obvious how the simple connection between neurons suddenly would possess intelligence. A mathematical model made on similar principals are developed and that forms the basis of artificial intelligence. Please refer to the following figure:

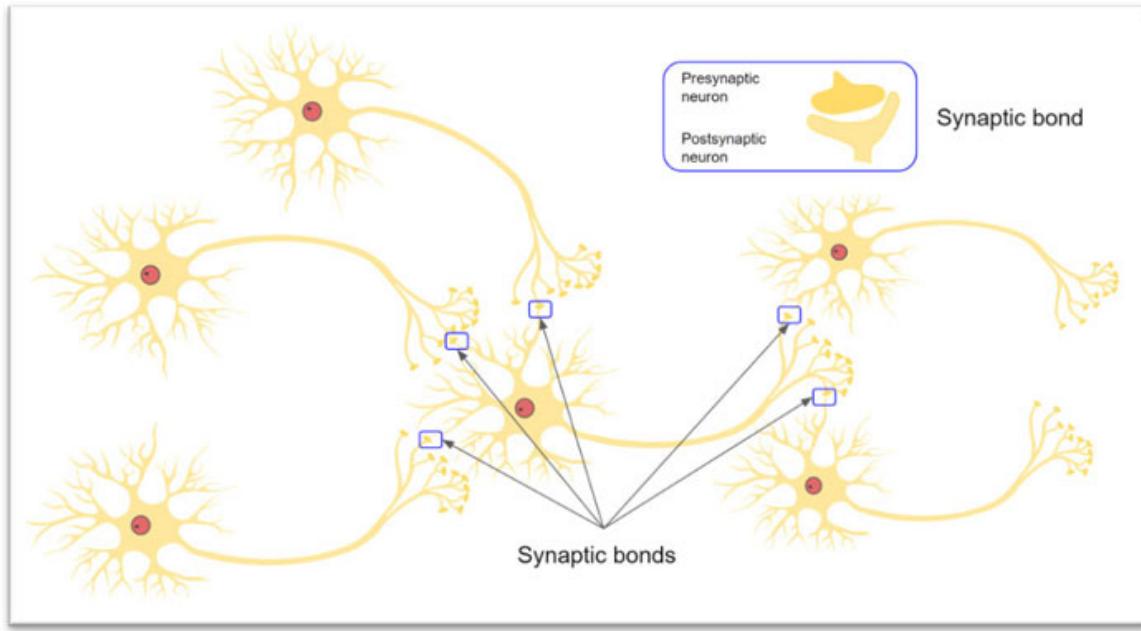


Figure 1.10: Illustration of a biological neurons forming neural networks with synaptic bonds

In a mathematical representation, the neuron simply sums the signals coming through the synaptic bonds and passes the signal to the next neuron. [Figure 1.11](#) draws a parallel between the biological neural network to a mathematical neural network. It shows how mathematical neurons are mimicking a biological neuron and how synapse connections are replaced by weights:

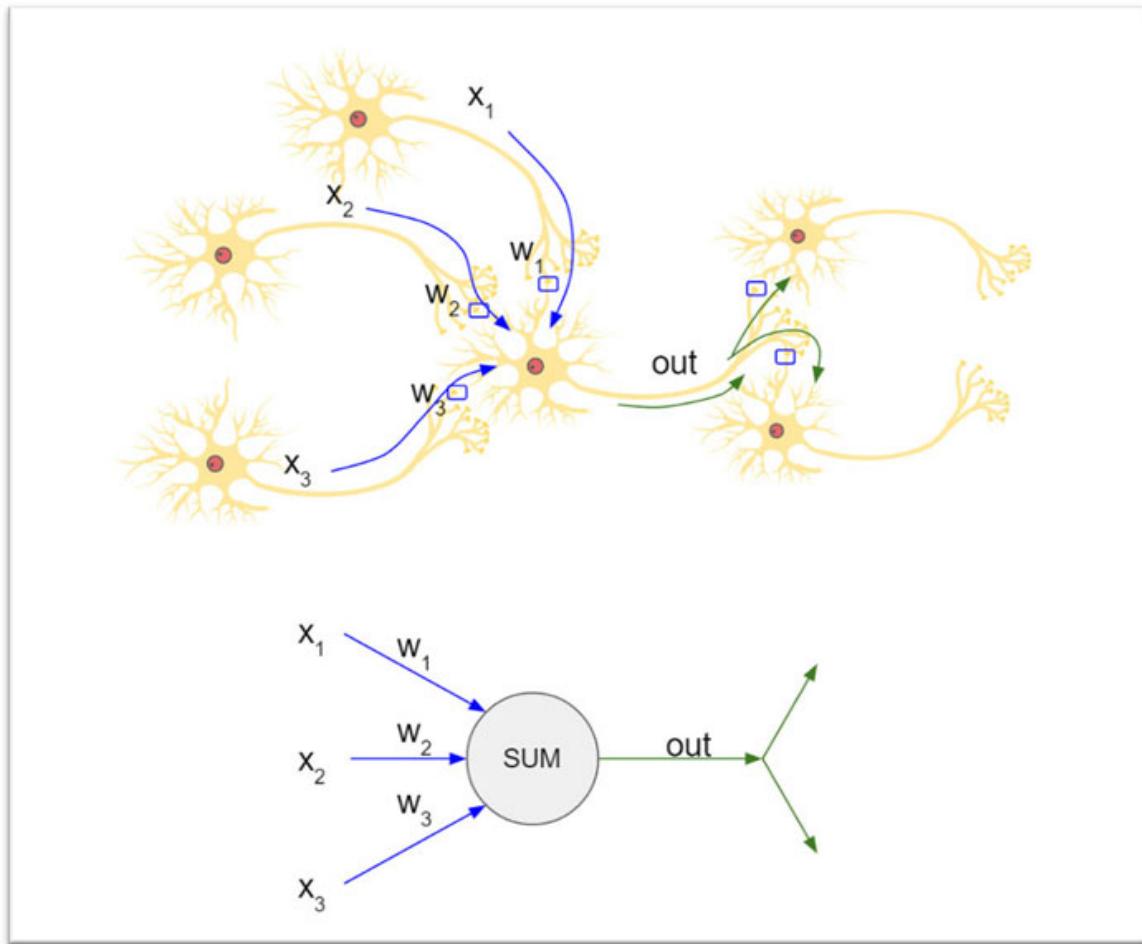


Figure 1.11: Parallel between biological neuron and mathematical neuron

After several tries over the years, many structured networks have been developed. The easiest network is defined as a fully connected neural network which is also termed as Dense Neural network. In a fully connected neural network, all inputs are applied to all neurons. The weight of the neuron is equivalent to the strength of the biological neuron. If there is no connection between two neurons, then the weight can be 0 in a mathematical neuron.

If one parameter is equated to one synapse of the human brain, then it is estimated that it will require several hundred trillion of parameters (reference Trillion). Let us approximate the number of parameters to be 1000 trillion parameters. Assuming a typical RAM of a computer is 8 Giga Bytes, a human brain is equivalent to 1,25,000 of such laptops. A typical data center can have 1 million to 10 million servers, which can be shown to have more capacity than one human brain. So, as of today, it is not impossible to mimic the human brain in a data center. It will be a while before full emulation of human brain will be economical and widely used.

However, the number of parameters used in artificial intelligence is growing at an exponential pace. A linear-log plot shows how the number of parameters has grown since 1952 to today, as shown in [Figure 1.12](#):

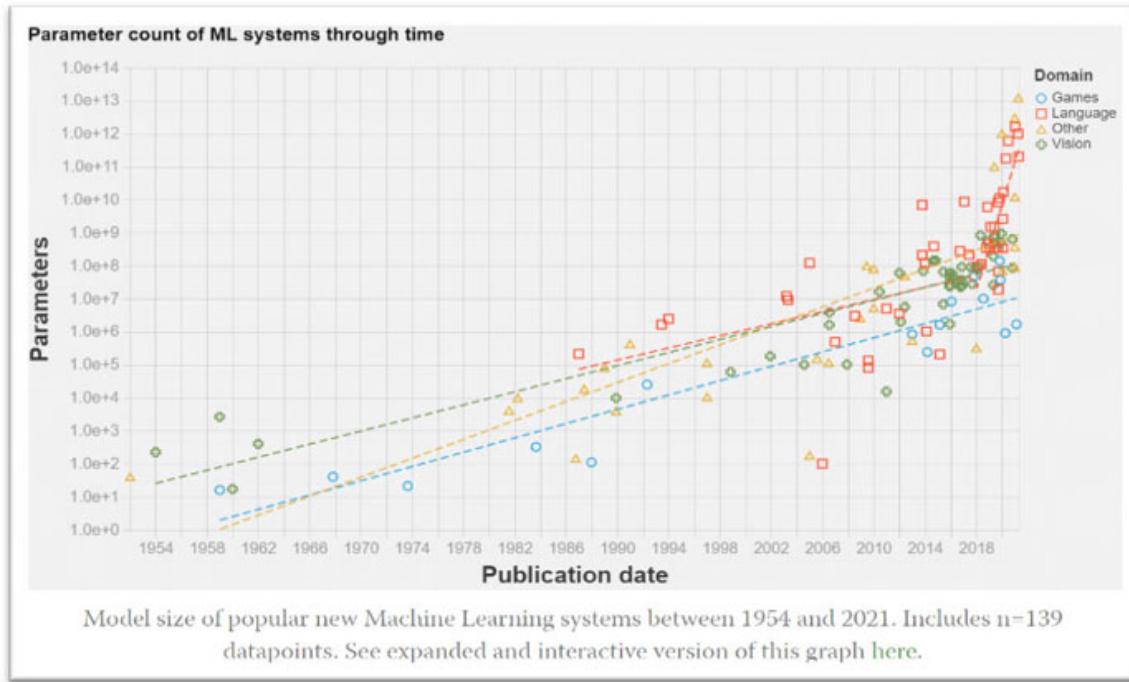


Figure 1.12: Model size of popular new Machine Learning systems between 1954 and 2021.

Even in the linear-log curve, the plot looks exponential towards the end, showing that the growth is faster than mathematical exponential growth. As of today, the highest parameter neural model in Google search shows a 175 billion parameter model, named as OpenAI LLC's GPT-3 natural language processing model (Reference GPT3). This model still is only 1/5000 of the human brain.

It is not sufficient to just have a neural network to recognize a pattern. Even in the biological world, it takes years to train the brain. Similarly, a neural network needs to be trained. As mentioned earlier, a neural network is defined with parameters. The parameters are like variables which are placeholders for a number. For different applications, the numbers will be different. The process of finding a set of these numbers comes under machine learning. Let us take a deeper look at how machines learn.

Machine Learning

We now see that there are thousands of parameters present in a neural network. These are like placeholders of m and c in a line representation of $Y = mX + c$. Before making a specific model, the values are typically initialized as random

values and are normalized between +/-1. Sometimes, the initial values can be taken from the previously developed model.

There are several methods for how these variables are optimized. **Backward propagation** of errors (in short backpropagation method) is one of the most popular methods. These are deployed inside TensorFlow or similar programs used for machine learning. Most users do not need to understand the mechanism of how it works, as these are work horses working under the hood.

However, a few things should be understood. By default, the training starts with random variables. These variables are tweaked in each iteration, which is termed as **epochs** in machine learning jargon. It can take anywhere between 10 to 1000s of epochs to converge. If inputs are well defined and are similar, the convergence can be robust and fast. However, if the data is very noisy, ill conditioned or has data poisoning (mislabeled data), then the convergence will be slow and at some times, it may not converge at all.

The convergence is described with two variables: **Accuracy** and **Loss**. Accuracy is determined by how many training sets are passing. Typically, 90%-99% accuracy is reasonable. In the classical classification approach, one of the classifiers is forced to be matched using the **softmax** formula. For example, if there are only two classifiers, and predicted values come out to be 0.4 (wrong classifier) and 0.6 (correct classifier), then it will be considered accurate because 0.6 is larger than 0.4. Loss is a measure of how close a match is found. In the preceding case, though it was accurate, the class which should have been 0, is instead 0.4, and the class which should have been 1, is 0.6. The error (0.4) is a measure of loss. Ideally, loss should be 0, but 0.1 value is also reasonable. Even for casual users of machine learning, it is expected that they keep an eye on accuracy and loss value. By analyzing how quickly accuracy reaches the 90% mark and loss of 0.1, the quality of the model can be predicted. It can also be used to estimate the quality of data.

[Figure 1.13](#) shows typical convergence with successive epochs. We see that the accuracy increases and loss reduces, reaching to asymptotic values:

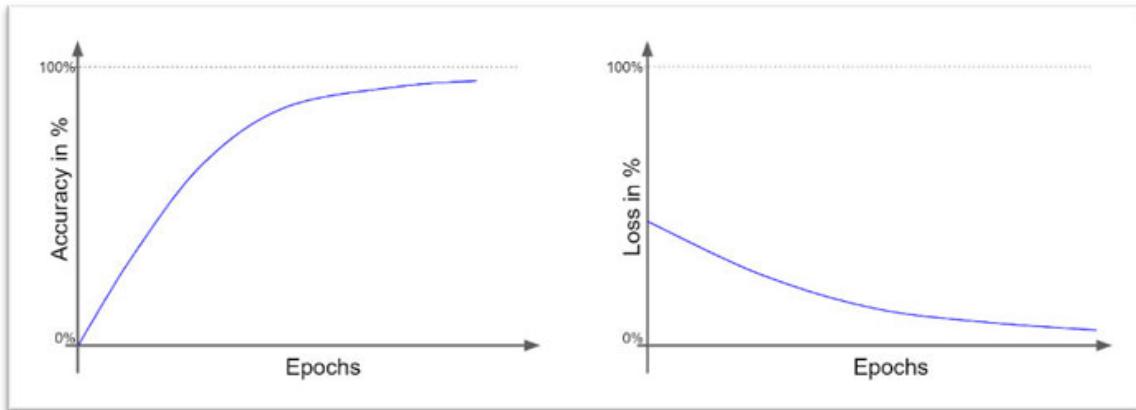


Figure 1.13: Monitoring convergence during Machine Learning process

By default, each time the training starts, it starts with random variables. So, it is almost certain that the models generated will have nothing common between two tries, if matched parameter by parameter. However, the net result on accuracy and loss will be close (within 1%-5%). This explains that there will be multiple solutions. One may run multiple runs and choose the best model at the expense of more compute power.

Sometimes, it is advantageous to start from an existing model. This will provide more consistent results but it will be a “biased” solution from previous data and solutions. Machine learning is an empirical solution, meaning, it encourages more experiments at the expense of compute power. One methodology which might have worked for one type of problem, may not be optimum for another. Thus, when in doubt, try different ideas and settings. To save time, multiple experiments can be distributed on different machines. In this way, you can get the best answer at the same elapsed time, at the expense of compute cost.

Now that we have a trained neural network which can interpret the data, it can be deployed. Deployment is also a very compute intensive job. Bigger and complex neural networks can take a significant amount of compute power, thus requiring expensive compute platforms and associated power needed. With the growth of the world wide web, data centers emerged with cloud computation. Thus, the deployment of the artificial intelligence payloads started in the cloud. However, in some applications, it makes sense to compute right at the edge. In this book, we will focus on deployment of the artificial intelligence in small devices known as **Internet of Things (IoT)**. In the next section, we will discuss the main motivation why computation in the device itself, is preferred over the cloud computation.

Intelligent IoT System vs. Cloud based IoT system

By definition, IoT systems are internet connected devices. Most of us are used to the very high speed of the internet, and thus we do not feel any issue with providing high speed internet connectivity to all the devices. At the same time, we expect the IoT devices should be taking small amounts of power, be untethered and run-on small batteries. This places a conflict between high-speed connectivity and low power expectation. Most IoT devices are connected with **Bluetooth Low Energy (BLE)**, **Long Range Radio (LoRa)** or **Narrow band IOT (NB-IOT)** protocols. All these protocols have a very low bit rate, probably lower than the dial up modem which we struggled with, back in the 1990s.

If the internet bandwidth is not an issue, then collecting all the data in the cloud and running the inferences will be just fine. As a matter of fact, that is how most IoT systems run as of today. This basic limitation is the main reason why adaptation of AI on IoT devices is slow.

With recent advances, AI workload can now run-on low power remote devices. This is the fundamental topic of this book. With built-in intelligence, it is safe to say that these devices should rather be called **Intelligent Internet of Things (IIoT)**. It is also possible that some of these devices are not even connected to the internet, for example, a device which is opening and closing the door to a voice command, does not need to be connected to the internet. In that case, we may just call these devices as Intelligent Things.

Power is not the only reason why IIoT devices need to exist. Privacy is a key concern as well. Most people are not comfortable having their voice and image data going to the internet all the time. The Intelligent Things terminate the data right in the device and can be strictly not connected to the internet, thus leaving no way of leaking their privacy. It will be creepy to find out your voice activated lamp is transmitting your voice data to the cloud or the built-in camera in your TV is watching you, as much as you are watching TV.

Latency and reliability are yet more concerns. In the internet protocol, the information is packetized and goes through several devices before it reaches its destination. Unlike old school telephony which worked on dedicated lines, the internet-based protocol sends a packet with finite guarantee on arrival time or arrival. Even in the year 2022, we see choppy voice connection in video conferences, even after the Covid days where the internet was tested to its limits. For mission critical situations, or situations where fast reaction is required, cloud based artificial intelligence is not sufficient. One such example is autonomous vehicles.

To avoid dependency on cloud computation, it is possible to compute locally. Most IoT systems are designed around a **Microcontroller Unit (MCU)**. An MCU is essentially a computer chip which you will find in a personal computer with

some subtle differences. The speed at which the operations are done, is quite low in a typical MCU. Lower clock rate is amenable to lower power. Typically, MCU have significantly low on chip **Random Access Memory (RAM)**, and there is no provision for external RAM. So, these microcontrollers provide limited compute power at low power and low-price points. While the compute power is limited, recent innovations in machine learning are enabling these small microcontrollers to be able to run AI payloads. As a matter of fact, a class of these devices and applications are referred to as **TinyML**.

There are many microcontroller boards available which can run TinyML payloads. *Arduino Nano 33 BLE Sense*, *Silabs thunderboard 2*, *Texas Instruments' LaunchXL-CC1352P*, *Synaptics Katana KA10000*, and *Syntiant TinyML board*, are just a few examples. In this book, we will focus on a few boards which are available under \$100.00. Some solutions are based on MCUs and others are specialty solutions. From the boards which are based on MCUs, we have chosen Arduino Nano 33 BLE Sense, that is built around Nordic's nRF52480 MCU. From the specialty boards, we have chosen Syntiant's TinyML board and *Arduino's Nicla Voice boards*, built around *Syntiant's Neural Decision Processor* technology. The following section provides deeper insight into these boards.

Arduino Nano 33 BLE Sense board

There are many boards where an MCU is used, which have reasonable resources to put on Artificial Intelligence. One of the very popular boards is Arduino Nano 33 BLE Sense, shown in [Figure 1.14](#). It is readily available under US\$100.00 from multiple sources. Please refer to the following figure:

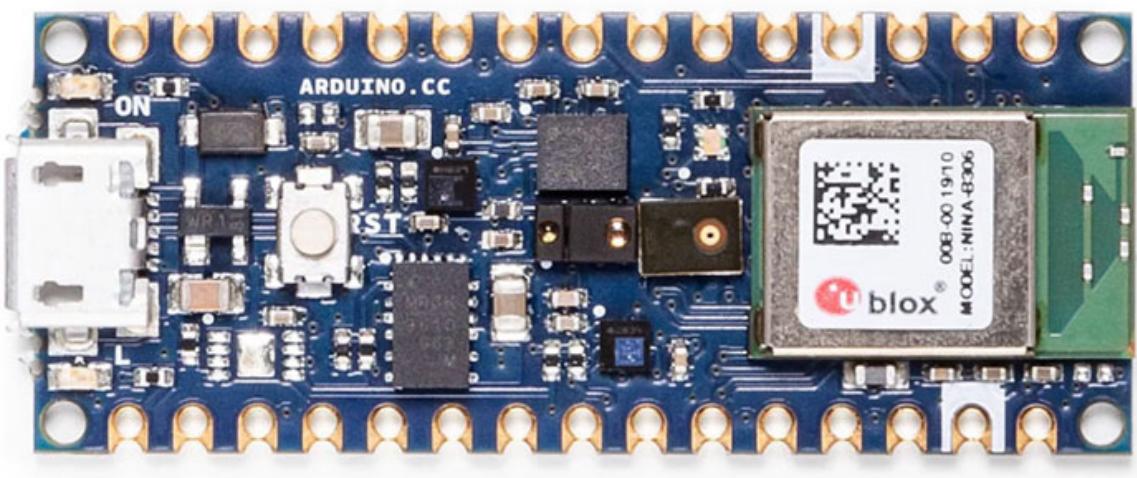


Figure 1.14: Arduino Nano 33 BLE Sense

[Figure 1.15](#) shows the diagram of Nano BLE Sense 33:

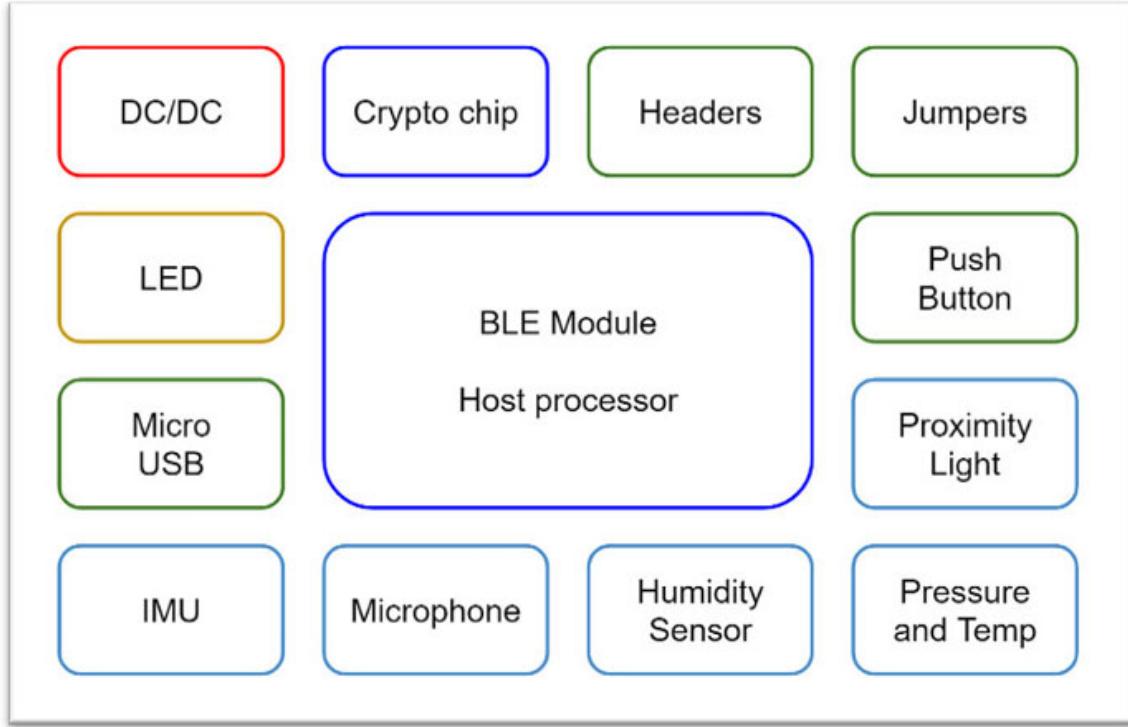


Figure 1.15: Block diagram of Nano 33 BLE Sense

The ideas of Artificial Intelligence, Neural networks and so on, were conceived almost at the same time when computers were developed in the 40s and 50s. However, it saw several “winters” before it caught on in the last two decades. The main reason behind latest popularity of artificial intelligence is the availability of server grade compute power which is needed by the neural networks.

IoT systems running on microcontrollers have the same issues that were faced two decades ago, when the cloud computing was not available. If a voice-controlled faucet requires a server grade computer, then it will be an unviable/impractical product. Despite the MCU technology being very useful, there are two main limitations to this, which will be discussed as followed.

Limited compute resources

The RAM is generally quite limited in most MCUs and thus the models are very small. The throughput is also limited due to limited clock rates.

Battery power limits

In many cases, the batteries are expected to last over months. While the power dissipation of MCUs is much lower than single board computers, for example,

Raspberry Pi, the power is not low enough for many battery power applications.

Another wave of technology is becoming available which is implementing neural circuits, either in analog domain or in digital domain. These purpose-built chips are rivalling single board computers in terms of total compute resources and throughput. So, now we can get to see good size neural networks present in IoT systems. This is a good segue into talking about the boards which are based on Syntiant's Neural Decision Processor.

TinyML and Nicla Voice board

A TinyML board which is readily available, incorporates NDP101. The block diagram of the TinyML board is shown in the following [Figure 1.16](#):

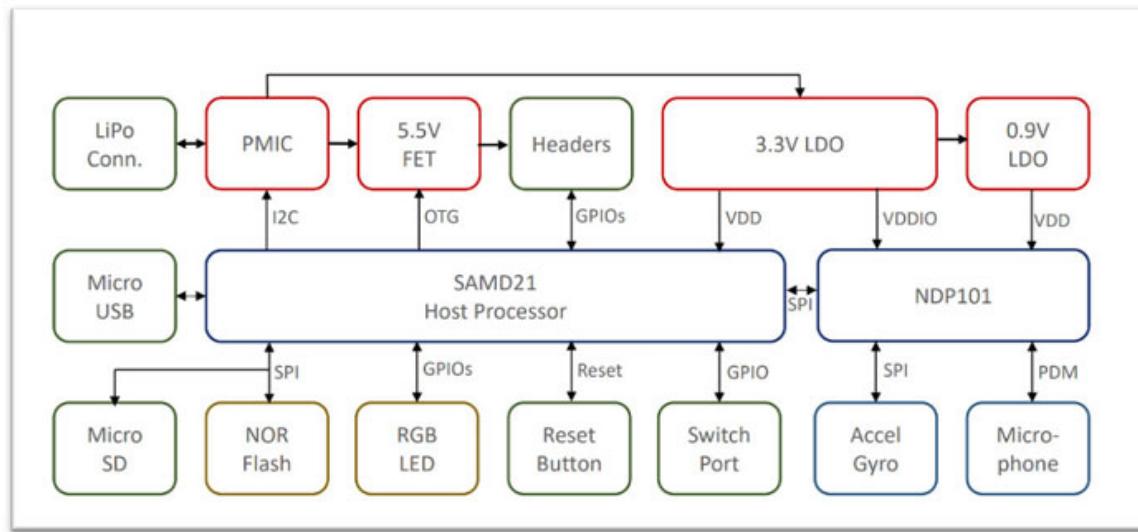


Figure 1.16: Block diagram of TinyML board

The TinyML board, apart from incorporating NDP101, also has SAMD21 processor working as the host processor, as can be seen in [Figure 1.17](#):

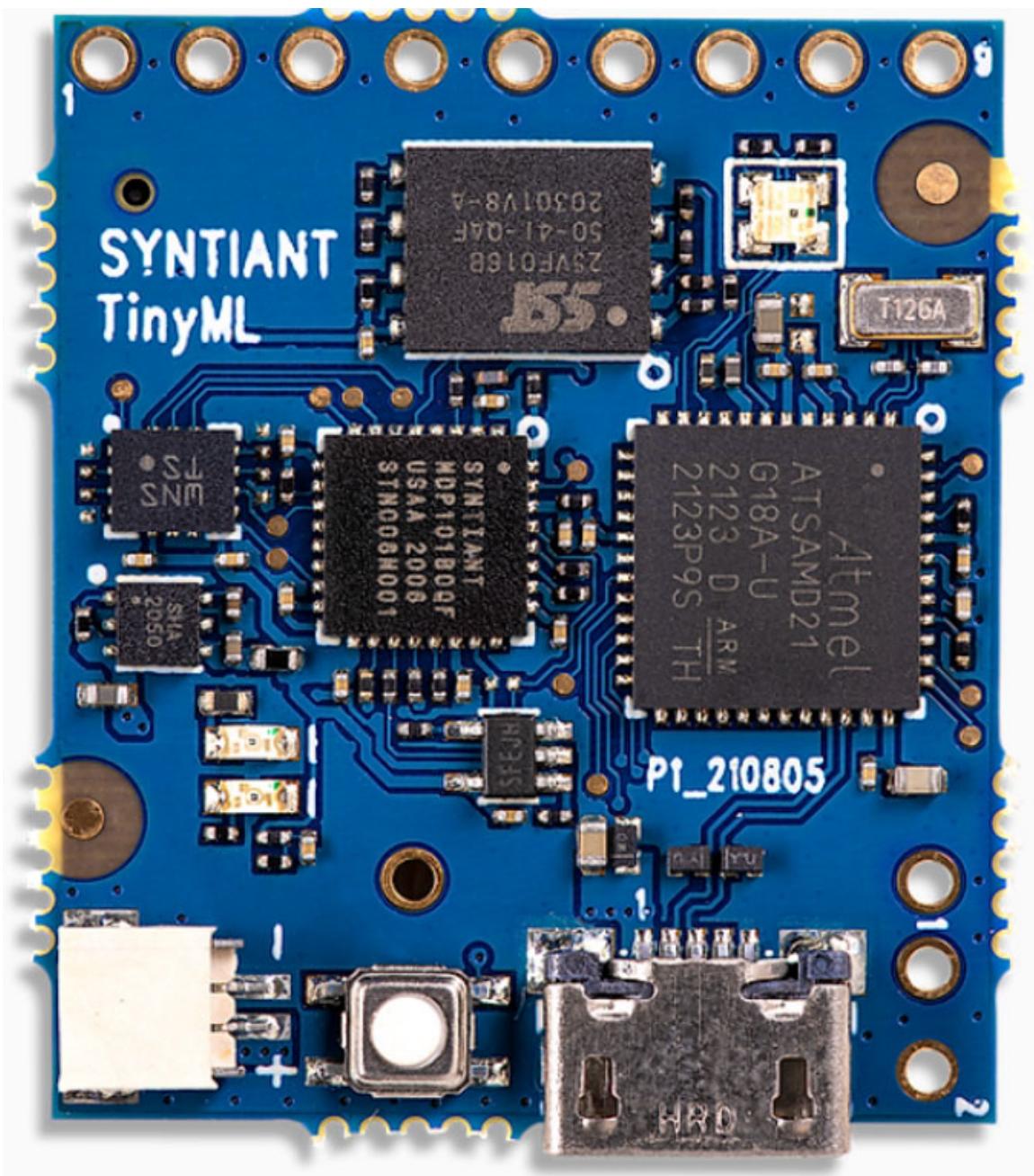


Figure 1.17: Syntiant Core I technology in TinyML board

While TinyML board has been available since late 2021, the Syntiant's second generation chip, NDP120 based system has been recently launched by Arduino. The Nicla Voice board is readily available. The block diagram of the Nicla Voice board is shown in [Figure 1.18](#).

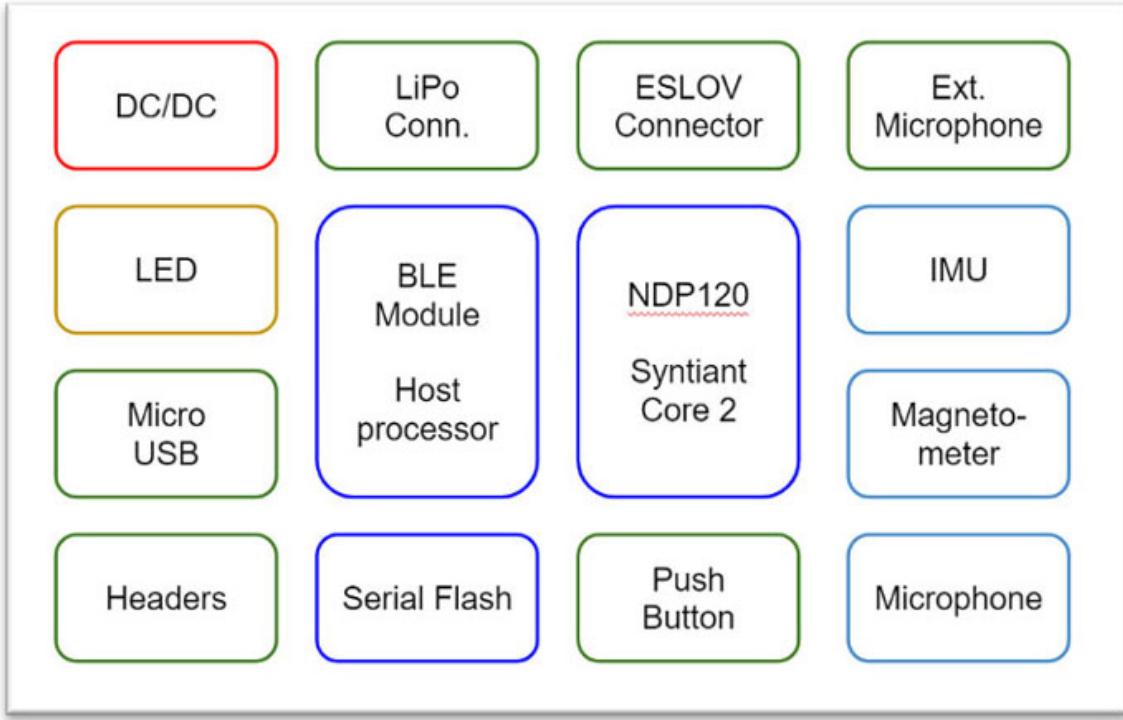


Figure 1.18: Block diagram of Arduino Nicla Voice board

[Figure 1.19](#) shows the picture of Nicla Voice board.



Figure 1.19: Syntiant Core 2 technology in Arduino Nicla Voice board

There are three main benefits of Syntiant's purpose built AI chips over other solutions.

>10x parameters

Typical AI models use hundreds of thousands of parameters. The typical RAM in MCUs, range between 32kB to 64kB, while the RAM available in TinyML board is roughly 256kB. It has 4 bits of precision, which is more than sufficient in many use cases. The board can use this memory to support over 500k parameters, which is over 10x than competing solutions.

>200x Power advantage

Whether it is a **Complex Instruction Set Computer (CISC)**, or **Reduced Instruction Set Computer (RISC)**, the computation flexibility is achieved at a very high cost of power inefficiency. For a simple addition of two variables, the variables are moved around in the chip dissipating a lot of power. Syntiant chips are designed ground-up as a neural compute engine. This architecture optimized the data flow right to the transistor level and thus saving power. Several architectures have been benchmarked and have shown >200x power improvement, than same class MCUs.

>20x Throughput

For good latency and fine grain analysis of input, more throughput is required. For example, a simple wake word such as “*Alexa*” takes just a little under 1 second. Even after a lot of optimizations, most MCUs in the same class can analyze 1 to 2 frames per second. As shown in [Figure 1.20](#), the analysis window may not line up with the actual word and consequently the keyword may not be detected. Please refer to the following figure:

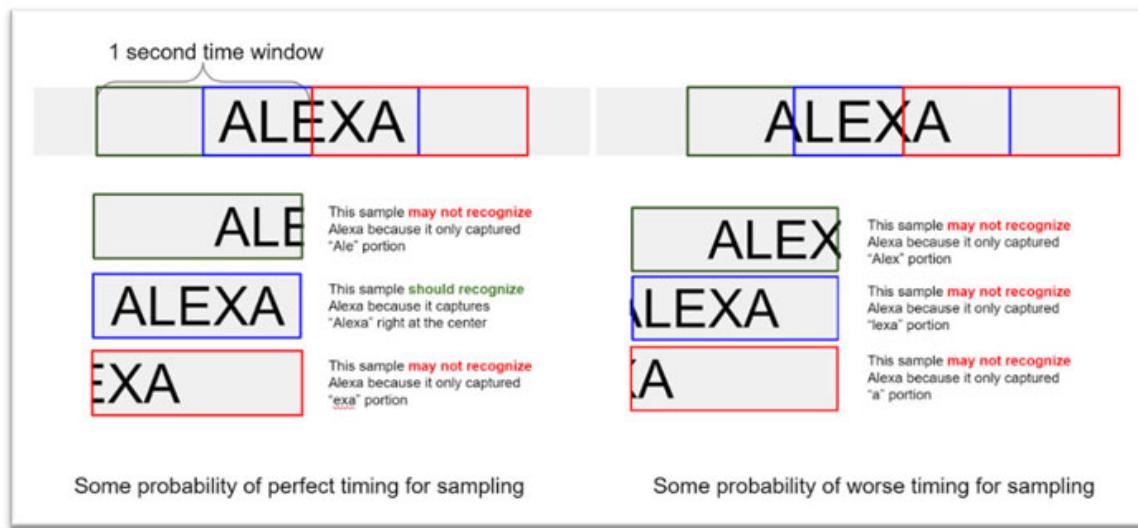


Figure 1.20: Two inferences per second could hit or miss the wake words

Syntiant’s Neural Decision Processor chips typically process inferences 40-100 times a second. This oversampling provides much better resolution which can be further processed, termed posterior processing, for robust field performance, as shown in [Figure 1.21](#):

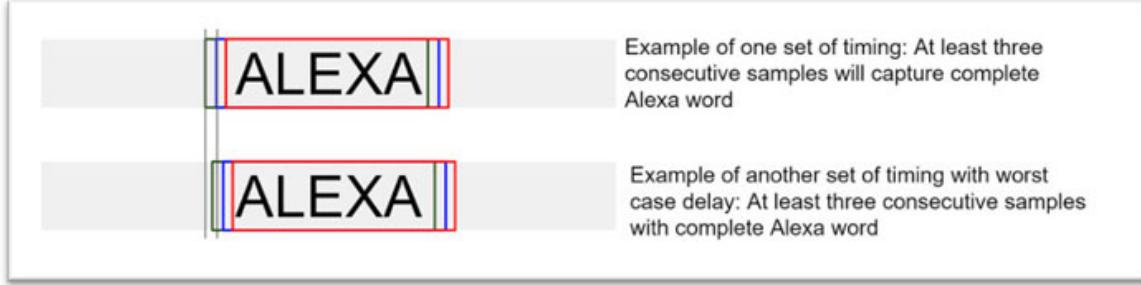


Figure 1.21: With high rate of inferences, the wake word cannot be missed

TinyML Ecosystem

A vertically integrated company with sufficient resources can collect data, make a machine learning model, and deploy it using tensor flow lite libraries on their own. However, small companies may not have all the different skills set and thus must rely on the ecosystem. An ecosystem that evolves around TinyML devices, will streamline the complete AI systems development at the edge, right from collecting the data to the eventual AI model deployment. The purpose of this book is to show how to leverage this ecosystem. It will also be shown how readers can write their own code to fill in the gap. It is expected that more and more companies will be working in tandem to provide data, data sanity utilities, machine learning automation and automatic code generation for deployment. *Edge Impulse*, *Imagimob*, *SensiML*, *Qeexo* and so on, are example of few companies which are in the forefront of the TinyML ecosystem. Some of the chapters written in this book are using Edge Impulse Studio. A non-profit organization, www.tinyML.org, is dedicated to bringing the TinyML ecosystem together.

Key applications for Intelligent IoT systems

The IoT market can be diced and sliced in many different manners. There are some products and ideas which may belong to multiple markets or may generate new segments of their own. However, to classify the nebulous market, the following [Figure 1.22](#) attempts to classify the IoT market:

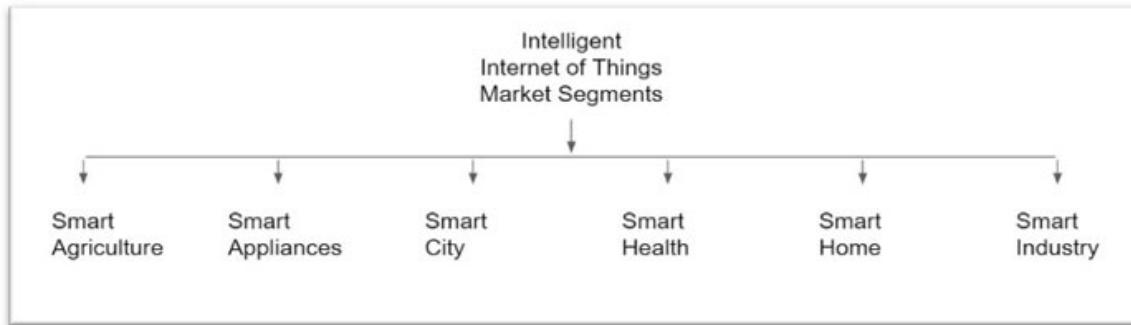


Figure 1.22: Key IoT markets

The adoption of AI for the IoT market still is in its infancy. The applications and innovations are really limited by the reader's imagination. It is expected that the readers will be coming with a lot of ideas to apply AI into the IoT devices. The purpose of this book is to arm readers with the tools, so that they can execute their ideas. Some examples are mentioned here to just spark the imagination.

Smart agriculture

As the population is increasing, the burden on agriculture is also increasing. Due to overuse of fertilizers and pesticides, the ecosystem is already crumbling. IoT devices will help in reducing the pesticide usage by applying where it is most effective. The fertilizers will be applied where they are best absorbed. Agricultural robots, such as for planting trees, picking up ripe fruits, and so on, are making use of AI for navigation and fruit ripeness detection. The humming sounds of insects are used for early detection of invasive and harmful pests. Smart farming is happening even as we speak. With image recognition, weeds are found and then they are either mechanically removed, or are destroyed by spraying weedicide right over them, thus reducing weedicide chemicals over 90%.

Smart appliances

Many IoT devices are making the appliances smart. The television can now be controlled with hand gestures or voice controls, and thus there is no need for a remote. Refrigerators can detect odors, providing early warning of rotting food.

Smart cities

AI is making cities smarter. Drones can be used to find illegal parking. Facial recognition can alert authorities about fugitives, terrorists and so on.

Smart health

Wearables are providing crucial information about our health before damage is done. Some wearables claim to predict impending heart attack or epilepsy episodes. This can not only save the person but others around them, if he or she is driving a vehicle. The sleep can be monitored and recommendations can be made for better sleep.

Smart homes

Many IoT devices found their way in our homes, for example, thermostat, power meters, earbuds, car chargers, sprinkler system, smart watches and so on. With smart devices, the air conditioning, for instance, can be managed to run only in rooms where people are present, thus reducing the carbon footprint. Imagine a scenario where pets can be fed while owners are away.

Smart industry

Industries are also benefiting from IoT devices. Old analog dials can be read by an IoT device and now the system can be regulated digitally. A change in vibration or acoustic signals is used for predictive maintenance. Advanced devices can find the location of the bar code and then read it. The tiny robots used in the warehouse can find their way using image recognition.

Similarly, drones can be used for autonomous flight using image recognition. They can find their way by recognizing obstacles in the path and then avoid it.

Conclusion

In this chapter, a brief introduction is given to what is Artificial Intelligence. The concepts of **Neural Network (NN)** and **Machine Learning (ML)**, which are the key components of AI systems) are also presented. An example is taken which shows that there is continuum of traditional programming style and the neural networks. A reader who is a seasoned computer scientist, can appreciate the problems which are better suited for AI.

Once the theoretical abstraction is presented around AI, practical deployment vehicles are presented. The reader should feel comfortable about the problems which can be solved, and in approaching towards the provisioning of a complete solution. The next few chapters will take some concrete examples and demonstrate the deployment of AI, based IoT solutions in more detail.

Key facts

- Artificial intelligence is not replacing human intelligence, but it is aiding to finding solutions where traditional programming is not practical.
- Neural networks are a generalized function with linear and non-linear functions with multiple layers and thousands of parameters. It is inspired from the biological brain but when it is implemented, it is nothing like a biological brain.
- Machine learning is a way of optimizing the parameters of a neural network, such that its output matches with the input patterns.
- The burden of code writing has been shifted to data collection and monitoring, if machine learning is producing expected performance or not.
- With recent advances in hardware and software, it is possible to deploy meaningful AI system right in the IoT systems without breaking power, cost, and size envelope.
- The ecosystem is evolving for TinyML development, thus reducing burden of detailed programming.
- There are myriads of applications where the TinyML has already been used. The purpose of this book to further enable readers to get inspired and develop complete AI systems while leveraging the ecosystem.

Questions

1. Is AI threat to human intelligence and existence?
2. What is TinyML? How it is different than mainstream machine learning?
3. Why is traditional style programming not suited for certain applications that AI is used in today? Provide two examples to prove your point.
4. For power starved edge systems, what options are available to use bigger network which will not fit in standard MCUs?
5. What will be more important in future? Code writing or enabling AI systems?
6. Come up with a use of AI to make life easy at home? Explain why the application cannot be completed using traditional coding?

References

1. AI: (https://en.wikipedia.org/wiki/Artificial_intelligence)

2. **MNIST: MNIST Dataset | Kaggle**
3. **Trillion: Synaptic Connection - an overview | ScienceDirect Topics**
4. **GPT3: GPT-3 - Wikipedia**
5. **Arduino: <https://docs.arduino.cc/hardware/nano-33-ble-sense>**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 2

Traditional ML Lifecycle

Introduction

Arthur Samuel, an American pioneer in the fields of computer gaming and Artificial Intelligence (AI), first used the term "machine learning" in 1959. He gave the definition as follows, "without explicit programming, Machine Learning (ML) enables the computers to acquire knowledge."

ML is a subfield of AI that, with the help of statistics, other disciplines and huge amount of data generated from different sensors and devices, usually develops a model which can identify interesting patterns. ML also helps in making predictions and decisions through machines by a self-learning process. Huge amounts of data from a variety of different sources, require strategies for proper processing and analysis. In general, any ML algorithm uses patterns in the data for developing predictive models, and utilizes these models for prediction. This kind of developed models can be used in identifying patterns in unseen data (known as inference) and make decisions based on this.

Over the last few years, we have seen a rapid development and growth of many ML algorithms on the **Internet of Things (IoT)**. The IoT has billions of devices connected to its network, and the sheer volume of these IoT devices allows developers to build many novel software applications. The purpose of these devices is to continuously collect and analyze the physical parameter/features of their deployment environments. This can lead to a drastic rise in the amount of data that is being generated, which in turn needs high computing performance and storage space. The integration of ML algorithms within the IoT devices makes these devices intelligent to make proper decisions with the generated data.

In literature, there are many ML algorithms (for example, linear regression, SVM, decision trees), but the focus of this book will be more on TinyML, as it helps in deploying ML algorithms on the IoT devices. In this context, we

will be exploring and focusing more on the basic concepts required for understanding TinyML.

Structure

In this chapter, the following topics will be covered:

- Traditional methods
- Machine learning landscape
- ML Performance Metrics
- Basics of DL and different DL algorithms
- Transfer Learning
- Tools and Different ML & DL frameworks
- Embedded Machine Learning
- Difference between Learning and Inference
- ML model deployment and inferencing on different platforms

Objectives

In this chapter, we will learn about the traditional ML life cycle, and get a basic introduction for ML and concepts required in implementing the ML on the embedded devices and Microcontrollers.

Traditional methods

In traditional problem-solving approaches, we first formulate a hypothesis that will explain the phenomena and only then, will we be designing an experiment to prove or disprove the proposed hypothesis. After making the experiment ready, we gather the required data that will be analyzed for accepting or rejecting the proposed hypothesis. The following [Figure 2.1](#) shows solving a problem using traditional approach:

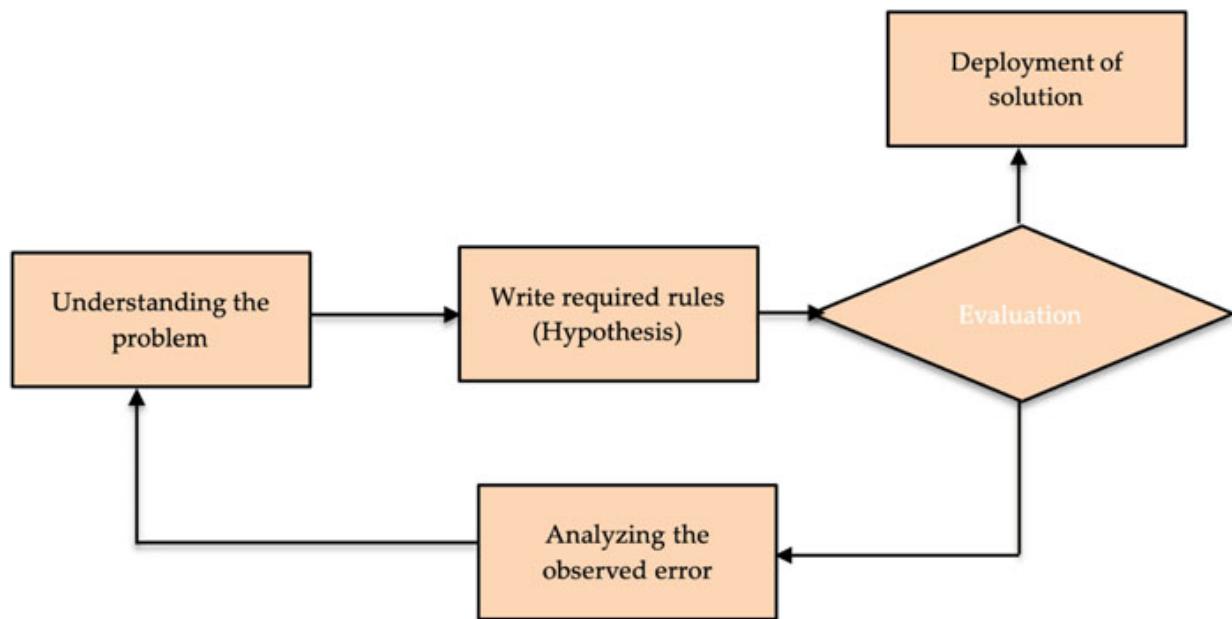


Figure 2.1: The traditional approach

Machine learning landscape

In general, the ML algorithms implement the principle of “**trial and error**” concept. These methods continuously validate and refine a model, based on the incurred loss by its predictions. Refer to [Figure 2.2](#) for a better understanding of the machine learning approach:

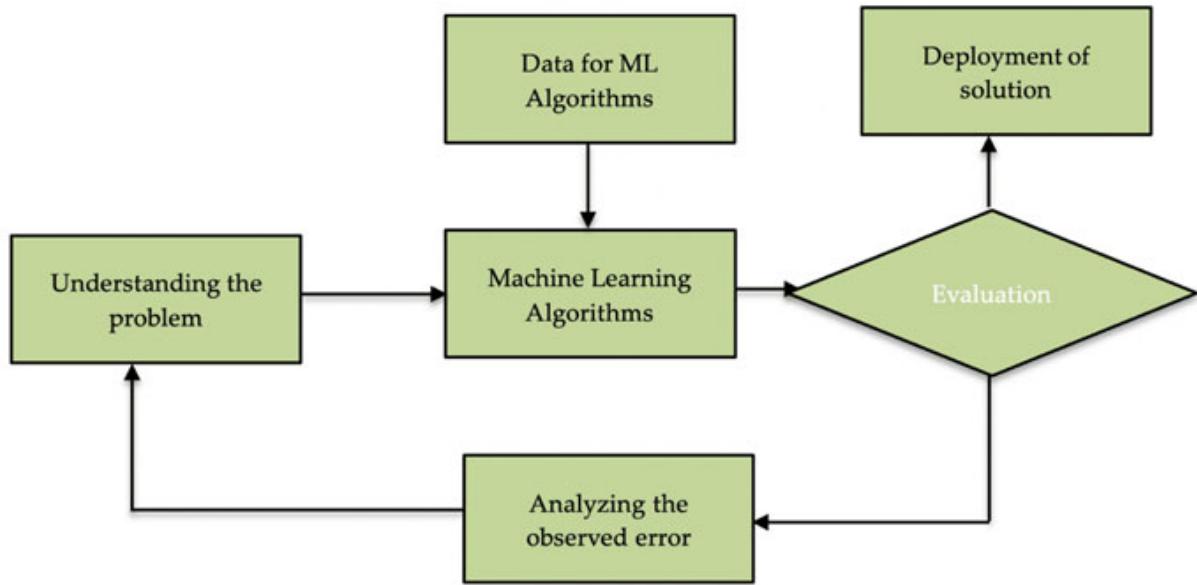


Figure 2.2: The Machine Learning approach

There are a lot of ML algorithms that are being used for classification and regression problems.

The following [Figure 2.3](#) shows the components and interrelation of ML which are data, model, and loss:

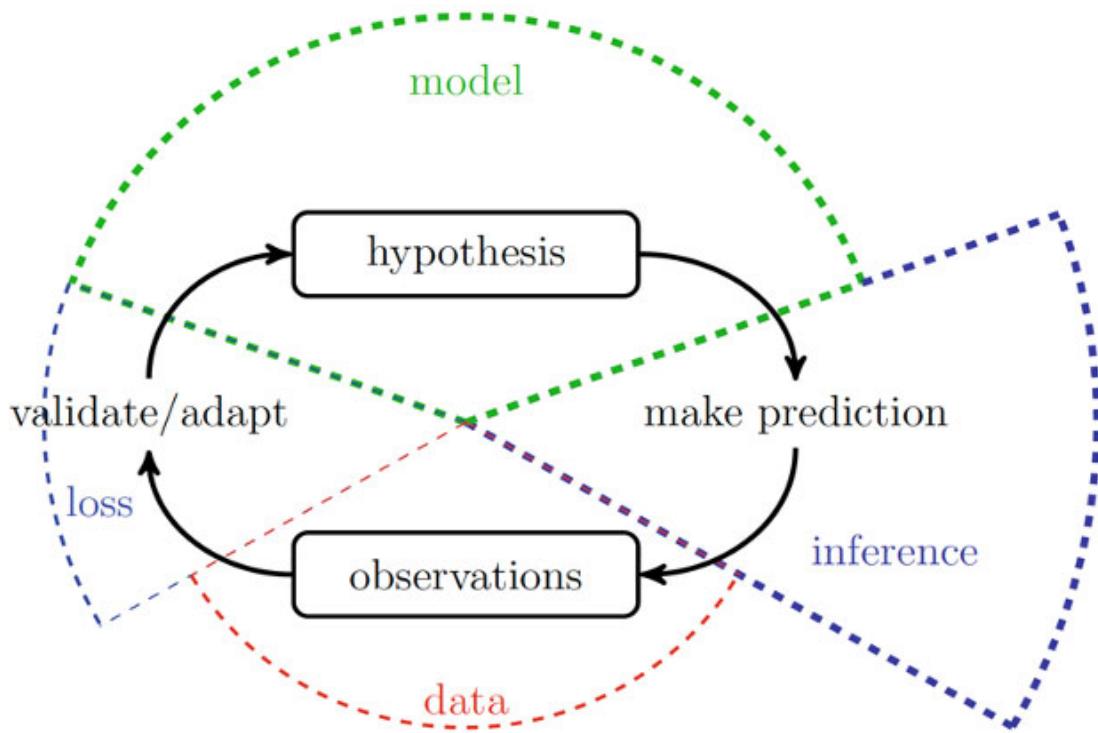


Figure 2.3: Main components of Machine learning

In general, the ML approaches can be classified into the following categories. [Figure 2.4](#) shows the taxonomy of ML algorithms and their applications in different domains:

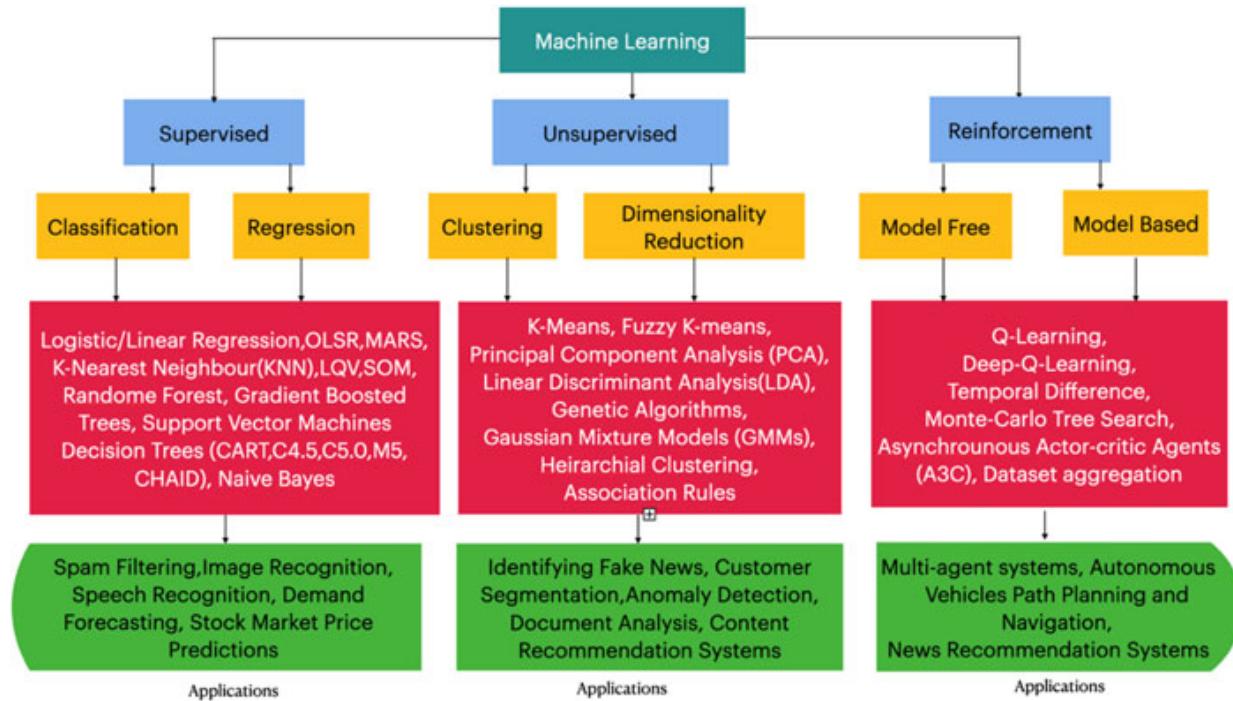


Figure 2.4: A taxonomy of ML algorithms and applications

Supervised learning

In supervised learning, the data is generally labelled as data. Therefore, the machine knows what kind of patterns to look for. This is one of the popular ML methods. If we input the model with a specific set of data, normally known as training data, which contains both the inputs and the corresponding expected output, then the training data will teach the model about the correct output for a particular input.

Unsupervised learning

The models operate independently of human direction or supervision, in unsupervised learning. The model oversees finding the hidden patterns because the data is not labelled. In unsupervised learning, the model receives input that has not been labelled or categorised, and thus, it has no idea of how the result should be presented. For example, if we submit an image of a fruit, we will not identify it as, say, "Orange." In contrast, the model will learn on its own if we merely provide data.

Due to the availability of labelled data, supervised learning has generally gained popularity and tends to give superior outcomes. The only option in many real-world situations when data cannot be consistently classified is to use unsupervised learning ML algorithms.

Clustering and **Principal Component Analysis (PCA)** are two common unsupervised machine learning approaches.

- **Clustering** is a technique for dividing a big number of data points into smaller groups, based on similarities. Clustering seeks natural groupings in data so that points in the same group are more like one another than points in other groups. K-Means, Hierarchical Clustering, and Density-Based Clustering are examples of clustering methods. Exploratory data analysis, market segmentation, image segmentation, and anomaly detection are all common applications for clustering.
- The Principal Component Analysis (PCA) technique is used to decrease the number of variables in a data set while maintaining as much information as feasible. PCA works by transforming the data into a new coordinate system, with the first coordinate representing the

most variance in the data, the second coordinate representing the second most variation, and so on. The transformed variables, referred to as the principal components, are a new set of uncorrelated variables that can be utilized for additional analysis, such as regression or clustering. PCA is frequently used to show high-dimensional data in two or three dimensions, in order to minimize noise in the data, and to improve machine learning algorithm performance.

Reinforcement Learning (RL)

The environment's responses or observations serve as the primary source of information for the RL. The implications of an RL algorithm's actions are fed into the system, via reward or punishment methods. As a result, it must base its decisions on what to do next, on both its prior experiences and its new options, a process known as trial-and-error learning. The working environment rewards the RL agent numerically. The success of an action's outcome is encoded in this reinforcement input data, where a successful action raises the cumulative reward. This teaches the RL agent how to choose activities that will maximize the total reward over time.

The goal of RL is to maximize rewards by acting appropriately in each setting. Operations research, information theory, control theory, game theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics, and genetic and bio-inspired algorithms are some of the fields that use RL.

ML Performance Metrics

Understanding the various assessment metrics is crucial for any ML algorithm, when analyzing the outcomes of any application. This is because the model's performance may vary greatly, depending on which measurement is used from which evaluation metric. Moreover, it is crucial to have a thorough understanding of the assessment and performance metrics, to make sure your ML model is working as efficiently as possible.

To evaluate ML models, the following metrics are being used in general.

Accuracy: The number of accurate predictions divided by the total number of predictions, multiplied by 100, is the definition of classification accuracy.

Accuracy = (Number of Correct Predictions / Total Number of Predictions) * 100

Confusion matrix

A confusion matrix is a $n \times n$ matrix, in which the actual classification of a particular piece of data is denoted by each row, and the classification that was anticipated (or vice versa) was denoted by column. The model's correctness and the number of accurate classifications can be assessed by checking the values of the diagonal elements. In general, any good model will have high values along the diagonal and low values off the diagonal. It is also observed that the highest values that are not on the diagonal, will give the information on where the model is having trouble. When we study both the values carefully, we can spot instances where a model's accuracy may be high, but it regularly misclassifies the same data. [Figure 2.5](#) features a confusion matrix:

		Predicted classes	
		Negative 0	Positive 1
Actual classes	Negative 0	TN	FP
	Positive 1	FN	TP

Figure 2.5: Confusion Matrix

For any given confusion matrix, a factor for evaluation is represented by each box, further explained as follows:

- **True Positive (TP)**: It is the number of positive class samples that the model accurately predicted. It is indicated by the term TP.
- **True Negative (TN)**: It is the number of negative class samples that the model properly predicted. It is indicated by term TN.
- **False Positive (FP)**: It is the number of negative class samples that the model mistakenly predicted. It is indicated by the term FP.
- **False Negative (FN)**: It is the number of positive class samples that the model predicted incorrectly. It is indicated by the term FN.
- **Precision**: Accuracy's limitation can be eliminated by using the precision metric.
- **Recall**: Calculating the percentage of actual positives that were wrongly detected is the goal of a recall.
- **F1-score**: The harmonic mean of precision and recall is given by the F1 score. F1 score is the weighted average of the precision and recall. The best value of F1 would be 1 and worst would be 0. We can calculate F1 score with the help of the formula mentioned in [Table 2.1](#). F1 score is having equal relative contribution of precision and recall.
- **AU-ROC: Area Under Curve (AUC) - Receiver Operating Characteristic (ROC)** is a performance metric, based on varying threshold values, for classification problems. As name suggests, ROC is a probability curve while AUC measures the separability. In simple words, AUC-ROC metric will tell us about the capability of model in distinguishing the classes. For a better model, the AUC should be high.

The following [Table 2.1](#) lists the formulas used for the preceding metrics:

Metric	Formula
Accuracy	$ACC = \frac{TP+TN}{TP+TN+FP+FN}$
Precision	$PRC = \frac{TP}{TP+FP}$
Recall or Sensitivity	$SNS = \frac{TP}{TP+FN}$
F1 Score	$F1 = 2 \frac{PRC \cdot SNS}{PRC + SNS}$

Table 2.1: ML performance metrics

Basics of DL and different DL algorithms

Have you observed how Google Translate, Google Maps, Amazon Alexa or any Amazon page or food delivery apps like Zomato work, recommend and deliver things in smart way? All these products use **Deep Learning (DL)** algorithms in the background which analyses data continuously and displays the best results to the user.

The DL is a subset of ML that has been in almost every sector these days. Many applications mentioned previously, employ different DL models/algorithms to process multiple kinds of sensor data, understand our voice, and recognize objects or images.

Both ML and DL use some sort of statistical methods, but each technique has its own approach for data collection and processing. ML and DL are two subfields of AI that are concerned with the development of algorithms, that enable machines to learn from data and make predictions or decisions without being explicitly programmed.

To put it simply, the amount of abstraction in the algorithms is where machine learning and deep learning diverge. To produce predictions based on the correlations between the characteristics and the target variables, ML algorithms are typically less complex and operate at a higher level of

abstraction. Such algorithms as linear regression, decision trees, and k-nearest neighbours are all examples of machine learning.

However, DL is a subfield of ML that use multi-layered neural networks to acquire knowledge. More complicated associations between inputs and outputs can be learned by a deep learning model since each layer represents an increasingly abstract level of representation. CNNs, RNNs, and autoencoders are all examples of DL algorithms.

For example, a simple ML algorithm could be used to predict the price of a house, based on its square footage, number of bedrooms, and location. A DL algorithm, on the other hand, could be used to perform image classification, such as identifying objects in an image.

Briefly explained previously, DL employs more sophisticated models to learn from data and make predictions than traditional ML does.

In general, any DL solution consists of three stages:

1. Data Understanding and Pre-processing
2. Model Development and Training
3. Model Validation and Interpretation/Inference

Figure 2.6 features the three stages of any DL solution:

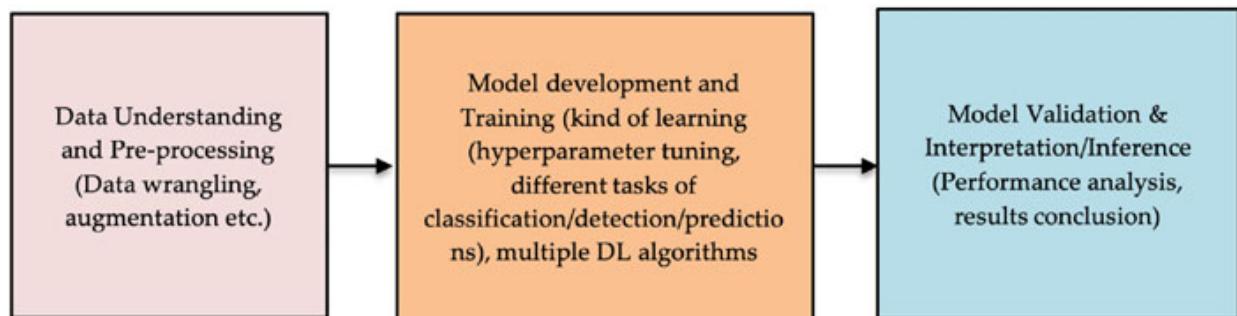


Figure 2.6: High level DL pipeline to solve real-world problems

The following *Figure 2.7* shows the basic differences of implementation steps in machine learning and deep learning, for the classification of two flowers, white and purple:

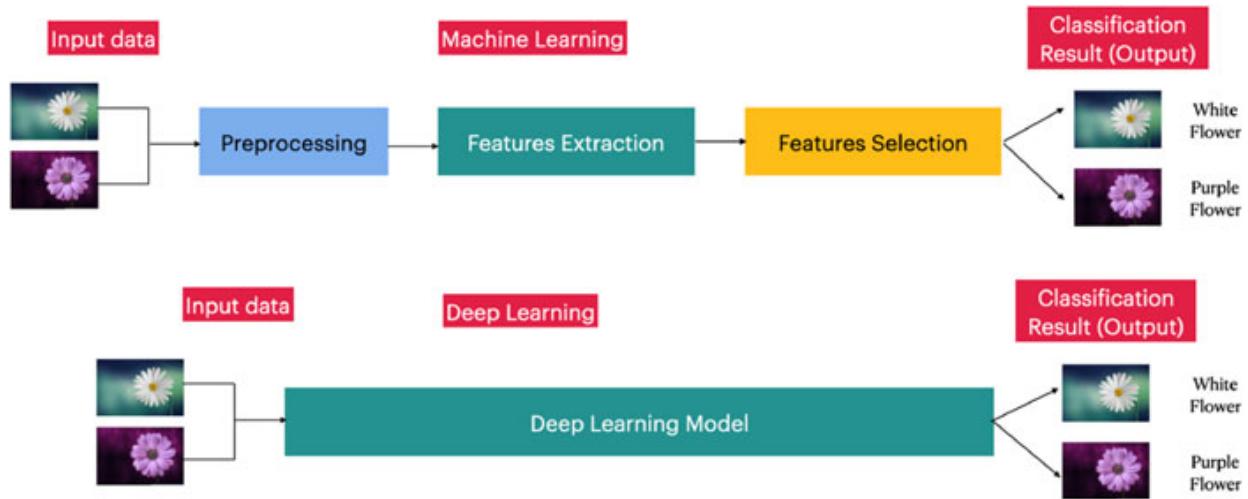


Figure 2.7: Example showing the DL and traditional ML methods

In general, the DL can be classified into the following categories:

- Supervised or Discriminative Learning
- Unsupervised or Generative Learning
- Hybrid Learning

2.8 shows the different DL algorithms available in literature, as well as their applications in multiple domains for solving complex tasks:

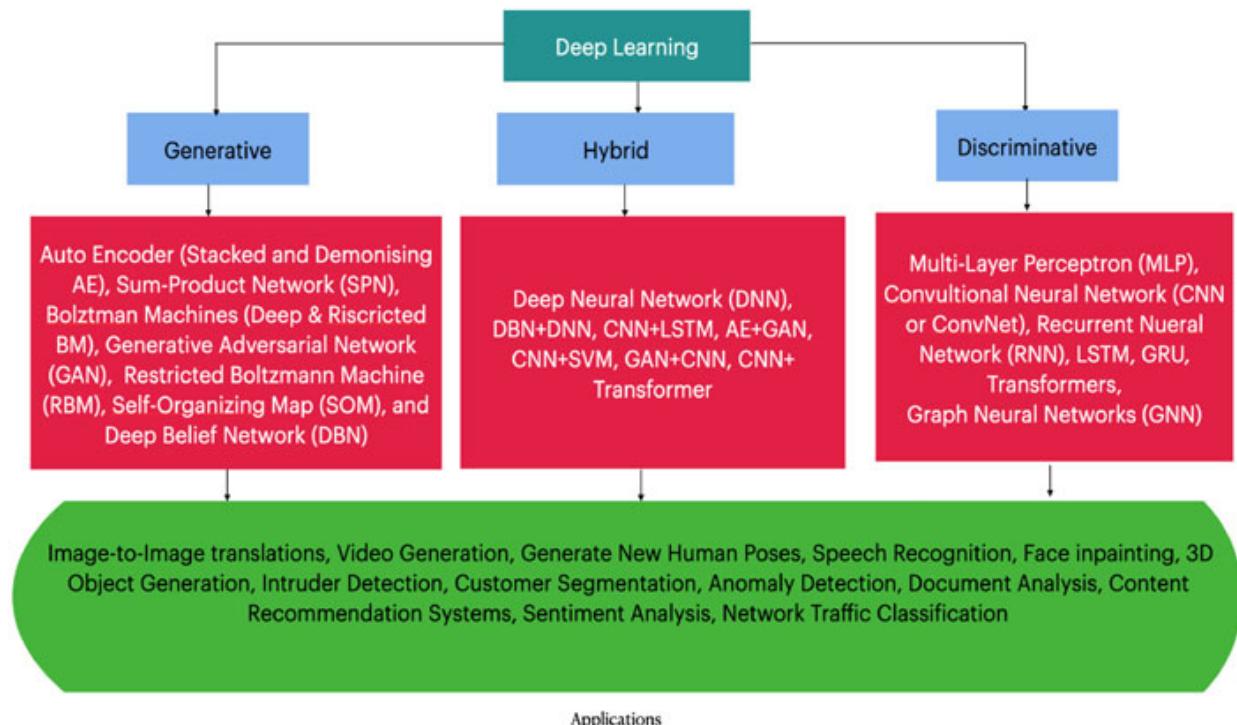


Figure 2.8: A taxonomy of DL techniques and applications

Transfer Learning

When any DL model is trained on a given set of input data, it gains some knowledge from this input dataset, which consists of “weights” of the network. These generated weights can then be extracted and we can transfer these weights to any other DL architecture. Instead of training the other DL architecture from scratch, we “transfer” the learned features from a given set of training data. This process will save a lot of time.

Transfer learning refers to instances such as storing knowledge obtained while solving a given problem, and using that knowledge for a different problem in a similar or related domains. For example, in image recognition tasks, deep learning models are known to provide **state-of-the-art (SOTA)** solutions. For example, in YOLO architecture instances, the transfer learning can be used to generate the required generic features from the pretrained DL model, and then those features can be used for building a simple DL model for solving the new problem. So, for the new model to build, we will be using very few parameters.

Tools and Different ML, DL frameworks

For any ML or DL activity, the development of many frameworks and libraries by numerous companies, including Google, Facebook, Intel, and Qualcomm, as well as diverse programming languages (Python, C, and C++), will be necessary, for the training and deployment of algorithms. Consider the fact that straight deployment of ML models intended for high-end edge devices such as single-board computers or mobile phones, is typically not appropriate for microcontrollers. Any average **Microcontroller (MC)** has 128 KB RAM and 1 MB of flash memory or even less, and yet the most recent mobile phones might have 4 to 6 GB of RAM and 64 to 128, or 256 to 512 GB of storage. Due to the extreme resource constraints of MCs, the onboard deployment of ML pipelines typically involves a methodical design of workflow and software tools (for example, Edge-Impulse).

In general, to select any good framework, we need to consider the following basic things:

- Very good parallel computation

- Good interface to run our models
- Huge number of inbuilt packages
- Better optimization techniques
- Flexibility and easy analysis of the required business problem

We will now be understanding some of the well-known frameworks and libraries that are being used for development and deployment of ML and DL models on MCs.

Python

The growth and study of ML has made Python, an interpreted, high-level, and general-purpose programming language, very popular. Python has several libraries and frameworks devoted to ML, which is one of its main benefits over other programming languages (such as TensorFlow, Keras, PyTorch). With the help of these libraries, we can drastically cut down on development time.

Jupyter Notebooks

The interactive web application, Jupyter Notebook, integrates code, computational results, explanatory text, and multimedia elements into a single document. This strategy makes it easier for various team members to work together on a project in a shared development environment. The use of Python and Jupyter Notebooks has grown significantly among researchers studying ML.

Google Colaboratory

A web-based program called Google Colaboratory (aka Google Colab) uses cloud computing and Jupyter Notebook to enable high processing capacity. It functions without any prior preparation, using the Chrome web browser. Additionally, it facilitates sharing and permits concurrent editing of a Jupyter Notebook file.

TensorFlow (TF), TFLite and TensorFlow Lite Micro

Google created the open-source ML framework, known as TensorFlow. Its C++ code is extremely efficient and has a Python front end. It has a sizable development community and is utilized in both the industry and academics. Its major goal is to make it easier to develop ML models. For data pre-processing, data ingestion, model evaluation, visualization, and serving, it includes a wide range of features.

TF is made to be extremely portable and can function on a wide range of hardware and software platforms. However, our microcontrollers only have about 1MB of memory, compared to the 400 MB size of the regular TF version. As a result, these tiny devices cannot be deployed with the regular TF framework. Fortunately, there is a scaled-down variation of the framework called TF Lite, that is made especially for devices with additional limitations.

TensorFlow Lite

Working with severely constrained devices that have little memory, processing power, or storage, is a key component of TinyML. The fundamental TinyML applications, on the other hand, typically do not require the full ML pipeline to function, nor do they require all the cutting-edge methods employed by researchers. Many TensorFlow functions are therefore not necessary. Only a small portion will be utilized. TensorFlow Lite is all about doing this. To run models on mobile, embedded, and IoT devices, TF Lite (TFLite) is an optimized version of TensorFlow.

TFLite provides on-device inference with minimal latency and tiny binary size (about 1MB). It is accomplished by eliminating auxiliary features that are not required for mobile deployment. A model is initially trained using the regular TensorFlow framework, and then it is compressed using optimizations like pruning or quantization to make it acceptable for edge devices. The built model can then be deployed and used for inference by TFLite. However, the 1MB size of TFLite will still not fit in the flash memory of our microcontroller. For this severe case, TensorFlow has been scaled down even more. The following Figure 2.9 shows the workflow of TensorFlow models:

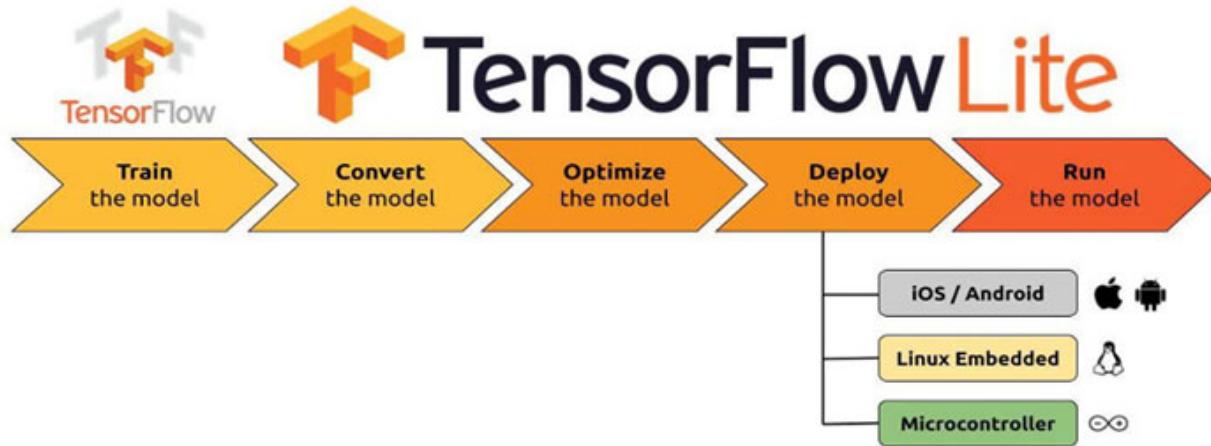


Figure 2.9: The workflow of deploying TensorFlow models on microcontrollers [10]

The advantages of using a TensorFlow Lite Model for on-device Machine-Learning are as follows:

- **Latency:** As inference is taken on the edge, there is no round-trip to a server, resulting in low latency.
- **Data Privacy:** Due to inference at edge, the data is not shared across any net network. So, the personal information does not leave the device, resolving any concerns to the data privacy.
- **Connectivity:** As no internet connectivity is required, there are no connectivity issues.
- **Model Size:** TensorFlow Lite models are lightweight as the edge devices are resource constrained.
- **Power Consumptions:** Efficient inference and lack of network connectivity led to low power consumption.

TensorFlow Lite Micro

Another cutting-edge inference framework from Google is called TensorFlow Lite Micro. It is intended for use with MCs and other embedded hardware, that has only a few kilobytes of memory. The standard framework is severely compressed with all but the most basic functions are being left behind. Numerous simple models can be run by the core runtime, which only takes up 16 KB on an Arm Cortex M3. It does not need a typical C or C++ library, dynamic memory allocation, or operating system support; it can run on bare metal. On the downside, the framework does not provide any

graphing or debugging tools, making it more difficult to diagnose any problems. Various embedded microcontrollers support TF Lite Micro.

To run any TF Model on the microcontrollers, follow the given steps:

1. **Developing ML model:** Create a compact TF model with supported operations, that can fit your target device (microcontroller/embedded device).
2. Use the TF Lite converter to convert to a TF Lite model.
3. Use the commonly available tools/software to convert it to a C programming byte array code and place it in the device's read-only program memory, that is, the ROM.
4. Utilize the C++ library to run inference on the device, and then process the results for the required application targeted for.

AI Model Efficiency Toolkit (AIMET)

The most recent DL models often include many model parameters and a large model size. Deploying DL models on the microcontrollers, which have limited computational power and memory availability, is difficult because of the size of the models. We will therefore need to quantize and compress the models. The task will benefit from the use of AIMET. It also aids in improving run-time speed with lower computation and memory requirements to deploy the ML models, and it has little effect on the deployed model's overall accuracy. [Figure 2.10](#) features the AIMET workflow:

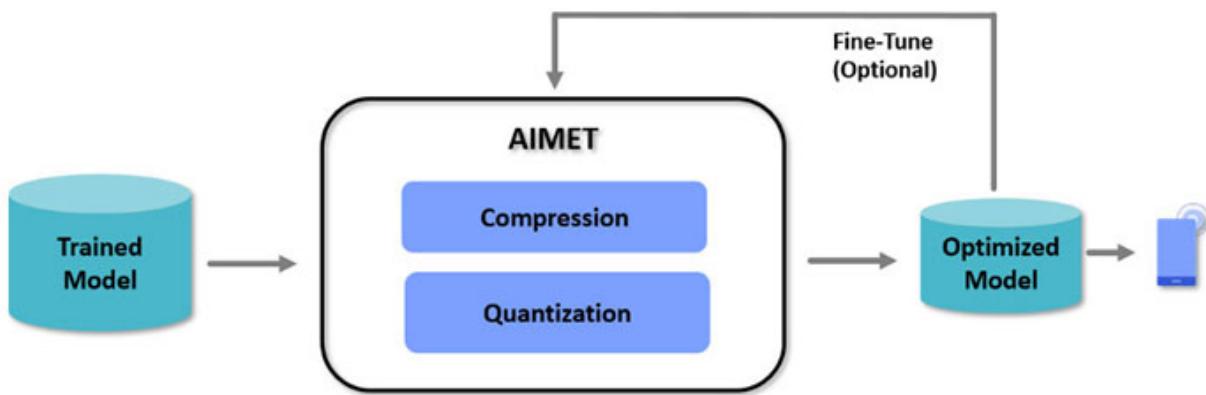


Figure 2.10: AIMET workflow [1]

The following [Figure 2.11](#) features a high-level view of AIMET architecture:

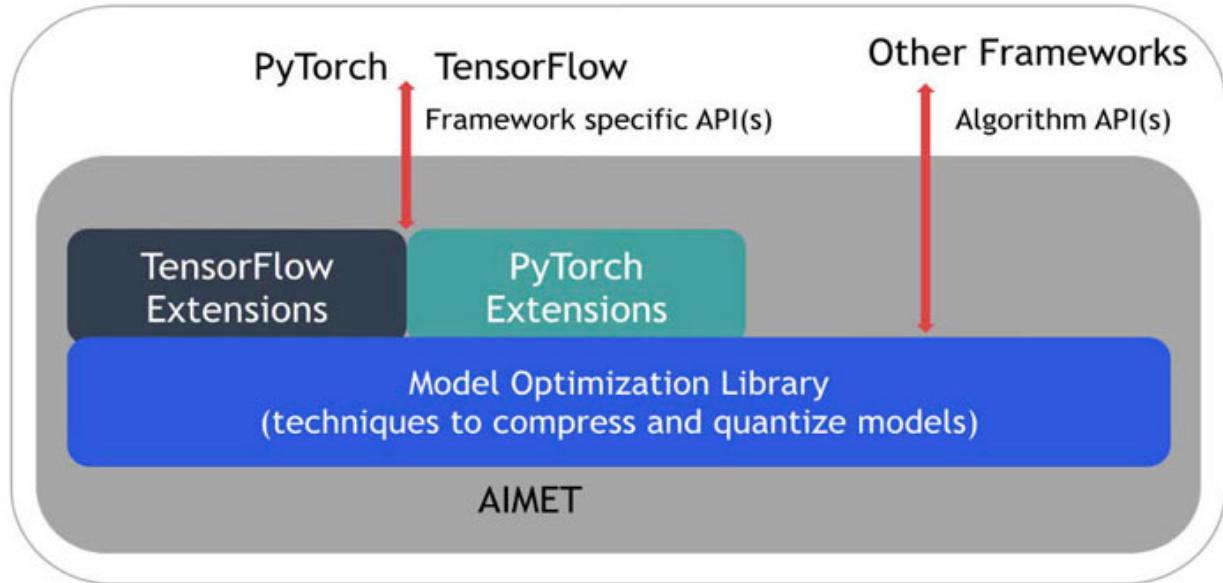


Figure 2.11: High-level view of AIMET architecture [2]

AIMET is developed mainly to work with PyTorch and TensorFlow models. The main benefits of using AIMET are as follows:

- Lower power
- Lower storage
- Lower memory bandwidth
- Higher performance

Convolutional Architecture for Fast Feature Embedding (Caffe)

Convolutional Architecture for Fast Feature Embedding (Caffe) is a DL framework developed by **Berkeley AI Research (AIR)** and many community contributors from all over the world. It is used widely in academic research projects for areas like **Computer Vision (CV)**, **Natural Language Processing (NLP)**, and so on. Caffe provides state-of-the-art DL algorithms and a collection of reference models. The framework is a C++ library with Python and MATLAB bindings for training and deploying general purpose **Convolutional Neural Networks (CNNs)** and other deep

models efficiently on commodity architectures. Caffe supports many different types of DL architectures geared towards image classification and image segmentation. It supports CNN, RCNN, LSTM and fully connected neural network designs. Caffe supports GPU and CPU based acceleration computational kernel libraries such as NVIDIA cuDNN and Intel MKL. With Caffe, models can be defined without hard-coding and with configuration. Caffe is used by most developers because of its speed. For example, a single NVIDIA K40 GPU device can analyze 60 million photos per day using it. Since Caffe lacks a higher-level API, it is difficult to conduct experiments and we must compile each source code before deploying our model on the end device.

CoreML

To develop a ML model, Core ML applies a ML algorithm to a set of training data. Based on fresh incoming data, we can utilize models to anticipate future events. A wide range of tasks that would be challenging or impractical to write in code, can be carried out using models. For instance, we can train a model to classify images or to directly identify certain items from a photo's pixels. [Figure 2.12](#) features CoreML integration with real world applications:



Figure 2.12: CoreML integration with real world applications [4]

The Create ML program/app included with Xcode allows us to create and train a model. The Core ML formatted models that were trained, are available for usage in our application. As an alternative, we can utilize a wide range of other ML libraries and then transform the model into the Core ML format using Core ML Tools. We can utilize Core ML to retrain or improve a model once it has been downloaded to a user's device using the user's data.

Utilizing the CPU, GPU, and Neural Engine while reducing memory footprint and power consumption, Core ML optimizes on-device performance. Running a model solely on the user's device eliminates the need for a network connection, thus preserving the privacy of the user's data and enhancing the responsiveness of your app. Neural networks, tree ensembles, support vector machines, and generalized linear models are just a few examples of the machine learning models that Core ML supports. The Core ML model format (models with a `.mlmodel` file extension) is required by Core ML.

Open Neural Network Exchange (ONNX)

ONNX provides an open-source format for AI models, both DL and traditional ML. It defines an extensible computation graph model, as well as definitions of built-in operators and standard data types. For exchanging DL models, it is a new standard. The DL models will supposedly become portable, preventing vendor lock-in. Think of a scenario where we have trained our model and want to deploy it to a new iOS app, so that anyone can use it to verify the safety. Our model was initially trained using PyTorch, but iOS anticipates utilizing CoreML inside the app. We may simply transition from one environment to the next thanks to ONNX, which serves as an interim representation of our model. [16].

Figure 2.13 features the ONNX usage:

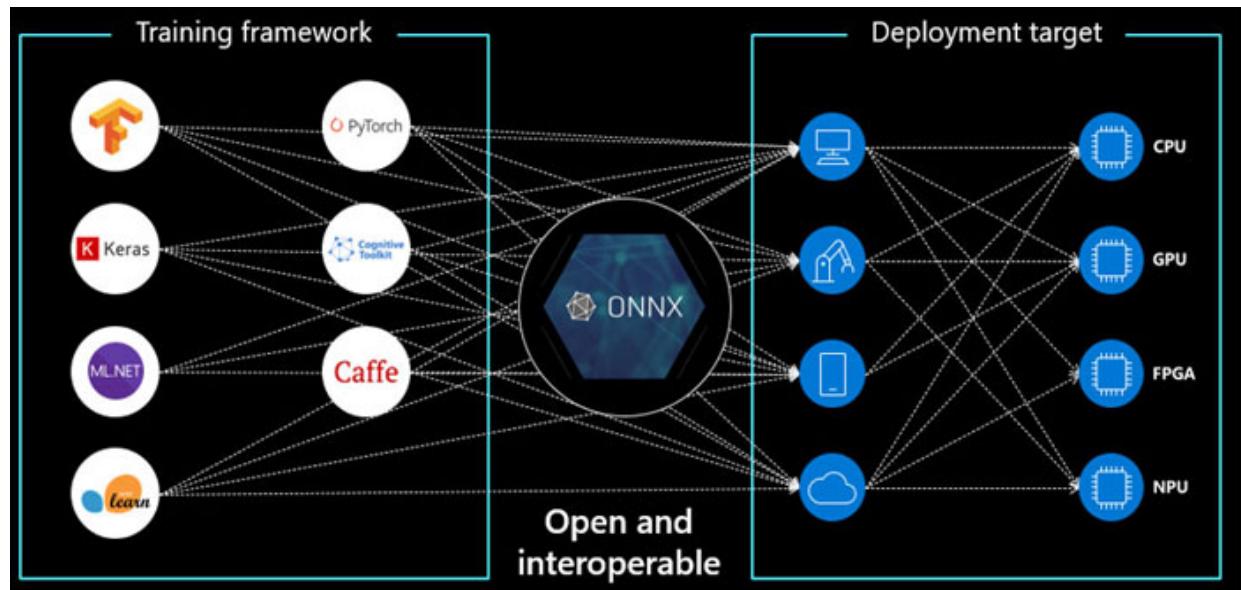


Figure 2.13: ONNX usage

Open Visual Inference and Neural network Optimization (OpenVINO)

OpenVINO is a free toolkit that makes it easier to deploy an inference engine-based DL model onto Intel hardware and optimize the models from a framework. The models are trained with well-known frameworks such as Caffe, Pytorch and TensorFlow, and they can be converted and improved. We can deploy the converted models in a variety of Intel hardware and software settings, including cloud, on-premises, and mobile.

There are two versions of the toolkit: Intel Distribution of OpenVINO Toolkit, which is supported by Intel, and OpenVINO Toolkit, which is supported by the open-source community.

Figure 2.14 features the OpenVINO supporting frameworks:



Figure 2.14: OpenVINO supporting frameworks [6]

Pytorch and PyTorch Mobile

Another open-source framework created by Facebook is Pytorch. It offers auto differentiation, which speeds up the backpropagation process. PyTorch has a variety of modules, including `torchvision`, `torchaudio`, and `torchtext`, which are flexible enough to operate with NLP and CV applications. It is a Pythonic approach of implementing our DL models.

When we compare the Pytorch flexibility for developers and researchers, it is more flexible and suitable for researchers.

Embedded Machine Learning (EML)

Deploying ML models on the embedded devices has attracted a lot of attention in the recent years, and number of application areas using EML has also increased a lot. Embedded devices for ML applications can fulfill many tasks in multiple industry areas such as IoT and Oil and Gas (video surveillance using cameras for safety applications). Consider for an example, an audio sensor detecting an acoustic anomaly. It can help and support quality assurance in production or condition monitoring of large systems. Cameras will be useful in identifying intruder detection, oil spillage and safety of workers.

The following [*Figure 2.15*](#) shows the sample stack/layout of EML. TinyML plays a crucial role in implementing EML applications.

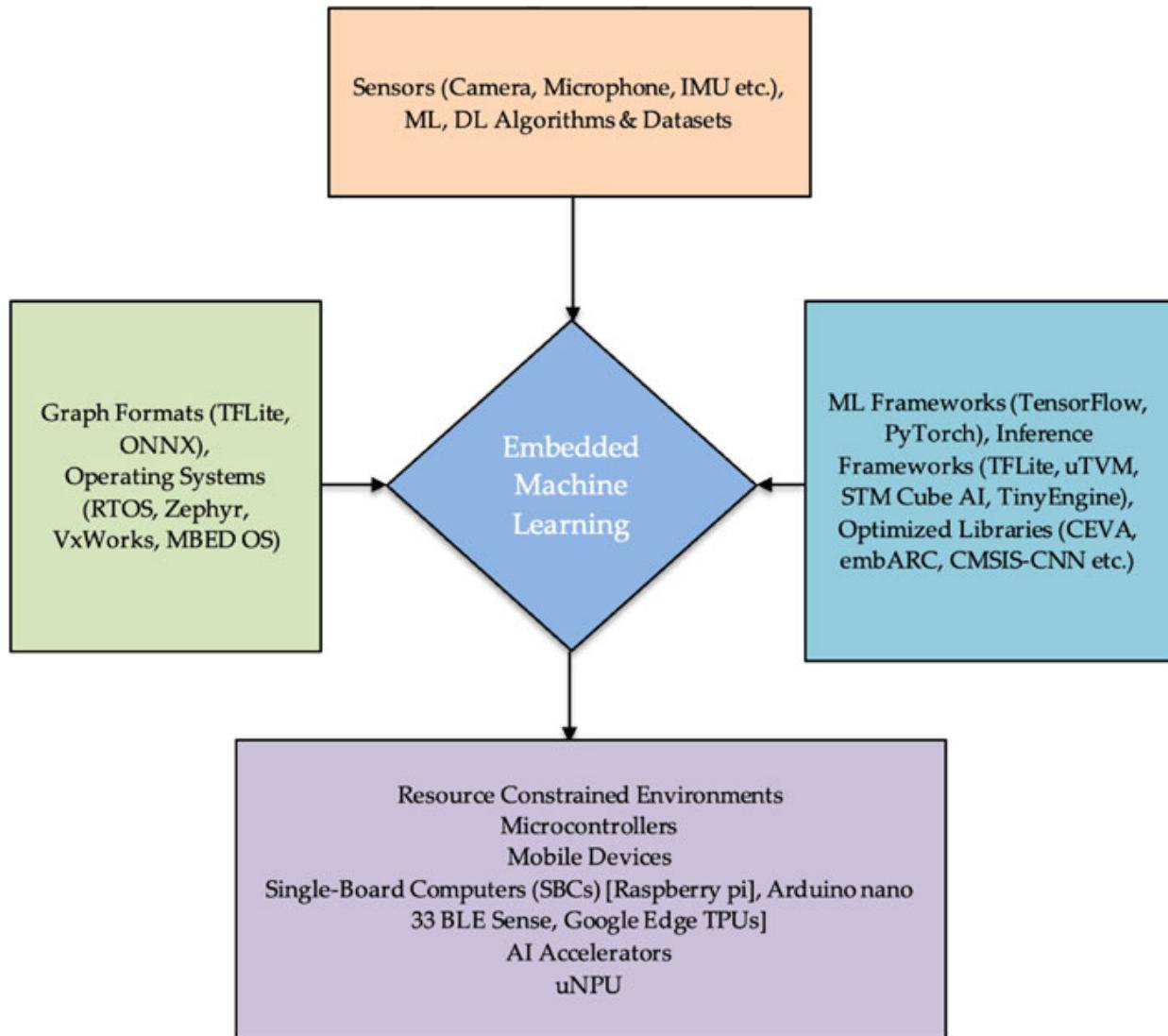


Figure 2.15: EML sample stack/components

Difference between Learning and Inference

The ML life cycle includes two main phases:

1. **The training/learning phase:** It involves creating a ML model, training it by running the model on labeled data examples, and then testing and validating the model by running it on unseen examples.
2. **Inference phase:** It involves putting the ML model to work on live data, to produce an actionable output/result. During this phase, the inference system accepts input from end-users, processes the data, feeds it into the ML model, and serves outputs back to users.

ML inference cannot happen without training. The difference between ML training and inference is that the former is a building block for the latter phase.

Let us consider an example, to differentiate between training and inference using ML. Say we are training an **Autonomous Vehicle (AV)** that follows traffic laws. Firstly, the AV system would need to know the traffic rules and regulations. So, a programmer can train it on datasets and instructions, which will help it in recognizing the traffic signs, stoplights, surrounding objects such as pedestrians, and animals among other things. The AV system will memorize the rules of the road by looking at examples and learn to analyze what it should do when it observes certain signs or scenarios in the environment.

But when driving on the road, the AV would need to make inferences based on its surroundings. It is simple to say that the model can return results based on the training it got, but if the input data has variations such as change in traffic light color or unknown object is encountered, then the model needs additional intelligence.

This is the precise situation where DL inference enters the picture. It aids in bridging the gap between the confusing information found in the real world and the data the machines were trained on. Thanks to the varied recent DL architectures (such as YOLO and its variants), which may extrapolate from traits it identifies, the AV could conclude that the sign it encountered is still a stop sign.

[Figure 2.16](#) shows the difference between DL model training and inference:

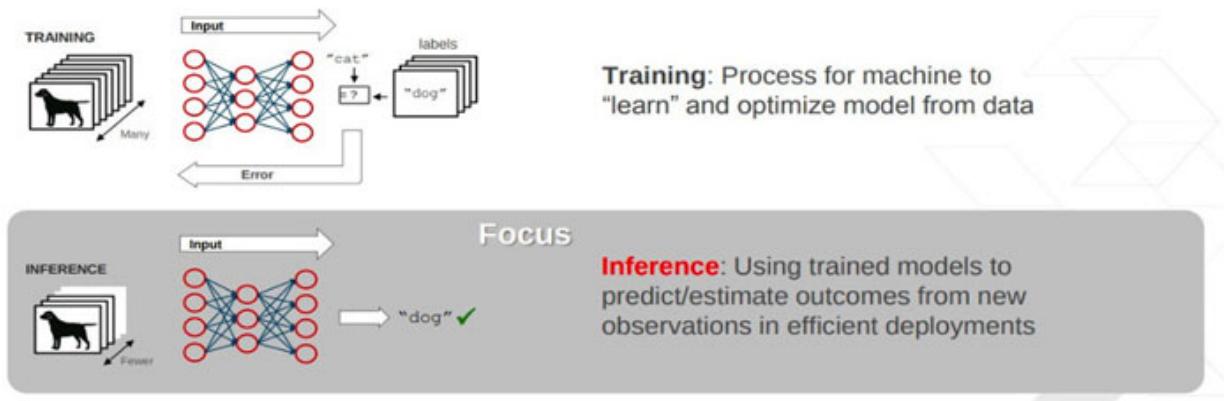


Figure 2.16: DL model training vs inference [11]

ML model deployment and inferencing on different platforms

Moving ML inferencing to edge devices, where there is already a sizable amount of sensor data being collected and is underused for in-depth analysis, is the most recent trend in the AI business. Think about the necessity for an edge device to be connected to data centers to execute various ML applications across various domains. The primary and most significant reason for directing ML toward edge devices, is to enable quick access to the vast amounts of real-time data produced by these devices. This will allow for quick model training and inference, which will further give the devices human-like intelligence to respond to a variety of real-time events.

As mentioned in the starting of the chapter, the main goal of this book is to understand the steps involved in deploying ML/DL algorithms on small, and low powered devices such as microcontrollers. With the rise of the IoT, the usage of MCs has also increased a lot. MCs are inexpensive, easy to program and are powerful enough to run sophisticated ML algorithms. Running the ML algorithms on the Microcontrollers have three advantages: latency, power consumption, and privacy.

Figure 2.17 features the concept of Edge Computing:

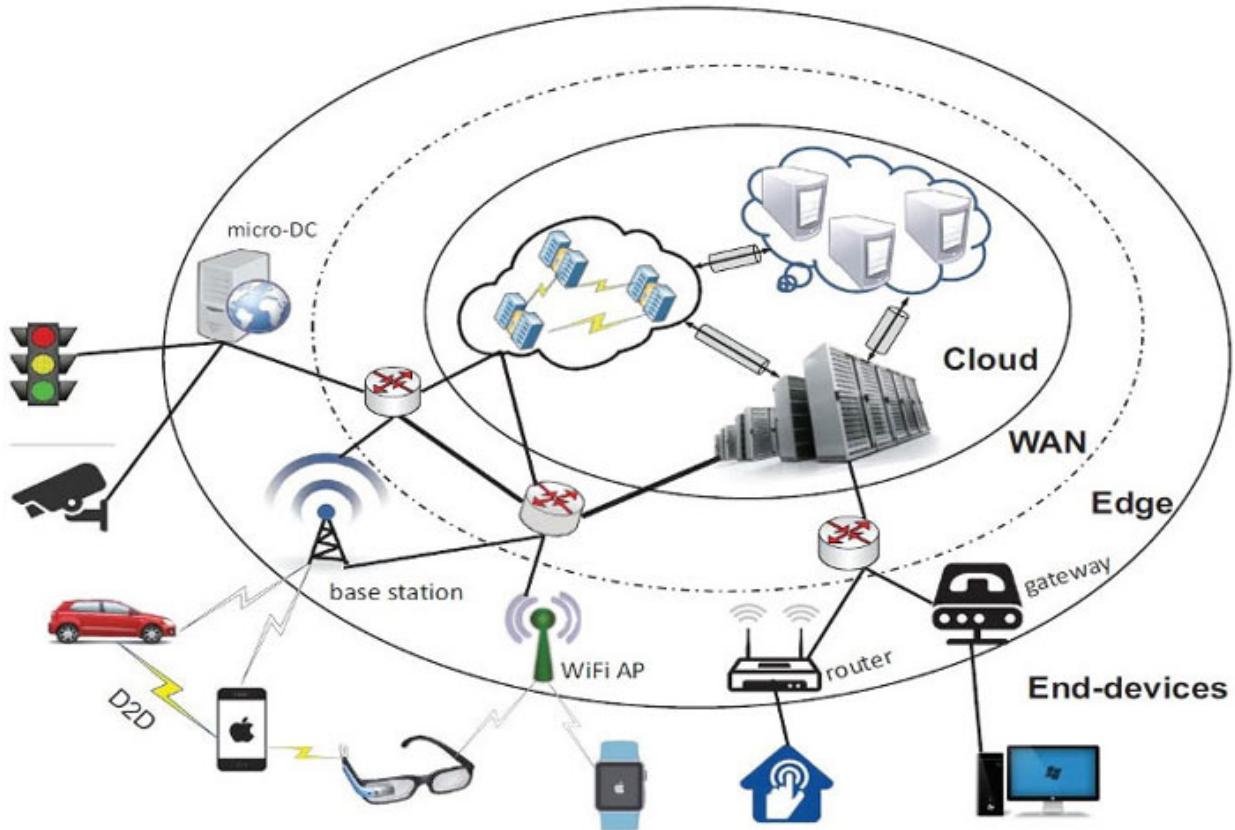


Figure 2.17: Concept of Edge Computing [12]

The DL model need not be fully trained or inferred at the edge, for edge intelligence to exist. As a result, there are cloud-edge situations that involve co-training and data offloading. [Figure 2.18](#) shows the concept of edge computing in general. Frame works such as **Federate Learning (FL)** use the previous concepts to improve the privacy and security.

The major difference between cloud and on-device inference is explained as follows.

ML/DL models are trained and inferred entirely on the cloud, which is what is meant by "cloud intelligence." Inference made on the device indicates that no data would be offloaded. It incorporates cloud-based ML/DL model training, with all inferencing taking place at the device level.

[Figure 2.18](#) explains the ML models training and inferencing at different levels in IoT:

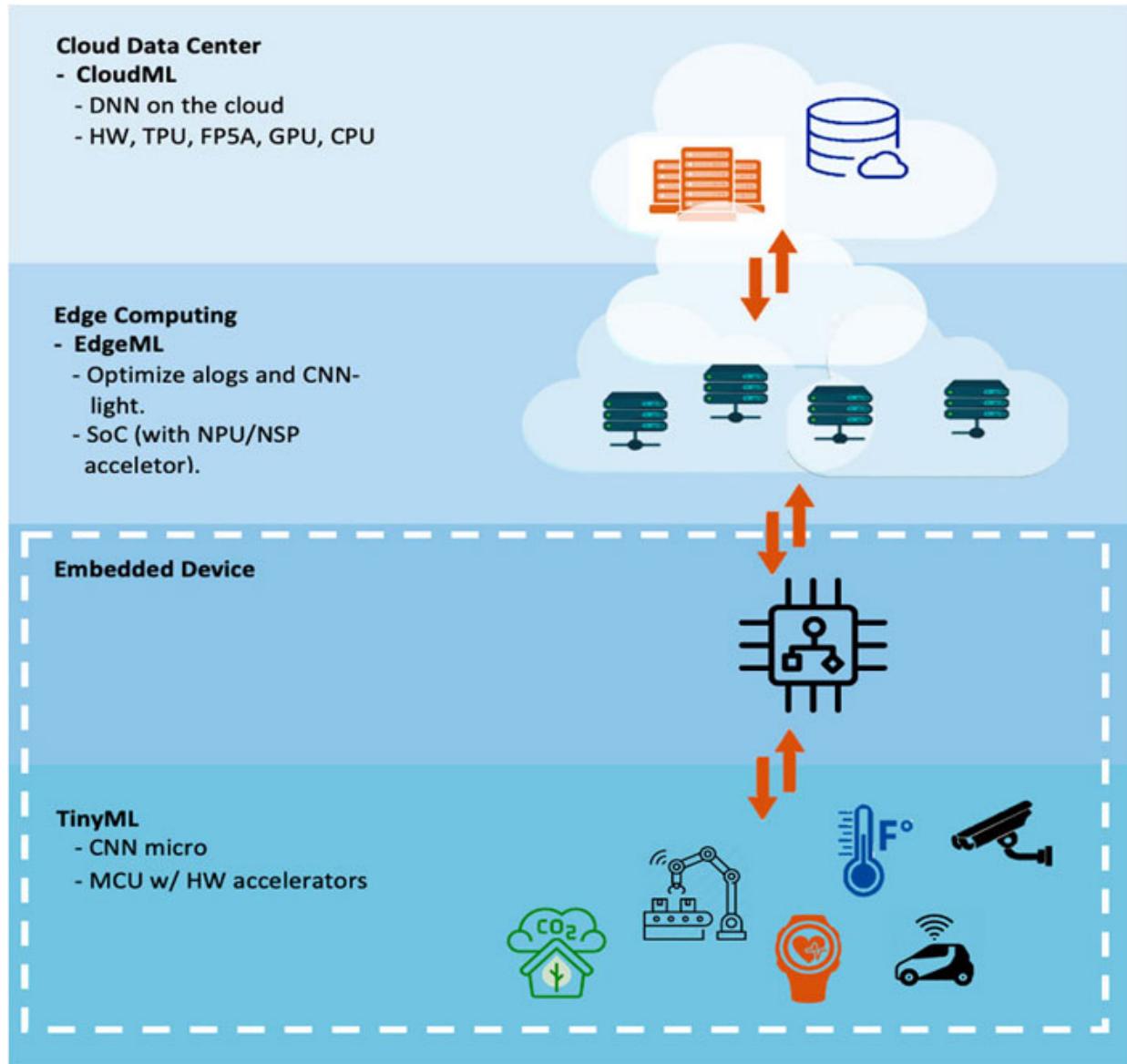


Figure 2.18: A generic framework for IoT applications with Cloud computing, Edge computing and TinyML [13]

Conclusion

In this chapter, we covered the traditional methods to solve complex problems, different types of ML categories and its performance metrics being used. We have also understood popular DL algorithms being used for solving problems in computer vision and other domains. We covered different tools, libraries and frameworks used for development and deployment of ML models, on various embedded devices and microcontrollers. Deploying ML models on embedded devices has many

benefits such as improved decision making and privacy of the data. We also listed the differences between the learning and inference using ML models at different levels of deployments. In the next chapter, we will explore the TinyML hardware and software platforms.

Key facts

- ML approach in solving complex problems has better accuracy when compared to traditional implementation. Moreover, a lot of intelligent automation is also possible.
- ML can be mainly categorized into three types: supervised, unsupervised and reinforcement learning.
- Important metrics to assess the overall performance of any ML model are accuracy, precision, recall, F1 score and ROC values.
- When compared with ML models, DL models require additional computational power due to its huge number of parameters.
- Transfer learning saves more time and effort to apply DL for applications in similar areas, when we already have a trained DL model available in solving the problem.
- With the aid of ML frameworks, hardware, various operating systems, optimization libraries, and algorithms, embedded machine learning attracted lot of attention in industry and academia.
- TinyML plays an important role in deploying ML models on Microcontrollers and edge devices.
- Both ML model training and inferencing is possible on edge devices, but only inferencing is possible on Microcontrollers. Federated Learning is being experimented for training of ML models on the MCs, but it is still in the basic research phases only.

Questions

1. How would you define Machine Learning?
2. What is a labelled training set?
3. Can you name four common unsupervised tasks?

4. What type of DL algorithm would you use, to solve image classification task for Autonomous Vehicle?
5. List the ML frameworks used for model development and deployment on microcontrollers.
6. Define transfer learning. Why is it useful?
7. List the steps involved in deploying ML model using TensorFlow Lite framework.
8. What are the benefits of deploying ML on embedded devices?
9. What is the difference between ML model learning and inference?
10. Can we train the model on Microcontroller? Explain.

References

1. <https://github.com/quic/aimet>
2. Siddegowda, S., Fournarakis, M., Nagel, M., Blankevoort, T., Patel, C. and Khobare, A., 2022. Neural Network Quantization with AI Model Efficiency Toolkit (AIMET). arXiv preprint arXiv:2201.08442.
3. Jia, Y., Shelhamer et. al., “Caffe: Convolutional architecture for fast feature embedding.” In Proceedings of the 22nd ACM international conference on Multimedia (pp. 675-678), 2014.
4. <https://developer.apple.com/documentation/coreml>
5. <https://medium.com/analytics-vidhya/speedup-pytorch-inference-with-onnx-284d6a47936e>
6. <https://www.intel.com/content/www/us/en/developer/tools/openvino-toolkit/overview.html>
7. <https://manisha-sirsat.blogspot.com/2019/11/machine-learning-overview.html>
8. Sarker, I.H., 2021. Deep learning: a comprehensive overview on techniques, taxonomy, applications and research directions. SN Computer Science, 2(6), pp.1-20.
9. <https://atiselsts.medium.com/running-tensorflow-lite-for-microcontrollers-on-contiki-ng-d938f25193f5>
10. <https://www.tensorflow.org/lite/microcontrollers>

11. <https://www.xilinx.com/applications/ai-inference/difference-between-deep-learning-training-and-inference.html>
12. <https://viso.ai/edge-ai/edge-intelligence-deep-learning-with-edge-computing/>
13. Z. Zhou, X. Chen, E. Li, L. Zeng, K. Luo and J. Zhang, "Edge Intelligence: Paving the Last Mile of Artificial Intelligence With Edge Computing," in Proceedings of the IEEE, vol. 107, no. 8, pp. 1738-1762, Aug. 2019, doi: 10.1109/JPROC.2019.2918951.
14. Alajlan, N.N. and Ibrahim, D.M., 2022. TinyML: Enabling of Inference Deep Learning Models on Ultra-Low-Power IoT Edge Devices for AI Applications. *Micromachines*, 13(6), p.851.
15. Alzubaidi, L., Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *J. Big Data* 8 (1), 1–74 (2021).
16. <https://github.com/onnx/onnx>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 3

TinyML Hardware and Software Platforms

Introduction

As the **Internet of Things (IoT)** became more popular, there was a significant surge in commercial interest in extending edge machine learning to **Microcontrollers (MC)**, which opened entirely new opportunities for edge machine learning in the form of TinyML. TinyML's deployment of machine learning models on systems with extremely low power consumption, is one of its primary goals. This enables it to execute inference, achieve robust performance, and break the power consumption barrier, all at the same time. There has been a lot of interest shown in TinyML in recent years by both industry and academic researchers. This interest is because TinyML has the potential to transform a wide variety of application areas, including **Computer Vision (CV)**, speech processing, and **Natural Language Processing (NLP)**, amongst others. Because of this, many libraries and tools have been regularly developed and put into use, to simplify the process of implementing machine learning algorithms on platforms with limited resources, such as MCs and embedded devices. The TinyML eco-system in its whole, is comprised of many hardware boards, sensors of varying types, software platforms, and algorithmic frameworks.

Structure

In this chapter, the following topics will be covered:

- Servers at Data Centers, CPUs, GPUs and TPUs
- Mobile CPU, Raspberry Pi board and its types
- Microcontrollers and Microcontroller with AI accelerators
- TinyML Hardware Boards

- TinyML Software Suites
- Data Engineering Frameworks
- TinyML Model Compression Frameworks

Objectives

In this chapter, we will learn the basic differences among CPUs, GPUs, Raspberry Pi boards, TPUs and Servers at Data Centers. We will also explore the different categories of microcontrollers that can be used for the TinyML implementation as well as different types of Raspberry Pi boards. Then we focus more on the TinyML hardware boards and software platforms, that are required to implement the machine learning using TinyML. In the software platforms, we will discuss in detail on the major platforms along with data engineering and model compression frameworks.

Servers at Data Centers: CPUs, GPUs and TPUs

Powerful computers called "servers" are used to store and process massive amounts of data at data centers. A data center is a big, highly protected, and specialized facility where these servers are housed. They have access to a rapid network that expedites data transfer and sharing. Redundant power and cooling systems, as well as stringent security measures, are standard in modern data centers, to ensure the safety of the servers and the information they hold. Sites, apps, and services that have a high need for storage and processing capacity, are hosted on servers in data centers. Clusters of servers in a data center work together to provide a reliable, scalable environment for hosting applications and services. Connected to a fast network, they may exchange data and talk to one another and to other devices such as servers and routers.

The **Central Processing Unit (CPU)** is the brain of the computer, responsible for tasks such as executing programs and crunching numbers. To render high-quality images and video, a dedicated graphics processor called a **Graphics Processing Unit (GPU)** is required. **Tensor Processing Units (TPU)** are specialized computers built for AI and machine learning applications, such as neural network inference and training.

When it comes to graphics processing and image rendering, nothing beats a GPU. A GPU can handle several rendering instructions simultaneously

because it has more cores than a CPU.

TPUs are a particular sort of processor optimized for the intensive computations needed by machine learning and AI systems. Its strength lies in its ability to do matrix operations, making it ideal for large NLP and image processing applications. TPUs supplement the capabilities of a CPU or GPU expanding their performance. Both GPUs and TPUs are specialized CPUs optimized for different tasks.

In general, GPUs are optimized for graphics-intensive operations such as 3D graphics rendering and video processing. Typically, they can accomplish these duties more quickly and effectively than a CPU. By contrast, TPUs were developed to handle machine learning tasks, such as those involved in training and running neural networks. They are more effective and quicker than a CPU or GPU in the mathematical calculations required for training and running neural networks.

Overall, machine learning workloads are better suited to TPUs, whereas GPUs are better suited to jobs that need quick and efficient graphics processing. TPUs are often more powerful and efficient for machine learning tasks than GPUs, although GPUs remain the more general-purpose and popular processor. TPUs are also more specialized and not often employed for various types of jobs, while GPUs are commonly used for a wide range of applications. The following [*Figure 3.1*](#) shows the CPU, and GPU sample images:



Figure 3.1: Servers at Datacenters, CPU and GPU sample images

Mobile CPU, Raspberry Pi board and its types

A mobile CPU is a processor that is developed primarily for usage in mobile devices such as smartphones and tablets. These processors are often smaller, lower-power, and more energy-efficient than regular desktop CPUs, allowing them to operate on battery power for longer periods of time. A CPU, or a central processing unit, is a computer's main processor, that is in

charge of executing instructions and doing calculations. The CPUs often consist of many cores that may run multiple instructions at the same time.

Raspberry Pi boards are single-board computers that are small and inexpensive, and are intended for educational and hobbyist purposes. They use the ARM architecture and usually run the Linux operating system. Raspberry Pi boards are popular for **Do-It-Yourself (DIY)** projects, such as building a home automation system or a media center, but they can also be used as a low-power desktop computer. The Raspberry Pi line-up contains numerous versions with varying features and capabilities, including different CPUs, memory capacities, and networking options.

The sort of architecture used by a CPU and a Raspberry Pi is one significant distinction. A CPU normally employs a **Complex Instruction Set Computer (CISC)** architecture, but the Raspberry Pi employs a **Reduced Instruction Set Computer (RISC)**. This means that a CPU can execute more instructions, but it may be slower and consume more power than a Raspberry Pi, which has a limited instruction set, yet can run faster and more effectively. Another distinction is that a CPU is normally a component of a larger computer system, but a Raspberry Pi is a stand-alone computer that can be used as a standalone device or as part of a larger system.

Since the first Raspberry pi model was launched in 2012, several different varieties of Raspberry Pi boards have been released. Here are some of the most popular models:

- **Raspberry Pi Model B:** The original Raspberry Pi board, launched in 2012, has since been decommissioned. It had a single-core processor running at 700 MHz, 256 MB of RAM, and a Broadcom Video Core IV GPU.
- **Raspberry Pi Model B+:** This was released in 2014 as an enhanced version of the original Model B. It had a slightly faster processor, 512 MB of RAM, and more USB and Ethernet connectors.
- **Raspberry Pi 2 Model B:** This was released in 2015 and included a 900 MHz quad-core processor, 1 GB of RAM, and a Broadcom Video Core IV GPU.
- **Raspberry Pi 3 Model B:** Released in 2016, this model represented a considerable upgrade over prior generations. It had a quad-core 1.2 GHz processor, 1 GB of RAM, and a Broadcom Video Core IV GPU.

- **Raspberry Pi 3 Model A+:** Released in 2018, this model was a scaled-down version of the Model B+. It had a single-core 1.4 GHz processor, 512 MB of RAM, and a Broadcom Video Core IV GPU.
- **Raspberry Pi 4 Model B:** This model, which was released in 2019, is the most recent and powerful Raspberry Pi board to date. It has a quad-core 1.5 GHz processor, up to 8 GB of RAM, and a Broadcom Video Core VI GPU.
- **Raspberry Pi 400:** This is a tiny desktop computer embedded inside a keyboard that was released in 2020. It has a quad-core 1.8 GHz processor, 4 GB of RAM, and a Broadcom Video Core VI GPU.

Figure 3.2 shows the components used in a standard Raspberry pi board:

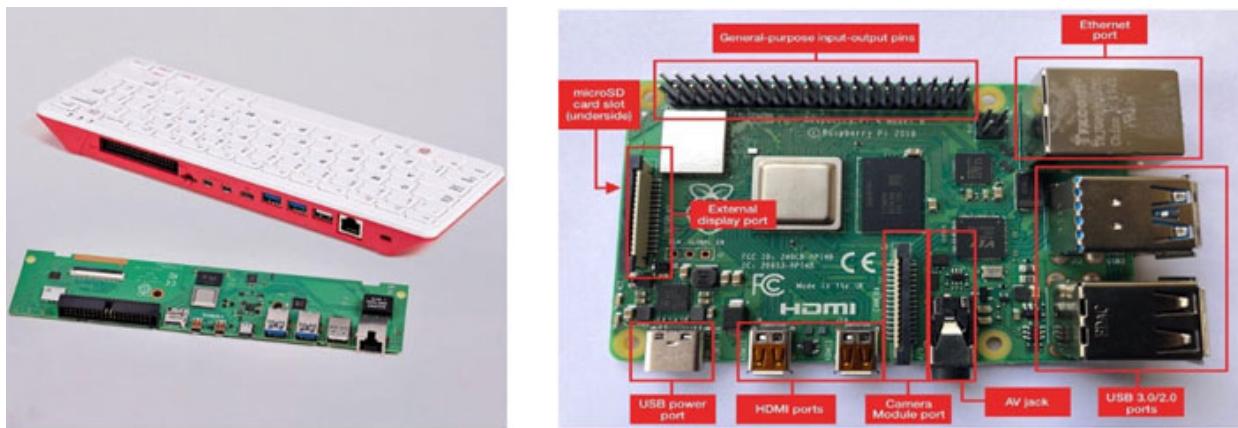


Figure 3.2: Raspberry pi board with its components and Raspberry Pi 400 model

Microcontrollers and Microcontroller with AI accelerator

The term "microcontroller" refers to a type of small, low-power computer used to manage various electronic systems. They are often coded in a specific language and consist of a processor, memory, and **input/output (I/O)** peripherals on a single **integrated circuit (IC)** or chip. They are designed to be embedded in other devices, and are commonly found in appliances, automobiles, industrial equipment, and many other types of electronic systems.

An AI accelerator is hardware specifically designed to speed up the execution of machine learning and artificial intelligence algorithms within a microcontroller. Microcontrollers such as these, are made to meet the

rigorous computing needs of AI and ML software while remaining remarkably energy-efficient. Facial recognition, speech recognition, and other forms of intelligent sensing and control are only some of the many uses for AI accelerators in microcontrollers.

The primary advantages of microcontrollers include:

- **Power Consumption:** MCs are milliwatt and microwatt level power consumers, making them low-power devices. Therefore, MCs are approximately a thousand times more energy efficient than a typical computer.
- **Price:** Over 28 billion microcontrollers were shipped in 2020, and they cost very little.
- **Multifunctional usage:** Microcontrollers can be used in all devices, gadgets, and appliances.

There are several ways to classify microcontrollers, including, by their architecture, memory size, clock speed, and (I/O) capabilities. One common way to classify microcontrollers is by their architecture, which refers to the underlying design of the microcontroller's CPU. Some common MC architectures are as follows:

- **8-bit microcontrollers:** These microcontrollers are typically the smallest and least expensive, and are used in simple, low-power applications. They have an 8-bit CPU and are typically limited to a few kilobytes of memory.
- **16-bit microcontrollers:** These microcontrollers have a 16-bit CPU and offer improved performance and more advanced features compared to 8-bit microcontrollers. They are often used in applications that require more computing power, such as in automobiles and industrial equipment.
- **32-bit microcontrollers:** These microcontrollers have a 32-bit CPU and offer even more advanced features and higher performance, compared to 16-bit microcontrollers. They are often used in high-end applications, such as in advanced medical devices and military systems.

Another way to classify microcontrollers is by their memory size, which refers to the amount of storage space available on the microcontroller, for

storing data and programs. Microcontrollers can have a wide range of memory sizes, ranging from a few kilobytes to several megabytes.

Microcontrollers can also be classified by their clock speed, which refers to the speed at which the microcontroller's CPU can execute instructions. Clock speeds for microcontrollers can range from a few kilohertz to several megahertz.

Finally, microcontrollers can also be classified by their I/O capabilities, which refer to the number and types of input/output channels available on the microcontroller. These channels can be used to communicate with external devices, such as sensors and actuators, and to send and receive data.

Figure 3.3 features the various components of MC and MCs from different manufacturers and sample AI Accelerator card:

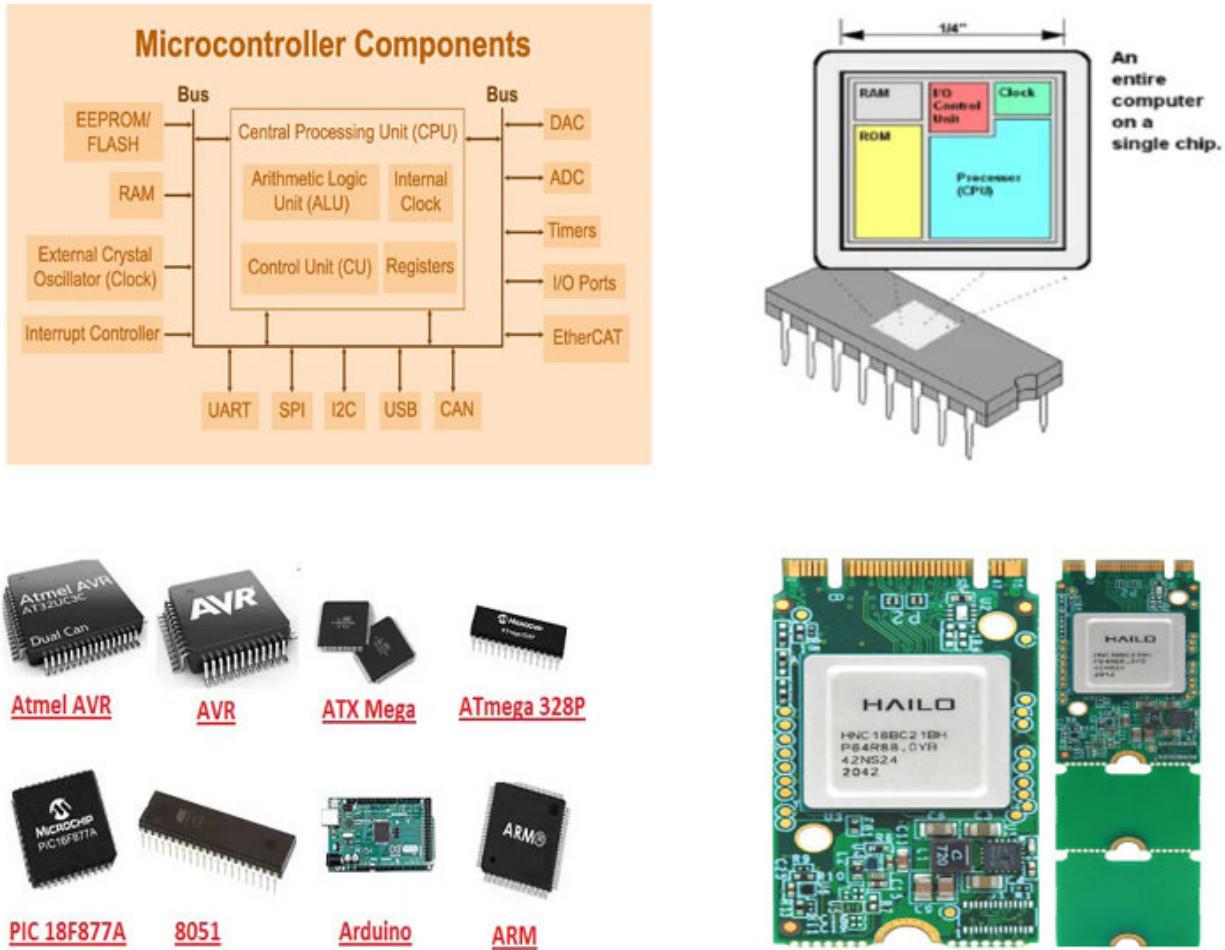


Figure 3.3: Components of MC, MCs from different manufacturers and sample AI Accelerator card

TinyML Hardware Boards

Many 32-bit microcontrollers with appropriate flash memory, RAM, and clock frequency can be used to execute machine learning models. In addition, the MC boards include many onboard sensors that may run any embedded program and apply machine learning models to the targeted use.

TinyML hardware boards are compact, low-power, single-board computers meant to execute machine learning algorithms at the edge. These boards typically contain a microcontroller as the CPU as well as technology designed to accelerate AI algorithms. Without the need for a separate AI accelerator or other specialized hardware, this enables the efficient and rapid execution of machine learning workloads. TinyML hardware boards are suitable for a variety of applications, including **Internet of Things (IoT)** devices, smart sensors, and wearable technology, where low power consumption and small size are essential. Without the requirement for a network connection or cloud computing, they can also be utilized for real-time processing of sensor data, such as object detection or speech recognition.

The Adafruit EdgeBadge, the SparkFun Edge, Raspberry Pi 4, the Coral Edge TPU, the Arduino Nano 33 BLE Sense and Syntiant TinyML board are some of the examples of TinyML hardware boards. These boards provide a comprehensive environment for executing TinyML algorithms and may be written with a variety of languages and frameworks. In addition, TinyML hardware boards are frequently quite inexpensive, making them accessible to a broad spectrum of users, including builders and hobbyists. Many TinyML hardware boards are open source, meaning that they are extremely adaptable and may be easily integrated into a variety of projects.

Here are some of the good things about TinyML hardware boards:

- **Small size and low power use:** TinyML hardware boards are made to be small and use little power. Because of this, they can be used in many situations where size and power are important.
- **Cost-effective:** TinyML hardware boards are usually less expensive than traditional ML hardware solutions. This makes them a good choice for companies that want to use ML.
- **Easy to use:** TinyML hardware boards are made to be user-friendly and easy to add to existing systems and apps. This makes it easier for

developers to get started and shortens the learning curve.

- **Flexibility:** TinyML hardware boards can be used in a wide range of applications, from IoT devices and wearables, to drones and robotics. This gives developers a flexible solution that can be adapted to a variety of situations.
- **Strong performance:** Even though TinyML hardware boards are small, they are still able to deliver strong performance. Many boards offer high-speed processing and advanced features such as deep learning and neural network support.

Following is the list of some of major MCs based hardware and TinyML hardware boards used for implementing different applications using machine learning algorithms.

- **Arduino Nano 33 BLE Sense:** This board features a powerful Cortex-M4F processor and Bluetooth Low Energy connectivity, thus making it ideal for developing tiny machine learning models.
- **Arduino Nicla Sense ME:** The Arduino Nicla Sense ME is a smallest form factor yet, with a range of industrial grade sensors packed into a tiny footprint.
- **Raspberry Pi:** The Raspberry Pi is a versatile, low-cost computer that can be used to build tiny machine learning models.
- **Adafruit Circuit Playground Express:** This board features an array of sensors, LEDs, and buttons, making it perfect for developing tiny machine learning projects.
- **SparkFun Edge:** This board features a powerful Arm Cortex-M4F processor, on-board sensor fusion, and Bluetooth Low Energy connectivity, making it ideal for building tiny machine learning models.
- **NVIDIA Jetson Nano:** This board features a powerful NVIDIA GPU, making it ideal for building and deploying machine learning models at the edge.
- **Google Coral:** A development board powered by a low-power AI processor that can be used for various TinyML projects.
- **Syntiant TinyML:** One of Syntiant's products is its TinyML platform, which is a suite of tools and technologies that enable the development

of machine learning models for edge devices with very low power consumption.

Arduino and Arduino Nano 33 BLE

Arduino is an open-source platform that makes it easy for people who are just starting out, to build a wide range of electronic devices. It does this by using microcontroller boards and a simple programming language. There are simple projects, such as making an LED blink, and more complicated ones, like making a robot or a weather station. Arduino is popular with hobbyists, students, and professionals because it is easy to use and can be adapted to a wide range of uses. The Arduino platform has both hardware and software parts. The hardware is made up of a microcontroller board and several sensors, motors, and displays, among other things. The Arduino **Integrated Development Environment (IDE)** software is used to write programs for the microcontroller board and send them to it.

Arduino also has a large and active community of users who share their projects, code, and tips online. This makes it easy to learn from others and get help when you need it. Moreover, Arduino is very flexible and can be used with a wide range of sensors and other parts, to make projects as simple or as complicated as the user wants. Because of this, people can make anything from simple LED displays to complex robotics projects. Overall, Arduino is a strong and flexible platform that is used for many kinds of electronic projects. It is great for both new and experienced makers because it is easy to use, has a friendly programming environment, as well as an active community.

The Arduino Nano 33 BLE Sense is a small but powerful microcontroller board with a **Bluetooth Low Energy (BLE)** module built in. It is based on the Arduino Nano 33 BLE board and includes inbuilt sensors such as a 9-axis **Inertial Measurement Unit (IMU)**, a microphone, and a pressure sensor. As a result, it is perfect for developing a wide range of IoT projects that require Bluetooth connectivity and sensor data. Tracking physical activity, voice control, monitoring environmental variables, and designing user interfaces with touch and gesture controls are some of the probable applications for the Nano 33 BLE Sense.

Figure 3.4 features the various components of Arduino Nano BLE Sense board:

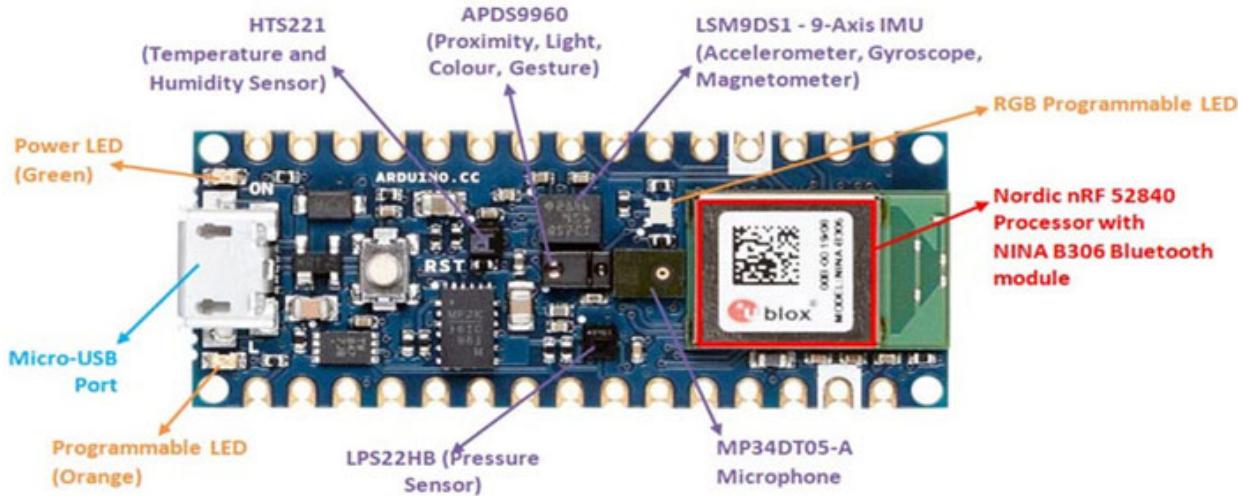


Figure 3.4: Components of Arduino Nano BLE Sense board

Arduino Nicla Sense ME

The Nicla Sense ME is a small, low-power tool that redefines intelligent sensing technologies. The board combines the following four cutting-edge sensors from Bosch Sensortec, with the ease of integration and scalability of the Arduino ecosystem:

- BHI260AP motion sensor system with integrated AI.
- BMM150 magnetometer.
- BMP390 pressure sensor.
- BME688 4-in-1 gas sensor with AI and integrated high-linearity, as well as high-accuracy pressure, humidity and temperature sensors.

The "M" and "E" in the acronym refer to the ease with which **Motion** and the surrounding **Environment** can be analyzed. The sensors in the board monitor rotation, acceleration, pressure, humidity, temperature, air quality, and CO₂ levels. Its small size and durable architecture make it ideal for projects requiring sensor fusion and AI capabilities at the edge. [Figure 3.5](#) features a sample image of Nicla Sense ME Board:



Microcontroller	64 MHz Arm® Cortex M4 (nRF52832)
Memory	512KB Flash / 64KB RAM
Sensors	<ul style="list-style-type: none"> BHI260AP - Self-learning AI smart sensor with integrated accelerometer and gyroscope BMP390 - Digital pressure sensor BMM150 - Geomagnetic sensor BME688 - Digital low power gas, pressure, temperature & humidity sensor with AI
Interface	USB interface with debug functionality
Dimensions	22,86 mm x 22,86 mm

Figure 3.5: Nicla Sense ME board

Adafruit Feather

The Adafruit Feather board is a microcontroller board that is being used in a variety of DIY projects. Because it is an open-source board, it may be easily changed and adjusted to meet the demands of the user. The Adafruit Feather board's compact size and lightweight design are major benefits. Its small size of 20mm x 51mm makes it excellent for usage in small projects and wearable devices. Furthermore, the board is designed to be easily stackable, allowing for the usage of numerous boards in a single project. The Adafruit Feather board is driven by an ARM Cortex-M0+ processor, which enables high-speed performance while consuming little power. It also has several built-in sensors and peripherals, including an accelerometer, a temperature sensor, and a real-time clock.

The Adafruit Feather board supports a variety of connecting choices, which is one of its distinguishing qualities. It supports Bluetooth, Wi-Fi, and LoRa, allowing for easy connectivity with other devices. Furthermore, the board is compatible with several popular development environments, including Arduino and CircuitPython, making it simple for users to program and personalize their creations. [Figure 3.6](#) features a sample image of Adafruit Feather board:

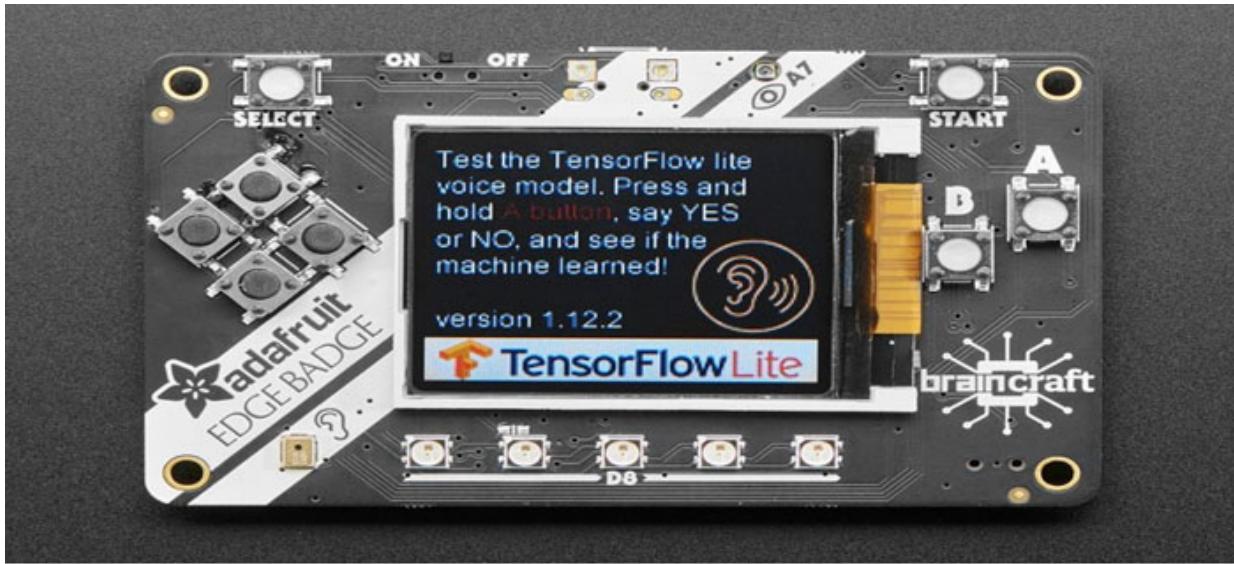


Figure 3.6: Adafruit Feather board

SparkFun Edge

The SparkFun Edge board was created exclusively for running TinyML apps. It is powered by an Arm Cortex-M4F processor. The SAMD51 microprocessor powers it, and it includes an integrated BOSCH BMA421 accelerometer and BME280 environmental sensor. In addition, the board includes a MicroSD card slot for data storage and a Qwiic connector for easy connection to other Qwiic-enabled sensors and devices. The SparkFun Edge is suitable for constructing lightweight, portable, and energy-efficient TinyML applications due to its compact size and low power consumption.

It also has an Arm Cortex M4F microprocessor and an Edge Impulse **Machine Learning (ML)** accelerator on board, allowing users to design and run ML models directly on the device. The Edge also features multiple communication choices, including Wi-Fi, Bluetooth, and USB, making it easy to incorporate into existing IoT systems. It works with a variety of programming languages and frameworks, giving it a versatile platform for developing and delivering TinyML applications. *Figure 3.7* features a the SparkFun Edge Board:

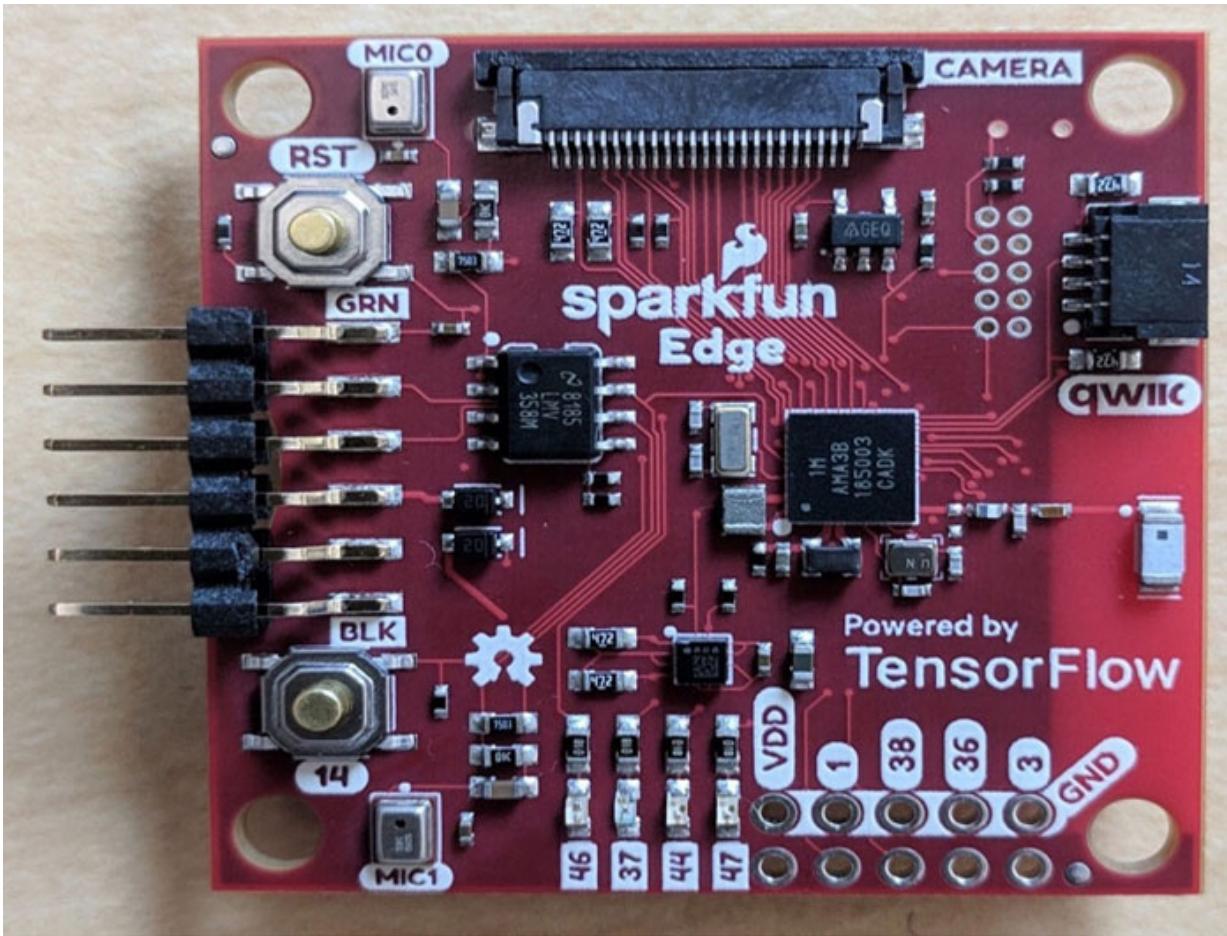


Figure 3.7: SparkFun Edge board

NVIDIA Jetson Nano

The Nvidia Jetson Nano is a compact, powerful computer built for machine learning and **Artificial Intelligence (AI)** applications. It is built around the Nvidia Jetson Xavier processor and has a 128-core NVIDIA Maxwell GPU, 4 GB of LPDDR4 memory, and several ports and networking options. The Jetson Nano is well-suited for image and video processing, robotics, and **Internet of Things (IoT)** projects. Because of its inexpensive cost and high performance, it is a popular choice among developers, researchers, and hobbyists. The Jetson Nano has the following main features:

- 128-core NVIDIA Maxwell GPU
- Quad-core Arm Cortex-A57 CPU
- 4 GB LPDDR4 memory

- HDMI 2.0 and eDP 1.4 display outputs
- MIPI CSI-2 camera connector

The Jetson Nano is simple to use and works with a wide range of software libraries and programming tools. It contains a JetPack SDK with support for prominent machine learning frameworks such as TensorFlow and PyTorch, as well as a variety of libraries and tools for computer vision and robotics applications. [*Figure 3.8*](#) features the Jetson Nano Developer Kit:



Figure 3.8: Jetson Nano Developer Kit

Google Coral Edge TPU

Google Coral Edge TPU is a small but powerful AI accelerator developed for use with the Google Edge TPU platform. It has a unique TPU designed for running machine learning models on edge devices such as cameras and IoT sensors. The Coral Edge TPU provides excellent performance while consuming little power, making it perfect for applications that demand real-time AI processing at the edge. It is also compatible with the TensorFlow Lite and TensorFlow frameworks, making it simple to incorporate into AI applications. Overall, the Coral Edge TPU is a strong and efficient accelerator of AI on edge devices. Coral Edge TPUs can be utilized to speed

up a variety of AI applications, such as image and video analysis, speech recognition, and NLP. Moreover, they are intended to be low-power and compact, making them ideal for usage in devices such as drones and smartphones.

[Figure 3.9](#) features the Google Coral Edge TPU:



Figure 3.9: Google Coral Edge TPU

[Qualcomm QCS605](#)

The Qualcomm QCS605 is a **System on a Chip (SoC)** that can be used in a variety of devices, such as IoT devices and smart home appliances. It is based on the Qualcomm Kryo CPU, which is known for being fast and usage of little power. This makes it a good choice for devices that run on batteries. The QCS605 also has an Adreno GPU for better graphics performance and support for Wi-Fi, Bluetooth, and Zigbee wireless connection technologies. It has powerful computing abilities, an AI engine built in, and support for machine learning on the device. The QCS605 also lets you record and play back 4K videos. This processor is makes the user's experience smooth and easy, with quick and responsive performance and low power use.

The QCS605 SoC-powered devices can use TinyML to do things such as image and voice recognition, gesture control, and predictive maintenance, without a cloud connection or a powerful processor. This makes it possible to make smart, connected devices that can work even in places that are far away or have few resources. [Figure 3.10](#) features the Qualcomm QCS605 is a **System on a Chip (SoC)**:

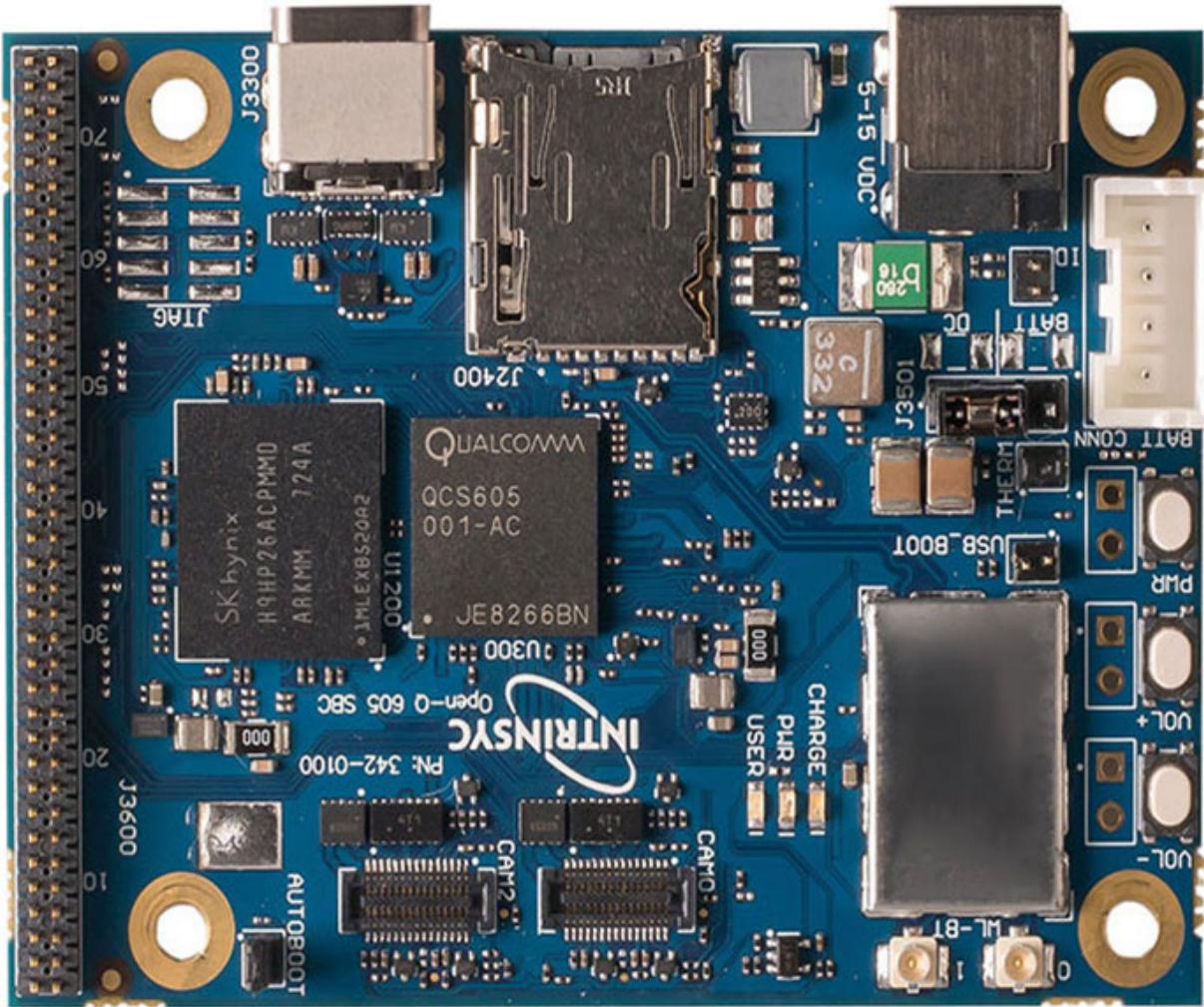


Figure 3.10: Qualcomm QCS605 is a System on a Chip (SoC)

NXP i.MX 8M

The NXP i.MX 8M is a microprocessor series based on the Arm Cortex-A53 that is intended for use in embedded systems. These microprocessors are ideal for applications that require high performance while consuming little power, such as IoT devices and other compact form-factor devices. The i.MX 8M TinyML is a version of the i.MX 8M developed primarily for machine learning applications such as edge computing and on-device AI. It is a low-cost solution for developers who want to design and deploy machine learning models on small, low-power devices. [Figure 3.11](#) features a NXP i.MX 8M:

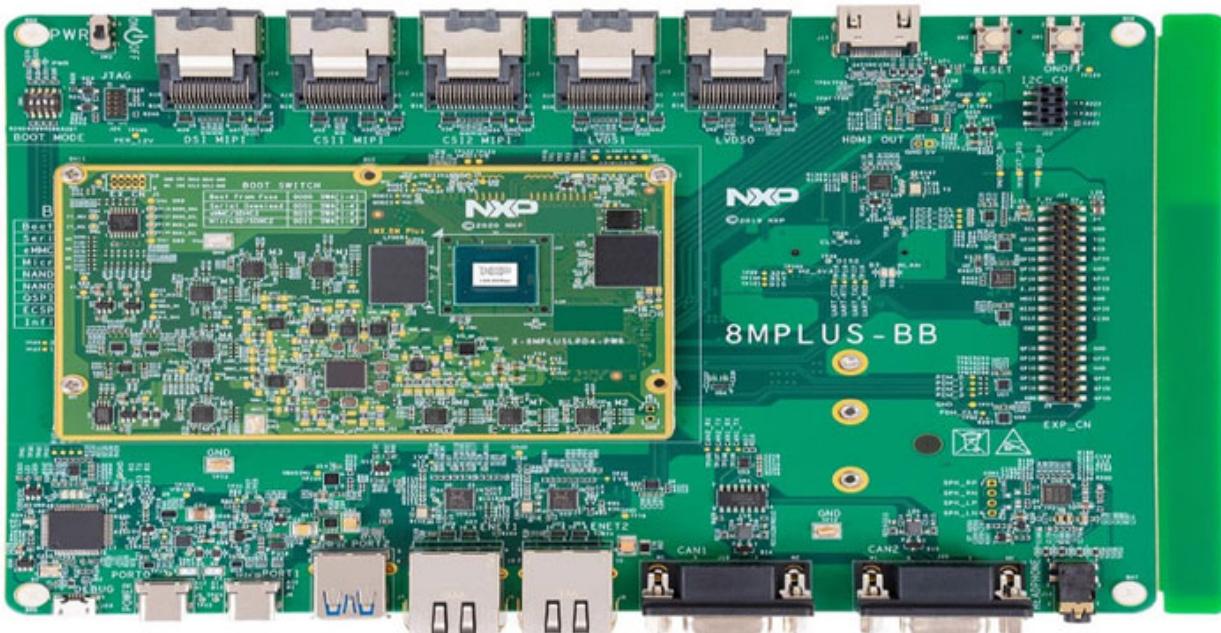


Figure 3.11: NXP i.MX 8M

STMicroelectronics STM32L4

The STM32L4 line of ultra-low-power microcontrollers from STMicroelectronics is based on the Arm Cortex-M4 CPU. They are intended for use in a variety of applications, including IoT devices, smart home appliances, and industrial control systems. The STM32L4 family provides great performance and a diverse collection of peripherals while consuming little power and lasting a long time on a single charge. The STM32L4 microcontrollers have the following essential features:

- Arm Cortex-M4 core with a maximum clock frequency of 80 MHz.
- Up to 1 Mbyte of Flash memory and 320 Kbytes of SRAM.
- Rich set of peripherals, including USB OTG, SDMMC, and multiple low-power modes.
- Support for a wide range of development tools, including STM32CubeIDE and various third-party IDEs.
- High-resolution **Analog-to-Digital Converter (ADC)** for precise measurement and data conversion.
- DSP instruction set for enhanced audio and digital signal processing.
- Advanced low-power modes for energy-efficient operation.

- Integrated LCD controller for easy display of text and graphics.
- Large memory options with up to 2 MB of Flash and 640 KB of SRAM.

[Figure 3.12](#) features the STM32L4 ultra-low-power microcontroller:

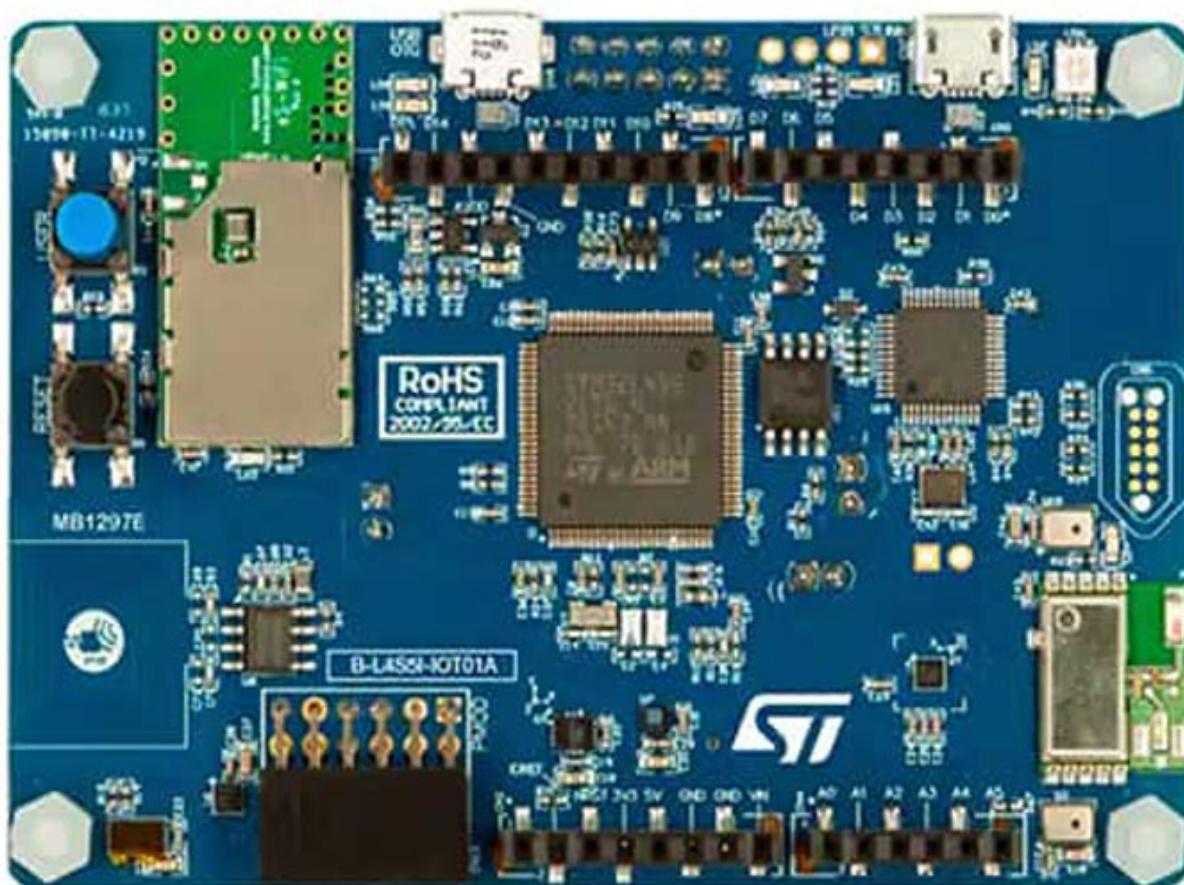


Figure 3.12: Sample image of STM32L4 ultra-low-power microcontroller

Intel Curie

Intel Curie is a microcontroller SoC designed by Intel for wearable devices and IoT applications. It has a low-power, 32-bit Quark SE microcontroller, BLE connectivity, 6-axis motion sensors, and a sensor hub that is constantly on. Because the Curie module is compact and lightweight, it is ideal for usage in wearable devices such as fitness trackers and smartwatches. It can also process machine learning algorithms, which can be used for a variety of

applications such as gesture recognition and activity tracking. [Figure 3.13](#) features the Intel Curie:

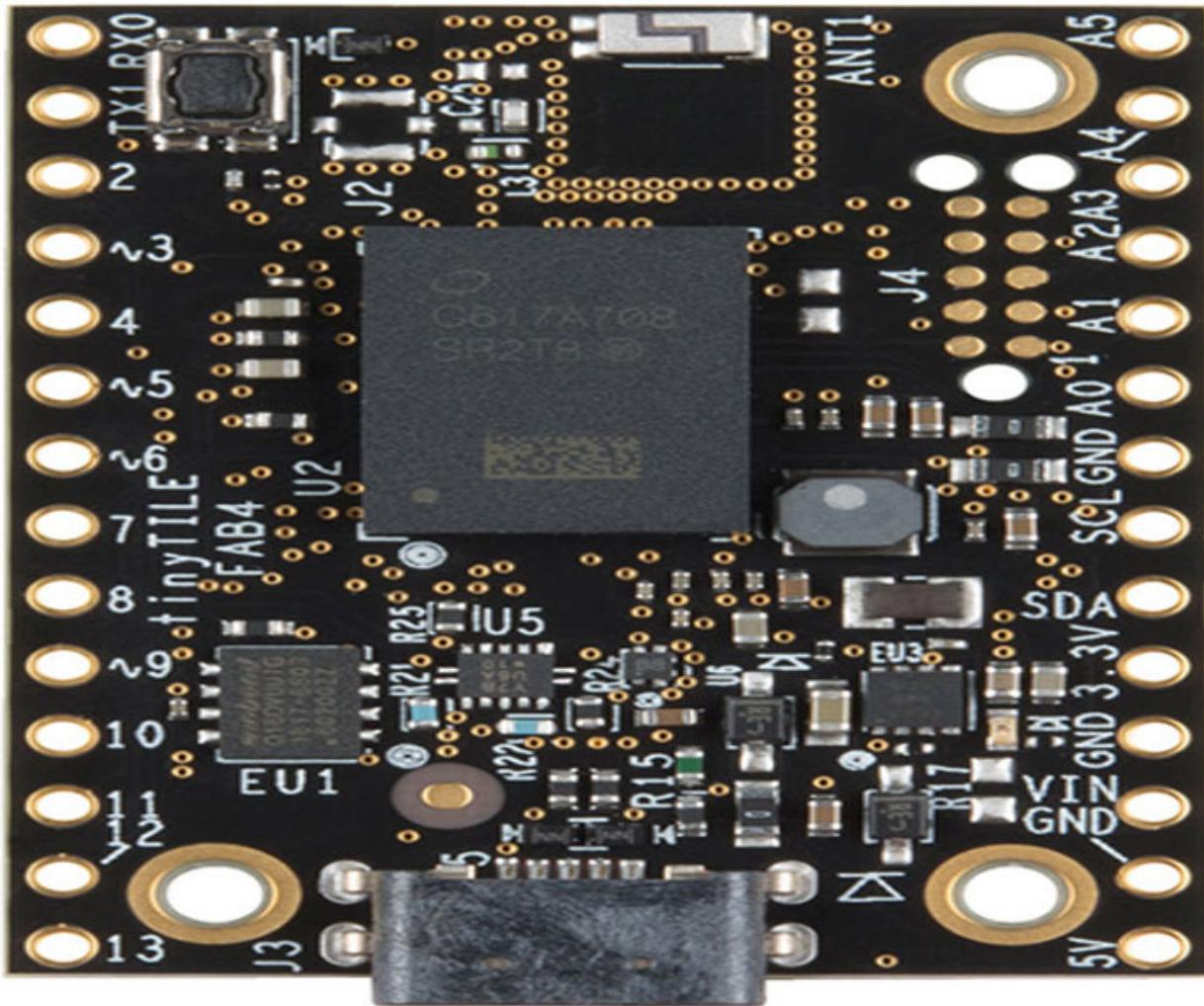


Figure 3.13: Sample image of Intel Curie is a microcontroller system-on-chip (SoC)

Syntiant TinyML

Syntiant's **Tiny Machine Learning (TinyML)** Development Board is a great platform for developing low-power voice, **Acoustic Event Detection (AED)**, and sensor ML applications. Anyone interested in trying out, testing, or creating using Syntiant's tech, can do so with the help of the development kit.

The TinyML board incorporates the ultra-efficient Syntiant NDP101 Neural Decision Processor to provide native neural network processing for even the most resource-intensive applications, while still maintaining an extremely

low power footprint. Edge Impulse and Syntiant have worked together to release this board. This hardware has the backing of Edge Impulse. Whether it is for voice recognition, **Automated External Defibrillation (AED)**, or sensing 6-axis motion and vibration, the integrated microphone and BMI160 sensor make the entire setup process a breeze. The TinyML board allows for the simple download of trained models through micro-USB.

The key features of the Syntiant board are as follows:

- Neural Decision Processor: NDP101
- Host processor: SAMD21 Cortex-M0+ 32bit low power ARM MCU, including:
 - 256KB flash memory, 32KB host processor SRAM
- 5 Digital I/Os compatible with Arduino MKR series boards
- 2MB on-board serial flash, 48MHz system clock
- BMI160 6 axis motion sensor
- SPH0641LM4H microphone

Some potential applications for Syntiant's TinyML technology include enabling devices to perform tasks such as image and speech recognition, language translation, and predictive maintenance, among others. [Figure 3.14](#) features the Syntiant TinyML board:

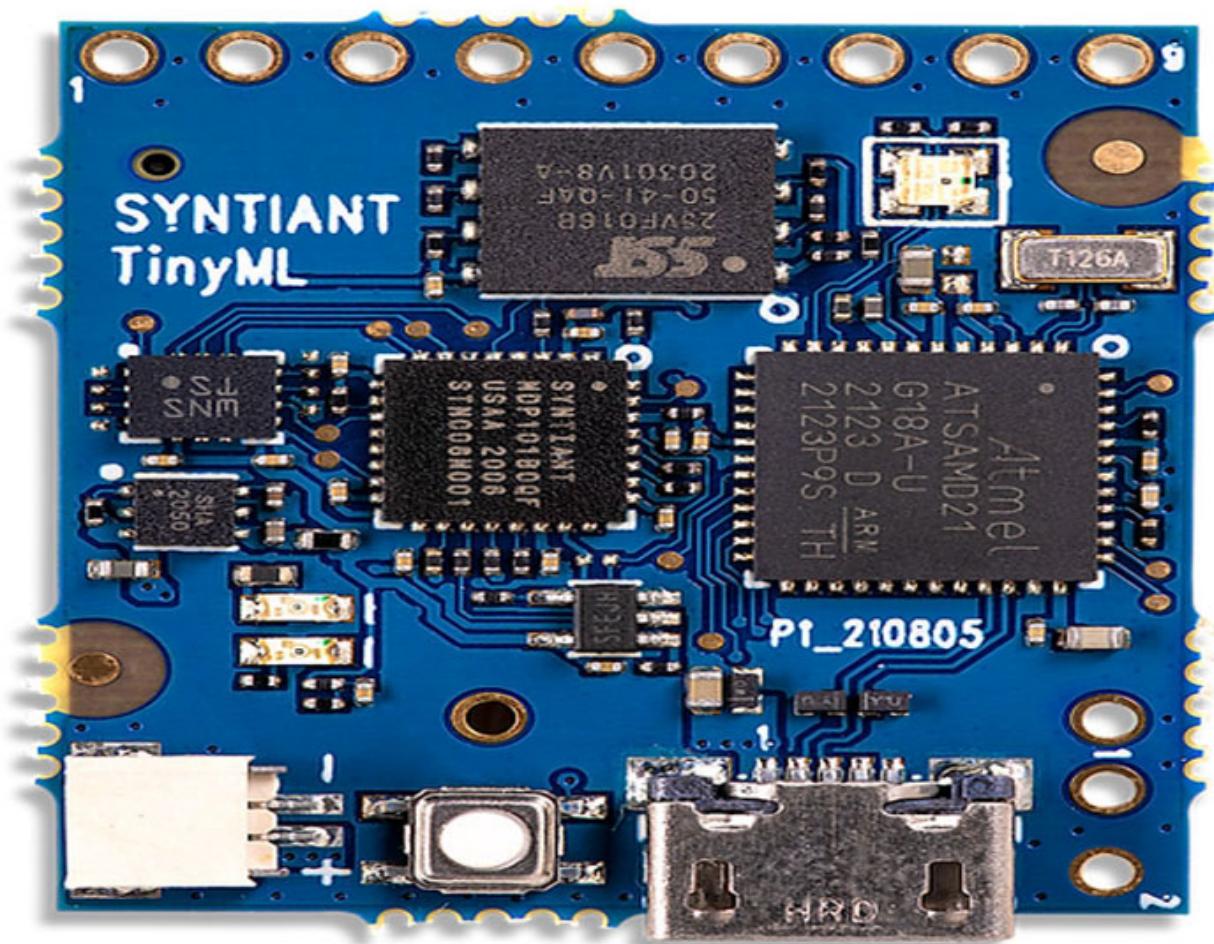


Figure 3.14: Sample image of Syntiant TinyML board

TinyML Software Suites

TinyML is the application of machine learning techniques to low-power computing platforms. These algorithms can be executed locally on the device and used for decision making in real time without requiring communication with a server.

Several different software packages, such as TensorFlow Lite, Edge Impulse, and Zerynth, can be used to create TinyML applications. Suites such as these, make it possible to develop and deploy machine learning models on low-power devices by providing the necessary tools and libraries. Some of these bundles also provide access to pre-trained models and other resources that can be utilized as a foundation for deploying ML models on MCs.

Some of the popular TinyML software tools are as follows:

- **TensorFlow Lite**: An open-source framework for deploying machine learning models on edge devices.
- **Arm Cortex-M microcontroller-based machine learning**: A suite of tools and libraries for implementing machine learning algorithms on Arm Cortex-M microcontrollers.
- **Zephyr Project**: An open-source real-time operating system for building and deploying machine learning algorithms on IoT devices.
- **Arduino and Arduino Create**: An open-source platform for building and programming electronic devices, including those with machine learning capabilities. Create is an online platform for creating and deploying IoT applications, including those using TinyML algorithms.
- **Edge Impulse**: A platform for developing and deploying machine learning algorithms on edge devices.
- **OpenMV**: An open-source hardware and software platform for developing vision-based applications on microcontrollers.
- **PYNQ**: This is an open-source framework for developing ML algorithms on Xilinx **Field-Programmable Gate Arrays (FPGAs)**.
- **MicroPython**: An open-source Python interpreter that can be used to develop and run TinyML algorithms on microcontrollers and other low-power devices.

TensorFlow Lite Micro (Google)

For deploying machine learning applications to mobile and other edge devices, TensorFlow Lite provides a lightweight framework. MCs are typically compact, low-power devices that are used in IoT applications, and hence, TensorFlow Lite Micro is a version of TensorFlow Lite tailored exclusively for them. Developers can now deploy machine learning models onto low-powered devices.

TensorFlow Lite Micro is a C++ implementation of TensorFlow that has been optimized for use on microcontrollers by virtue of its simplicity, tiny footprint, and straightforward API. [Figure 3.15](#) features the TensorFlow Lite Micro for Microcontroller deployment:

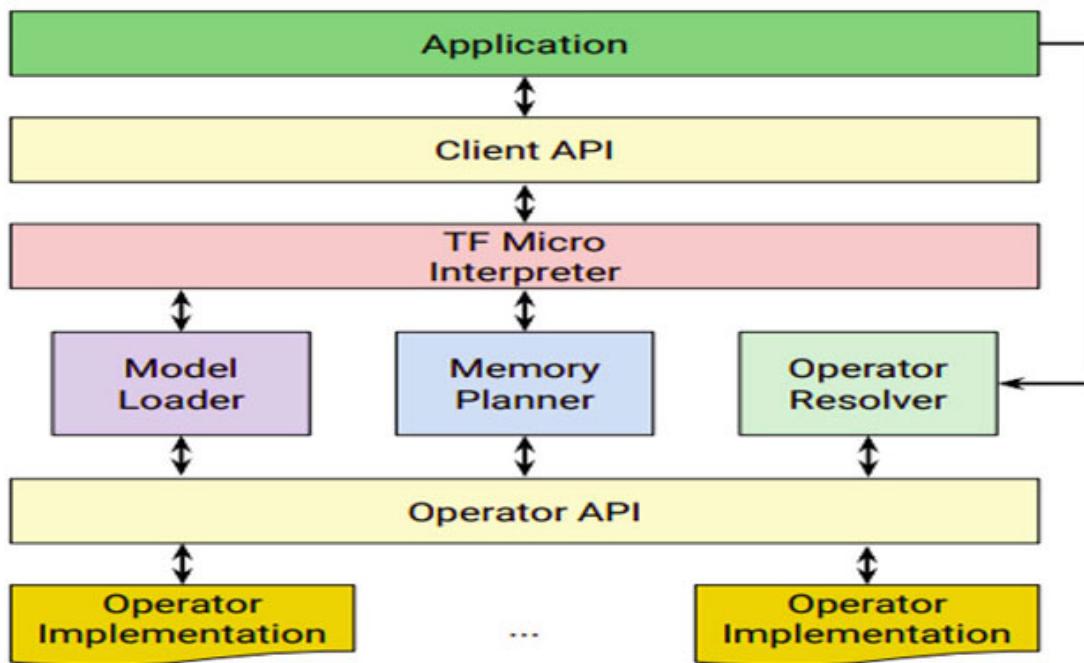
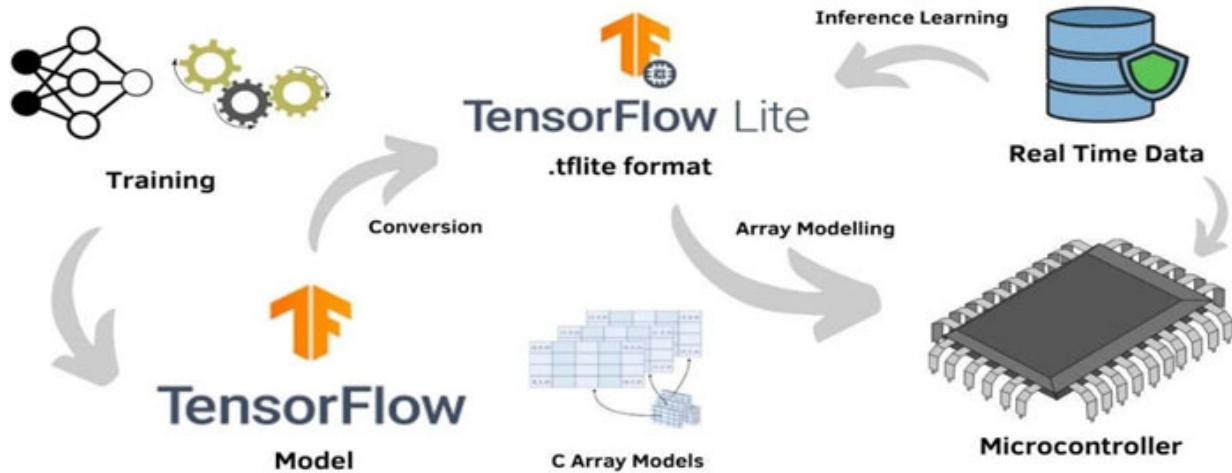


Figure 3.15: TensorFlow Lite Micro for Microcontroller deployment and steps involved in the deployment

uTensor (ARM)

To put it simply, uTensor is a microcontroller-friendly, lightweight deep learning inference framework. It enables programmers to execute machine learning models on microcontrollers, bringing AI functionality to low-power devices such as sensors, smart watches, and other IoT devices.

By enabling the execution of machine learning models locally on microcontrollers, uTensor facilitates the creation of intelligent and responsive IoT systems. This has the potential to enhance the security and dependability of IoT systems, while simultaneously decreasing their power consumption and response time. As a lightweight and efficient microcontroller platform, uTensor is ideal for **Internet of Things (IoT)** and other embedded systems, as it is based on the Arm Cortex-M microcontroller series.

[Arduino Create](#)

Arduino Create is an online environment for designing and writing code for Arduino-based projects. It comes with an IDE that lets you code, upload it, and debug it, as well as leverage a wide range of Arduino-specific tools and resources.

Create allows users to write code in C or C++ with the Arduino programming language, and then uploads that code to an Arduino board through a USB cable or a wireless connection. The platform also features a simulator that lets users test their code without the need for physical hardware and a library of pre-written code snippets and libraries that they may use into their projects.

Anyone curious about electronics and programming, as well as seasoned makers searching for a web-based platform for construction and prototyping, will find Arduino Create to be an invaluable resource. It is free for anybody to use, and you can even pay to unlock premium content and tools.

[EloquentML](#)

Using the Arduino Software Environment IDE, this library makes it easier to transfer models developed in TensorFlow Lite for Microcontrollers to boards. Eloquent TinyML's goal is to let programmers create smart, low-power apps that can run on microcontrollers, IoT devices, and edge computing hardware. Eloquent TinyML allows programmers to take advantage of machine learning to enhance the features and performance of their devices through the use of either pre-trained models or user-created models.

EdgeML (Microsoft)

EdgeML is a framework for machine learning that may be used on edge devices such as mobile phones and IoT devices. It is meant to facilitate the training and deployment of machine learning models on edge devices in a manner that is both efficient and scalable. With a focus on low-latency and energy-efficient inference, EdgeML offers a suite of techniques and tools for training, quantizing, and deploying machine learning models on edge devices. These tools facilitate a wide variety of uses, including picture and speech recognition, object detection, and anomaly detection, by allowing developers to rapidly create and deploy machine learning models on edge devices. As a result of Microsoft Research India's Intelligent Devices Expedition, this library was created. The goal of this library is to advance the state of the art in machine learning in order to provide low-resource computing devices with privacy-preserving, energy-efficient, off-grid intelligence.

EON Compiler (Edge Impulse)

When comparing **Edge Optimized Neural (EON)** to TensorFlow Lite for Microcontrollers, you will find that EON uses 25-55% less RAM and up to 35% less flash to execute a neural network with the same accuracy. While other embedded solutions rely on generic interpreters, EON eliminates complex code, device power, and valuable time by compiling your neural networks to C++.

The TensorFlow Lite for Microcontrollers operator kernels are used by EON Compiler, which efficiently invokes them without the need for an interpreter. It is possible that this will reduce memory usage by as much as half. It uses libraries such as Arm's CMSIS-NN automatically, when applicable, and it applies device-specific optimized kernels. [Figure 3.16](#) features the EON Compiler software platform for implementing TinyML:

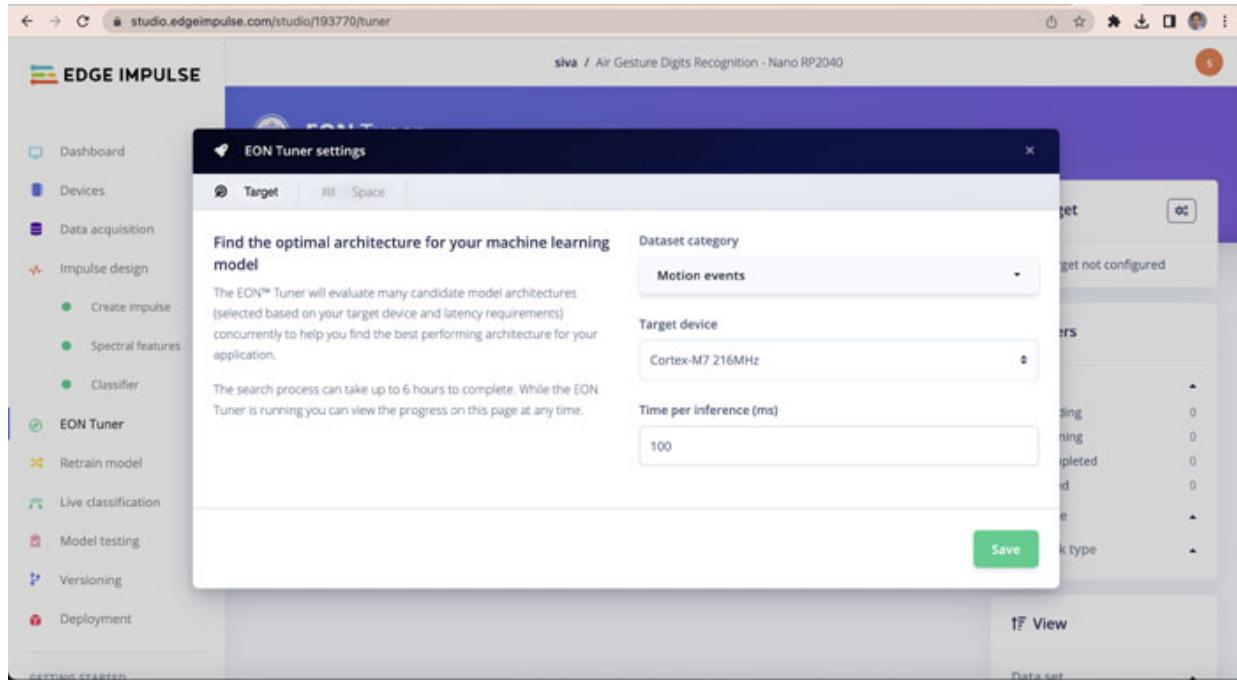


Figure 3.16: EON Compiler software platform for implementing TinyML

STM32Cube.AI and NanoEdge AI Studio (STMicroelectronics)

STMicroelectronics has developed a **Software Development Kit (SDK)** called STM32Cube.AI that can be used to learn and deploy AI algorithms on STM32 microcontroller hardware. It is integrated into the STM32Cube software ecosystem, a full suite of resources for developing applications for STM32 microcontrollers. By including pre-trained machine learning models, a graphical user interface for training and deploying models, and integration with the STM32CubeMX tool for generating code, STM32Cube.AI aims to simplify the use of AI approaches in STM32 applications. It integrates with well-known AI frameworks such as TensorFlow Lite and Keras.

NanoEdge AI Studio is an AI modeling and deployment platform. A developer or data scientist does not need to be a programmer to quickly build and deploy AI models on this platform. NanoEdge AI Studio simplifies and streamlines the process of developing and deploying AI models through the use of cutting-edge machine learning techniques and an intuitive graphical user interface. NanoEdge AI Studio is a great tool for businesses that want to improve their AI skills since it allows their users to quickly and simply incorporate AI into their current apps and systems.

Figure 3.17 features NanoEdge AI Studio software tool for implementing different application:

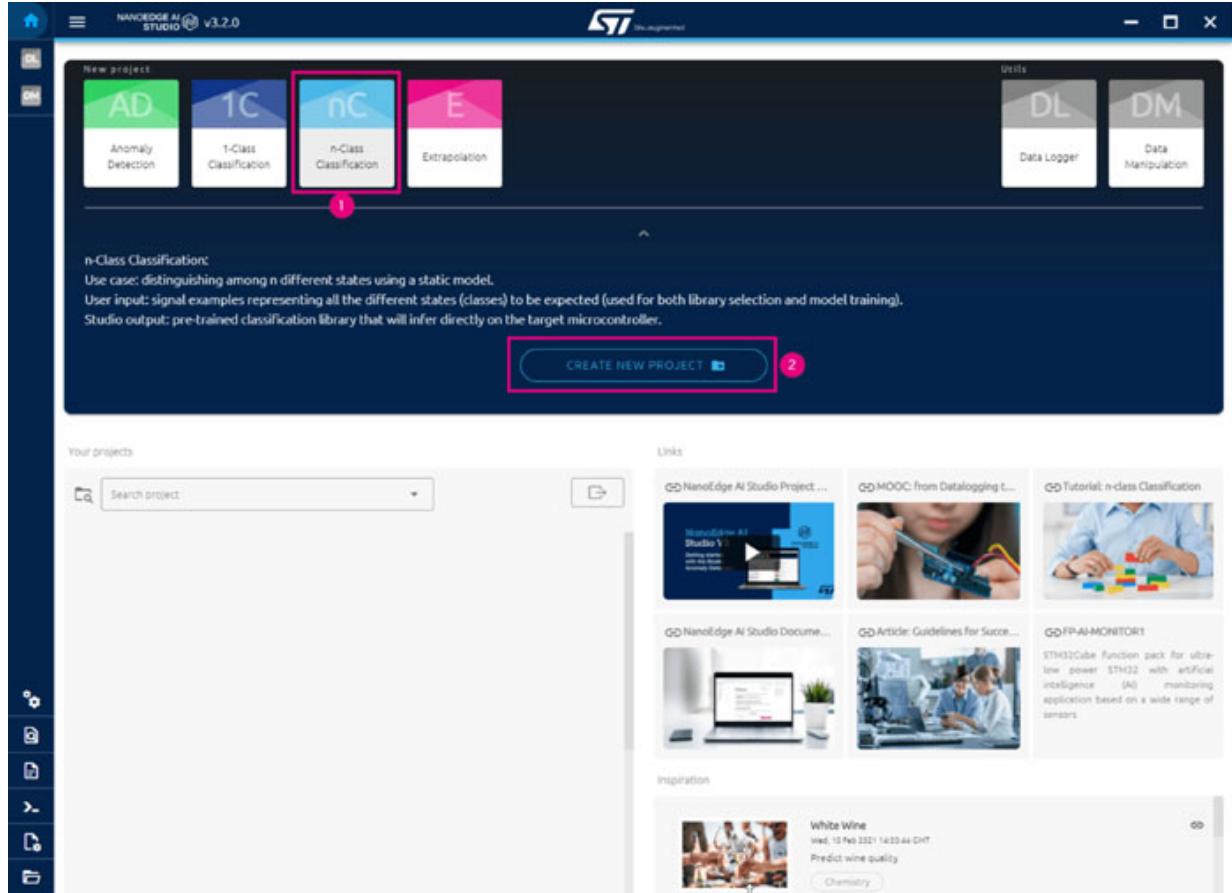


Figure 3.17: NanoEdge AI Studio software tool for implementing different applications

PYNQ

Xilinx's PYNQ (Python productivity for Zynq) project is an open-source initiative with the goal of serving as a framework for the creation and distribution of Python-based applications running on Xilinx FPGAs. Instead of using low-level hardware description languages such as VHDL or Verilog, it enables users to take advantage of the FPGA's power and parallelism, by utilizing high-level programming languages and libraries, such as Python. Using Python and the machine learning libraries PyTorch and scikit-learn, users of PYNQ can design their own hardware accelerators for tasks such as image processing and ML inference. Furthermore, PYNQ offers a variety of pre-built hardware overlays, which are FPGA designs optimized to accelerate a particular activity in hardware.

PYNQ is implemented on the Xilinx Zynq SoC, which integrates a dual-core ARM Cortex-A9 processor with an FPGA. Included in the PYNQ framework is a Linux system, the PYNQ libraries, and the Jupyter Notebook environment for rapid prototyping and development. All in all, PYNQ provides a robust and versatile framework for creating and implementing unique hardware acceleration solutions on FPGAs, allowing customers to speed up their applications, lower their power consumption, and keep the freedom to develop software. [Figure 3.18](#) features the PYNQ hardware board:

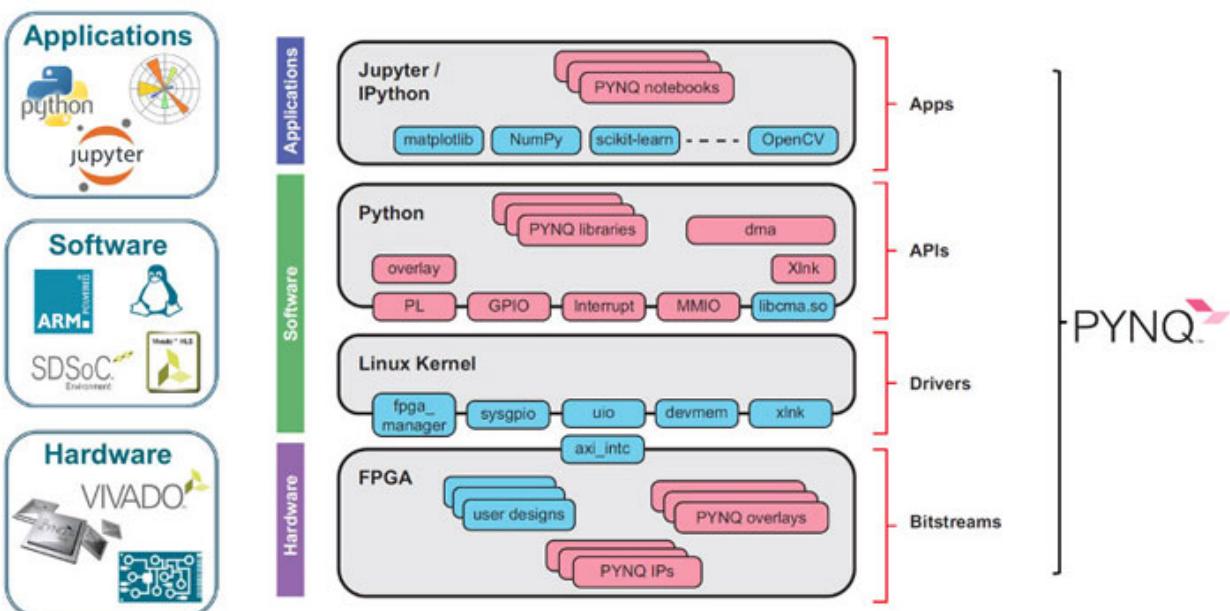
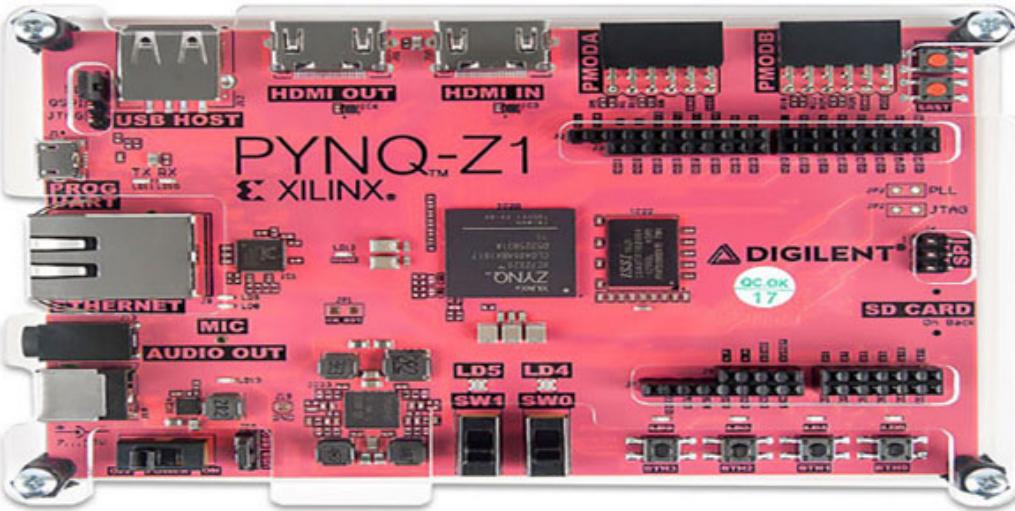


Figure 3.18: PYNQ hardware board and its software library high level architecture

OpenMV

OpenMV is a firm that develops free and open-source computer vision hardware and software. As an example, one of their products is a microcontroller board equipped with a camera module that can be used for object detection and tracking, barcode reading, and other tasks. The boards are Arduino-compatible, meaning they may be used in a wide range of projects, from robotics to security to factory automation.

Figure 3.19 features the OpenMV IDE for implementing TinyML applications:

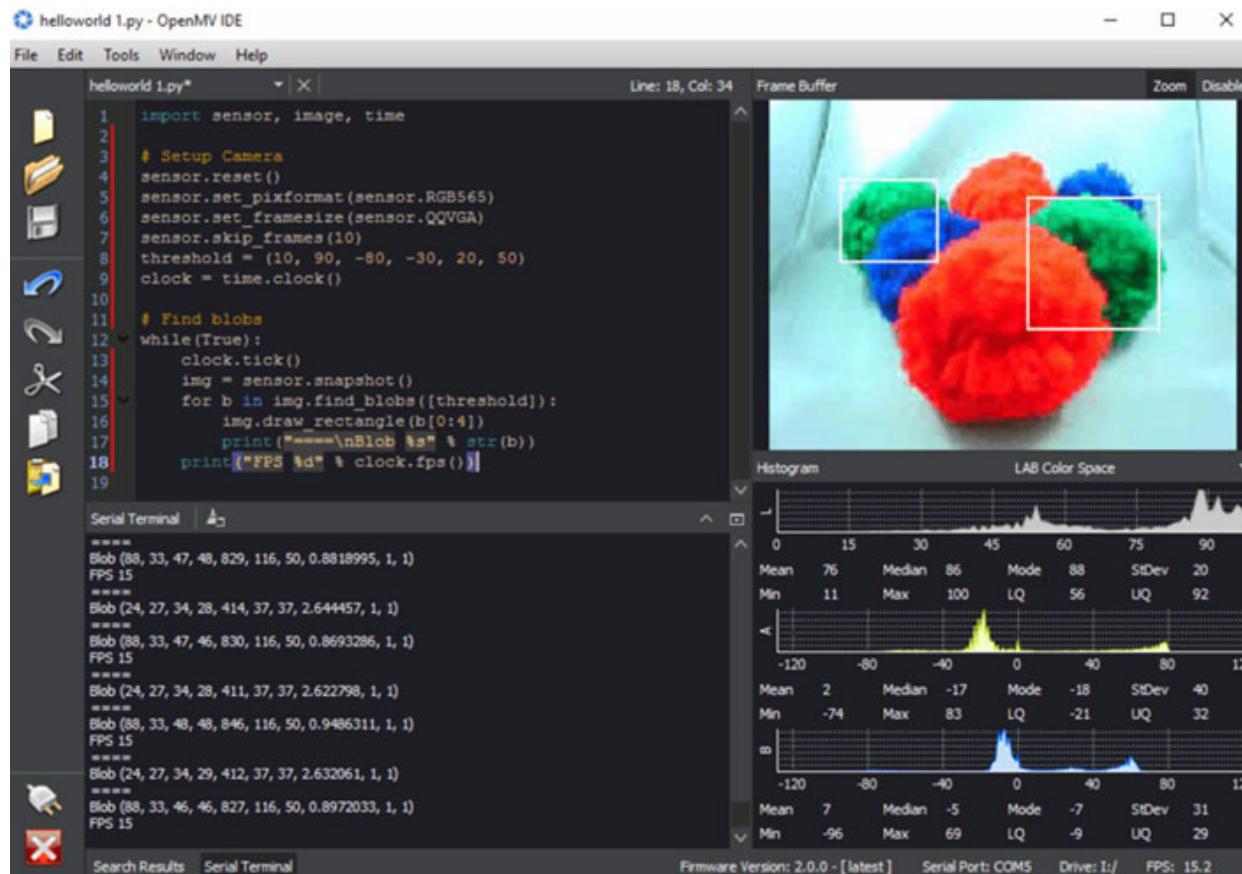


Figure 3.19: OpenMV IDE for implementing TinyML applications

SensiML

SensiML is a software platform that aids in the development of machine learning models for microcontrollers and other low-power devices. These models allow low-powered devices to carry out complex tasks, including

data classification, anomaly detection, and predictive maintenance. With this platform, developers can easily train and deploy machine learning models on low-powered devices, thanks to a workflow-based approach. Automatic features include feature extraction, model generation, and code development for many microcontroller architectures. With SensiML, programmers may streamline the process of building smart, networked devices that can make instantaneous decisions using information gathered from sensors and other sources.

Figure 3.20 features the SensiML development platform:



Figure 3.20: SensiML development platform

Neutron TinyML

Neutron is a firm that provides TinyML solutions for several industries. They provide a variety of microcontrollers, development boards, and software tools for use in creating customized TinyML solutions. In addition to providing developers with TinyML and the training and assistance, they also need to utilize the language effectively. Neutron also offers training and support for its other products. *Figure 3.21* features the Nueton TinyML development platform:

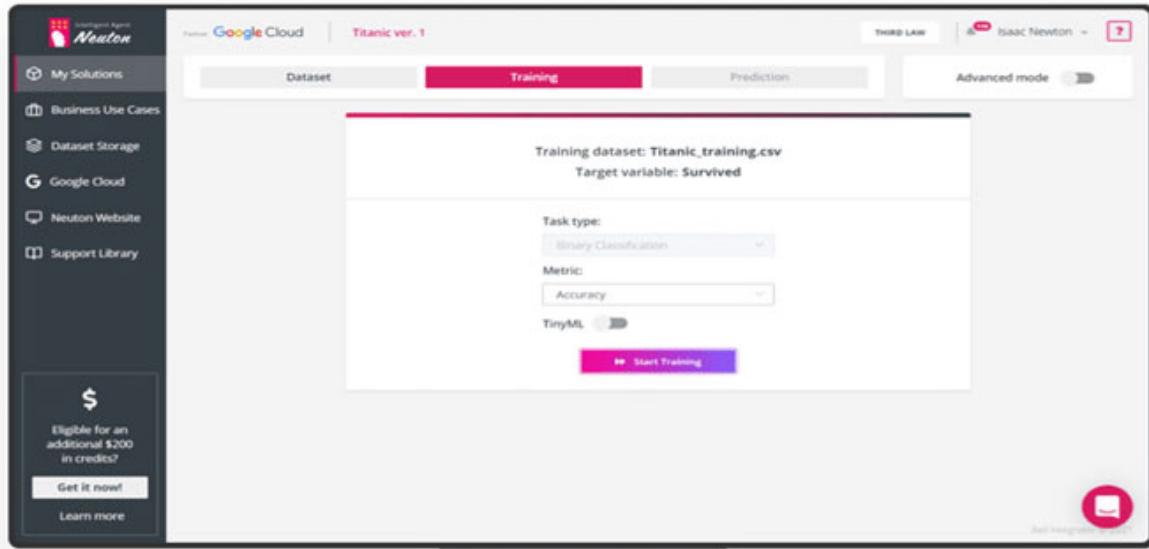


Figure 3.21: Nueton TinyML development platform

Metavision Intelligence Suite 3.0 (Vision applications)

Engineers in many fields, including manufacturing, the IoT, surveillance, mobile, medicine, the automobile industry, and more, will find this cutting-edge toolbox invaluable. Metavision Intelligence 3.0 offers a complete machine learning toolbox as part of its free modules, which can be accessed via C++ and Python APIs. The suite's no-code option lets users play freely available pre-recorded datasets without needing to purchase an event camera. In a matter of seconds, users can live-stream or capture events with the help of an event camera.

The suite has a total of 95 algorithms, 67 code samples, and 11 applications. High-speed counting, vibration monitoring, splatter monitoring, object tracking, optical flow, ultra-slow-motion, machine learning, and others are just some of the algorithms that can be used with the offered plug-and-play setup. To gradually teach the idea of event-based machine vision, it provides users with C++ and Python APIs, as well as detailed documentation and a wide range of samples arranged by its implementation level. *Figure 3.22* features the Metavision Intelligence Suite for Compute Vision Applications:

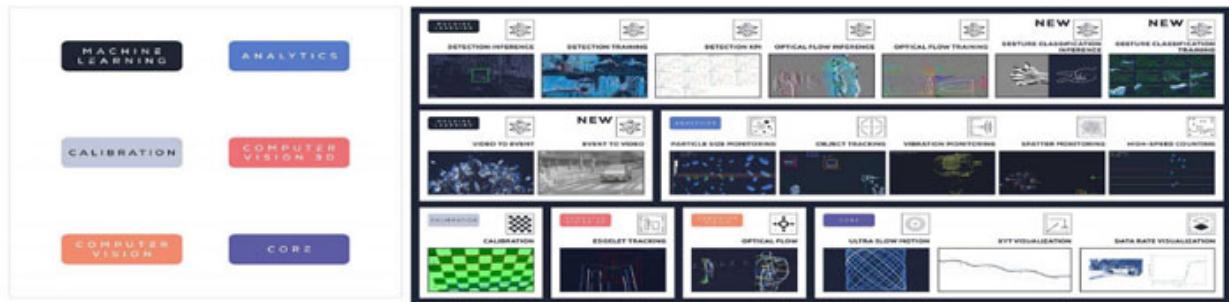


Figure 3.22: Metavision Intelligence Suite for Computer Vision Applications

Data Engineering Frameworks

Let us now discuss the following Data Engineering Frameworks.

Edge Impulse

Microcontrollers and other low-power devices can use the Edge Impulse platform to create and deploy machine learning models. Tools for data collection, model training, and deployment are provided on the platform to help developers for quick building and releasing machine learning models that can function on edge devices with limited compute power and connectivity. With Edge Impulse, programmers can build smart, internet-enabled products that use sensor data to make decisions in real time, even when they do not have access to the internet.

Figure 3.23 features Edge Impulse Suite for TinyML applications:

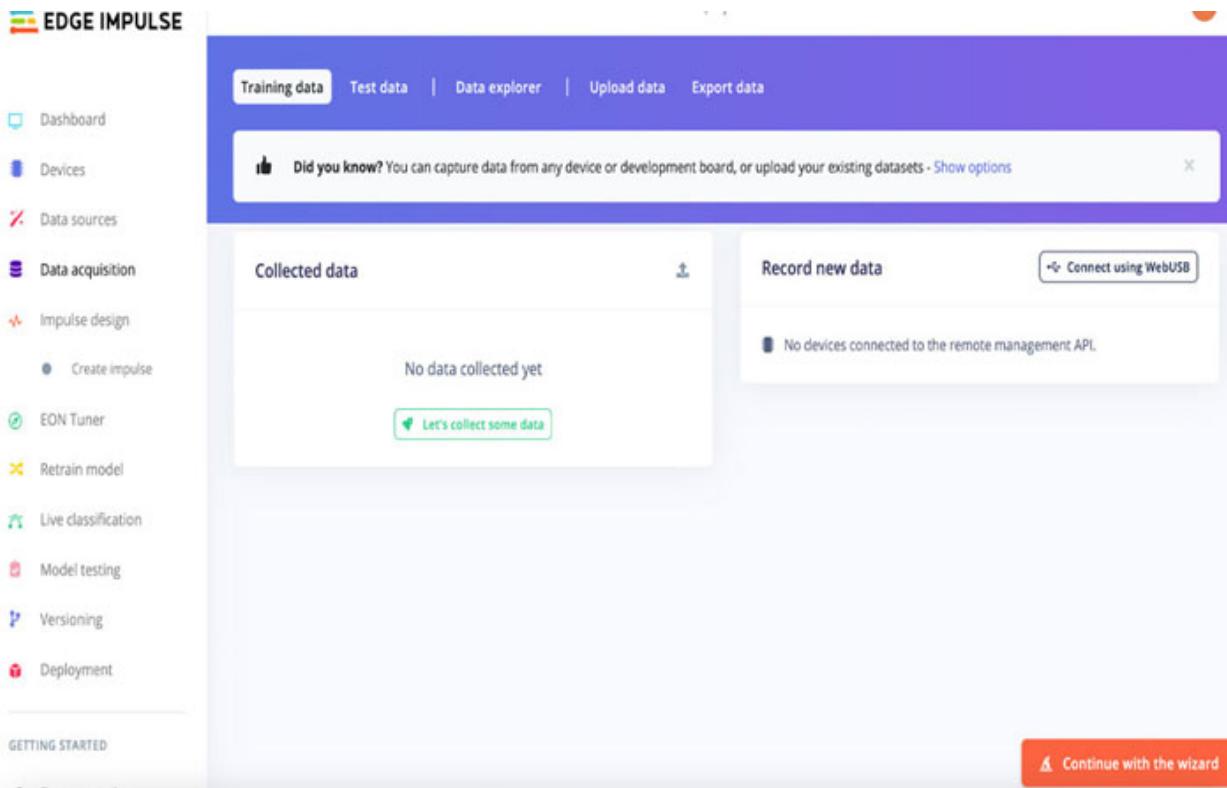


Figure 3.23: Edge Impulse Suite for TinyML Applications

SensiML

With the help of SensiML's tools and services, developers can rapidly and easily build machine learning models for Internet of Things devices. It offers a complete set of tools for data collection, labeling, and training, as well as capabilities for setting up and maintaining machine learning models on edge devices. SensiML is made to make it simple for programmers to incorporate intelligent features into their IoT hardware, such as predictive maintenance, audio and image recognition, and other applications. [Figure 3.24](#) features SensiML example data capturing from sensors:

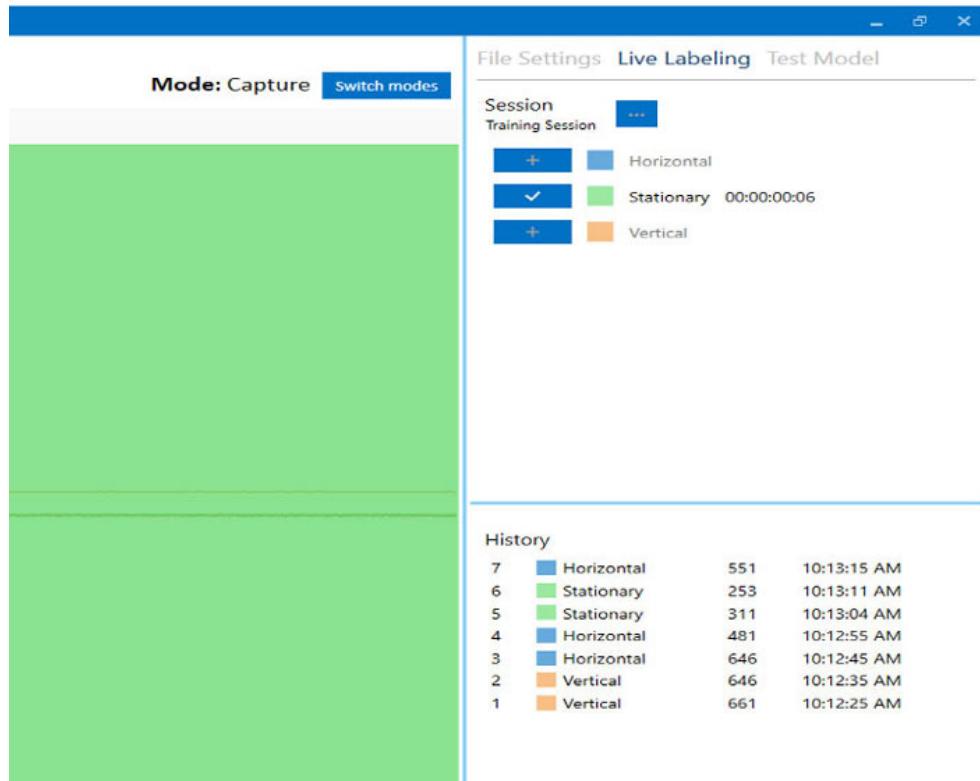
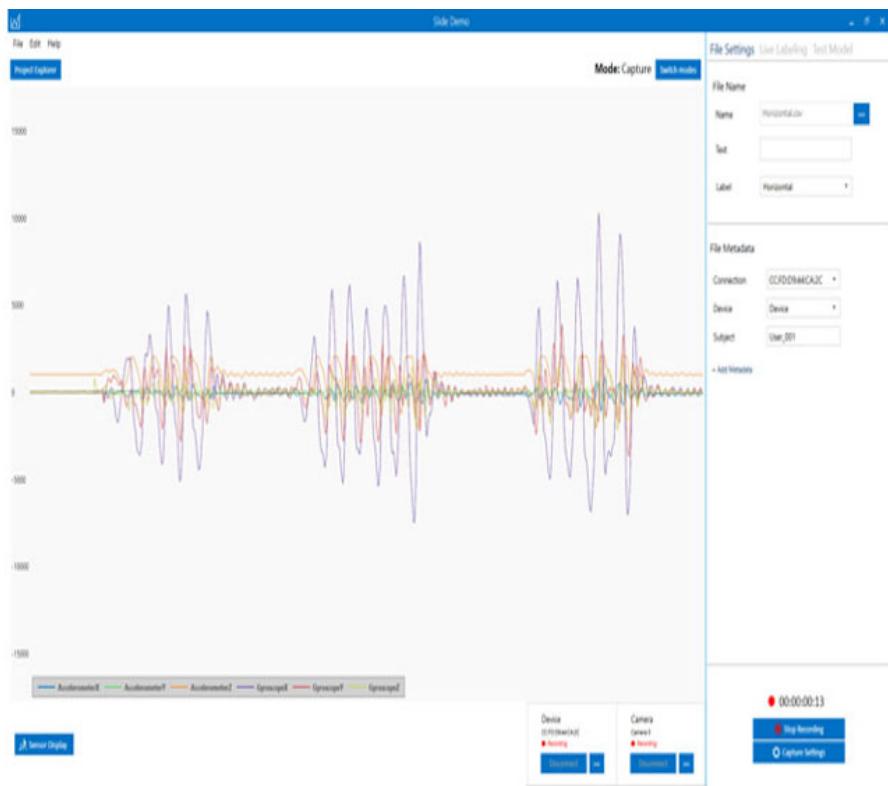


Figure 3.24: SensiML example data capturing from sensors

Qeexo AutoML

Qeexo AutoML is a technology created by Qeexo that allows users to automate the building of machine learning models. It is intended to make it simple for users to create unique machine learning models without the need for machine learning expertise. Qeexo AutoML can be used to create models for a variety of applications such as image classification, natural language processing, and predictive analytics. *Figure 3.25* features a comparison of Qeexo Auto ML's workflow with the traditional ML life cycle:

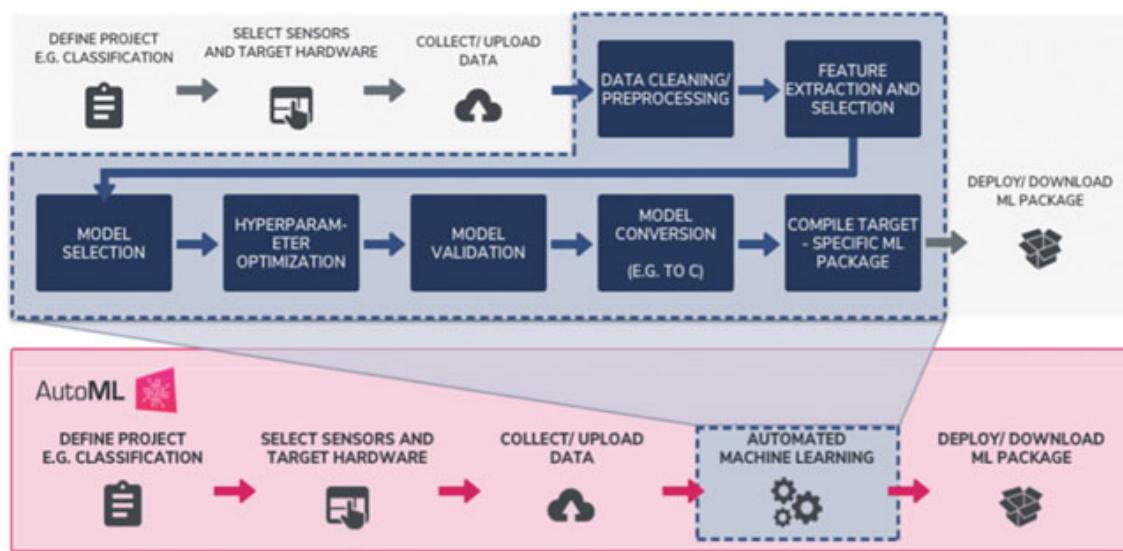


Figure 3.25: Comparison of Qeexo AutoML's Workflow with traditional ML life cycle

TinyML Model Compression Frameworks

Complexity, and thus size, is a major issue in the arms race to build increasingly accurate models. Larger amounts of time and memory are needed to construct these models because of their high complexity and resource requirements. (they need more storage space and it takes longer to predict than smaller models.) A high model size is often the unintended consequence of trying to improve deep learning's predictive abilities beyond what has previously been seen.

There are several reasons why model compression is needed:

- Limitations in memory and storage space, especially on low-powered devices such as smartphones, edge nodes and microcontrollers, are an issue for deploying large machine learning models. By compressing the model, its size can be decreased, making it more suitable for deployment on such devices.
- Large models can also take a long time to produce predictions, which can be an issue in time-sensitive applications such as real-time video or audio processing. Model compression can help to shorten the model's inference time, making it more useful in these types of applications.
- When it comes to the resources needed to train the model and the infrastructure needed for deployment, training and deploying large machine learning models can be quite costly. With model compression, machine learning models can be trained and deployed more quickly and with fewer resources.
- The size of a machine learning model could prevent its deployment on some platforms or in particular settings. In these cases, model compression can aid in making the model more portable and easier to deploy.

The ML model compression techniques are given as follows.

Quantization

Model quantization is a strategy for lowering the size and computational complexity of ML models by utilizing fewer bits to describe the model's weights and activations. It is frequently used to deploy ML models on devices with low memory and computing capabilities, such as smartphones and IoT devices.

There are two main types of model quantization:

- **Weight quantization:** Here, we will reduce the number of bits used to encode the model's weights. Consider switching to 8-bit integers to represent the weights rather than 32-bit floating-point numbers. This has the potential to drastically reduce the model's size, but at the expense of some accuracy.
- **Activation quantization:** To do this, we need to use a more compact representation of the activations (the results of the hidden layers).

Although this technique can lower the overall size of the model in a manner, analogous to weight quantization, it may also result in a drop in precision.

Model quantization can be done either during or after training. We can utilize quantized versions of the model during training and optimize them using the same loss function as the full-precision model. We can quantize a full-precision model after training by finding the closest quantized values for each weight and activation.

Using quantized models has various advantages, including faster inferences, fewer memory requirements, and lower power usage. However, when applying quantization, it is critical to carefully examine the trade-off between model size and accuracy, as too much quantization can dramatically reduce model performance.

Pruning

Model pruning is a method for shrinking a machine learning model by getting rid of pointless or superfluous parameters. Pruning aims to decrease the model's computational and storage requirements while also enhancing the model's generalizability.

Model pruning can be done in a variety of ways, such as weight pruning, unit pruning, and structure pruning. Setting the weights of some model parameters to zero, effectively removes them from the model. This is known as weight pruning. Pruning the model by whole units or neurons, is known as unit pruning. Pruning the structure of a model entails deleting entire layers or sub-structures.

Pruning a model can be done at any point, both before and after it has been trained. Deep neural networks, decision trees, and support vector machines are only few of the models that benefit from its application.

Model pruning has various advantages. Pruning can enhance a model's interpretability by clarifying the function of each model parameter, in addition to reducing the model's size and enhancing its generalization ability. In addition to reducing the model's resource needs, pruning can make it easier to deploy.

Low ranked approximation

Low-ranked approximations are often used in machine learning as a way to reduce the complexity and computational cost of working with large matrices. For example, if a matrix is very large and sparse (meaning that most of its elements are zero), a low-ranked approximation can be used to represent the matrix more efficiently, making it possible to perform calculations on it more quickly.

Low-rank factorization uses matrix and tensor decomposition to identify duplicate parameters in deep neural networks. When it is important to reduce the model size, a low-rank factorization technique can help by decomposing a large matrix into smaller matrices.

Low-ranked approximations can also be used to identify patterns or structures in data. By reducing the dimensionality of the data, it can be easier to visualize and understand what is going on, and to identify relationships between variables.

Knowledge distillation

Knowledge distillation is a method for condensing a large, sophisticated machine learning model (referred to as the "teacher" model) into a smaller, simpler model (called the "student" model). The purpose of knowledge distillation is to transmit the teacher model's information, or "dark knowledge," to the student model so that it can perform as well as the teacher model, but with fewer parameters and faster inference times. To accomplish knowledge distillation, the student model is trained to mimic the teacher model's predictions on a labeled dataset.

The following tools help in achieving the model compression for deployment of TinyML models on resource constrained devices.

TensorFlow Lite

TensorFlow Lite deals with the first two model compression methods listed previously and does a great job in abstracting the hard parts of model compression. TensorFlow Lite covers the following techniques.

1. Post-Training Quantization

a. Reduce Float16

- b. Hybrid Quantization
 - c. Integer Quantization
2. During-Training Quantization
 3. Post-Training Pruning
 4. Post-Training Clustering

Figure 3.26 further illustrates these techniques:

Technique	Data requirements	Size reduction	Accuracy	Supported hardware
Post-training float16 quantization	No data	Up to 50%	Insignificant accuracy loss	CPU, GPU
Post-training dynamic range quantization	No data	Up to 75%	Smallest accuracy loss	CPU, GPU (Android)
Post-training integer quantization	Unlabelled representative sample	Up to 75%	Small accuracy loss	CPU, GPU (Android), EdgeTPU, Hexagon DSP
Quantization-aware training	Labelled training data	Up to 75%	Smallest accuracy loss	CPU, GPU (Android), EdgeTPU, Hexagon DSP

Figure 3.26: Different quantization techniques and their performance comparison with hardware

Figure 3.27 features the steps showing the model compression using TensorFlow Lite:

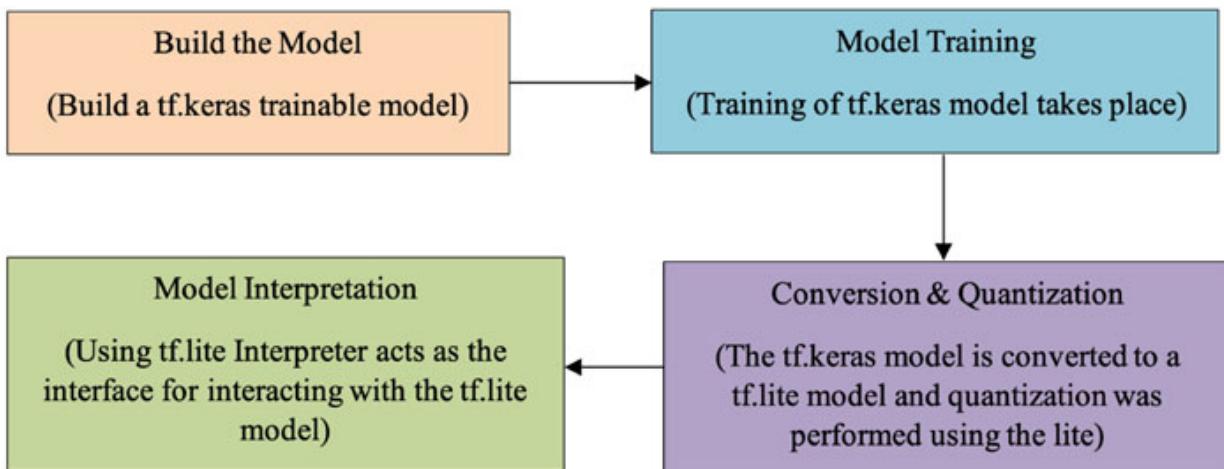


Figure 3.27: Steps showing the model compression using TensorFlow Lite

STM32 X-CUBE-AI

The STM32 X-CUBE-AI is a software expansion pack for STM32 microcontrollers that allows machine learning methods to be used on STM32

devices. It comes with a library of ML algorithms as well as an API for integrating them into STM32 applications.

The extension pack supports several major ML frameworks, including TensorFlow Lite and Caffe, allowing developers to leverage pre-trained ML models or construct and execute their own models on STM32 devices. In addition, the expansion pack provides tools for optimizing ML models for usage on STM32 devices, as well as examples and documentation to assist developers in getting started with ML on STM32 devices.

TensorFlow Lite is slower and uses more memory than X-CUBE-AI. However, it is closed source and only works on STM32 processors, which could stop some people from using it. [Figure 3.28](#) features the X-CUBE-AI performance comparison with TensorFlow Lite:



Figure 3.28: X-CUBE-AI performance comparison with TensorFlow Lite

QKeras

Keras is a Python-based, high-level, open-source deep learning library. It is made to make it easier to build and train neural networks and to give people working with deep learning models, an easy-to-use interface. Keras has many tools for building and training deep learning models, such as functions for creating and compiling models, training and evaluating models, and making predictions based on new data. Keras is often used with TensorFlow and PyTorch, which are also popular deep learning libraries.

QKeras is a quantization extension to Keras that lets us quickly make a deep quantized version of a Keras network. It does this by providing drop-in replacements for some of the Keras layers, especially the ones that create

parameters and activation layers and do arithmetic operations. [Figure 3.29](#) features the QKeras framework for ML model compressions:

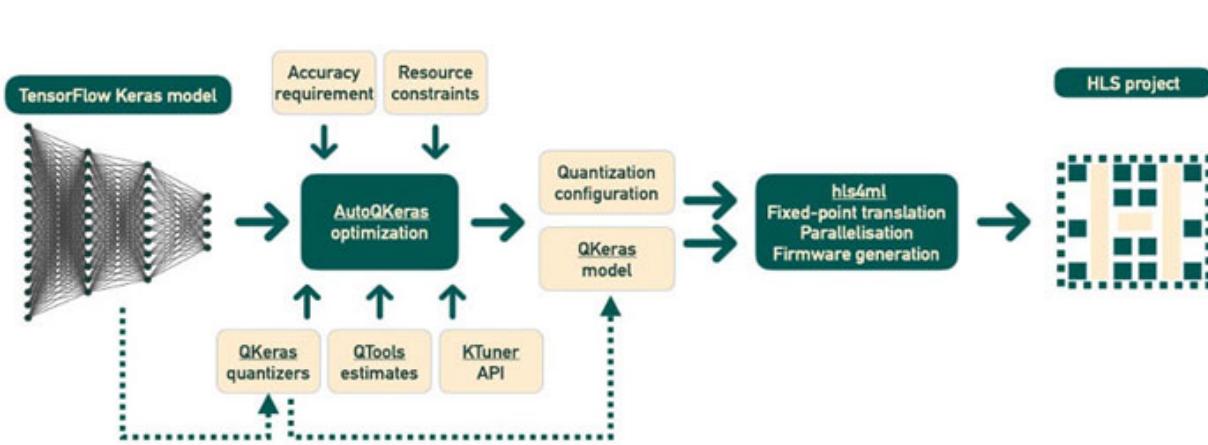


Figure 3.29: QKeras framework for ML model compression

[Qualcomm AIMET](#)

If you are looking to deploy machine learning models on Qualcomm platforms, go no further than the Qualcomm AIMET (**A**rtificial **I**ntelligence and **M**achine **L**earning for **E**dge **T**ensor **C**omputing) library. For trained models to function efficiently on Qualcomm's edge computing platforms, it uses techniques such as quantization and pruning to decrease their size and computational complexity. AIMET is intended to function in tandem with well-known deep learning frameworks such as TensorFlow and PyTorch. It is included in Qualcomm's Neural Processing Software Development Kit.

[Figure 3.30](#) features the Qualcomm AIMET framework for ML model compression:

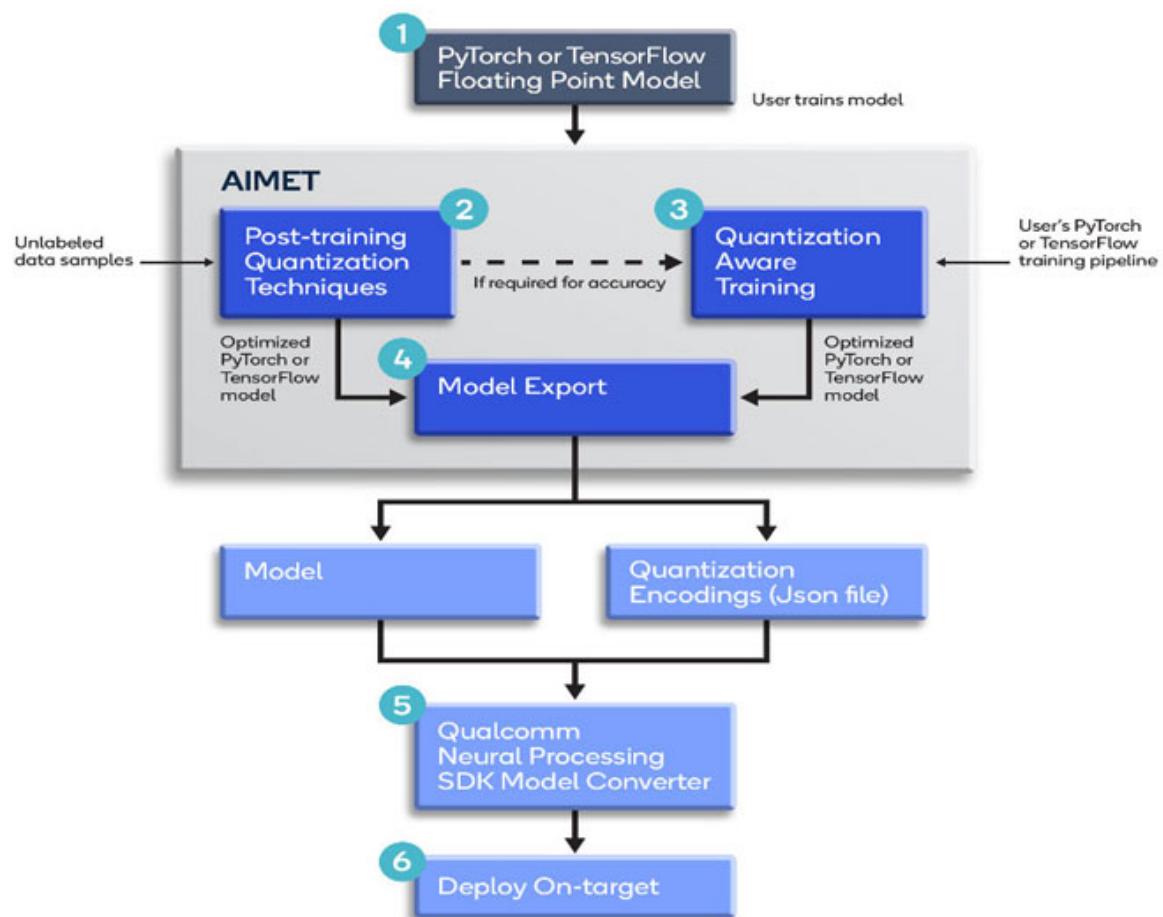
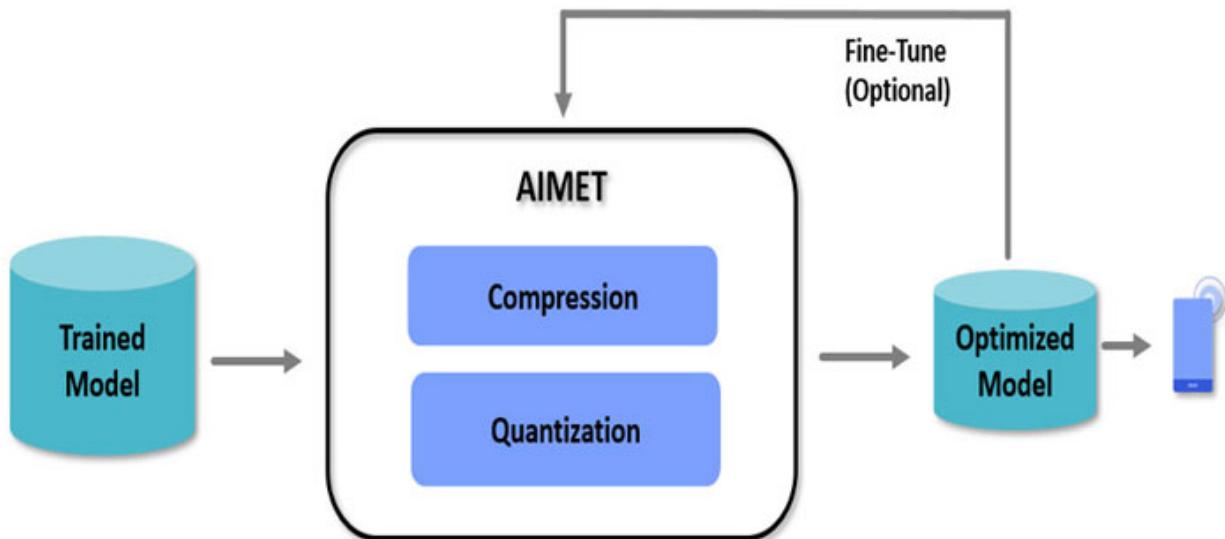


Figure 3.30: Qualcomm AIMET framework for ML model compression

Microsoft NNI

To facilitate the execution of **Automated Machine Learning (AutoML)** experiments, Microsoft has created a machine learning framework known as **Microsoft Neural Network Intelligence (Microsoft NNI)**. The goal of the system is to make large-scale deep learning dataset training faster and more effective. The Microsoft NNI is a suite of tools and libraries that facilitates the development and deployment of neural networks, with features such as distributed training across several workstations. It was developed on top of the free and open-source PyTorch machine learning framework and plays well with other Microsoft AI offerings including Azure Machine Learning and the Microsoft Cognitive Toolkit. Microsoft's NNI is a free and open source AutoML toolkit. Feature engineering, model compression, search for neural architectures, and hyper-parameter tuning are only some of the tasks that may be automated with its help. Compression and operationalization of models are also provided by the toolbox. [Figure 3.31](#) features a high-level architecture diagram of NNI:

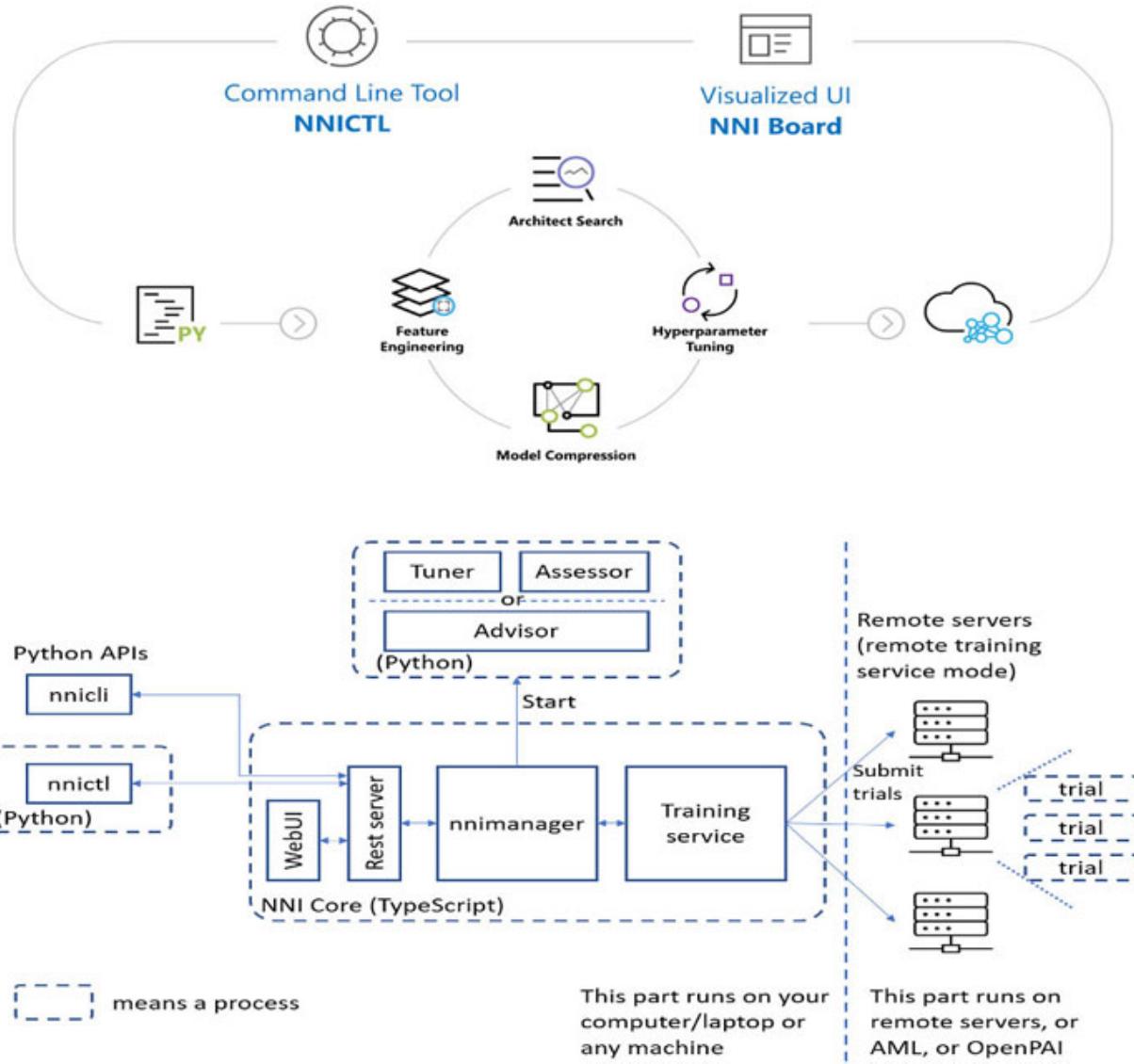


Figure 3.31: A high-level architecture diagram of NNI

CMix-NN

CMix-NN is a free and open-source mixed-precision toolkit for using quantized neural networks on microcontroller-based platforms. The library allows convolutional kernels to have any bit precision* in the range of 8, 4, and 2 bits, in contrast to the state-of-the-art deployment options available for MCUs where only 8-bit precision is allowed.

CMix-NN is a C inference library for ARM Cortex-M MCUs and has the following characteristics:

- CMix-public NN's header file can be found in the directory `Include`.

- The implementations of the convolutional kernels supported by CMix-NN can be found in the directory `source`.
- The code generator for CMix-sources NN's can be found in the directory `scripts/codegen`.

OmniML

OmniML was created by OpenAI and is a machine learning platform. It is made to simplify the process of training and deploying large-scale language models for NLP applications, including text classification, translation, and summarization. OmniML allows users to train and tweak language models using their own data, then apply the trained models to a variety of NLP tasks. The platform's evaluation and analysis capabilities make it simpler for users to comprehend their models' performance and enhance them accordingly. Furthermore, it provides a variety of features and tools to assist users in optimizing their models' efficiency.

[Figure 3.32](#) features a high-level architecture diagram of OmniML platform:

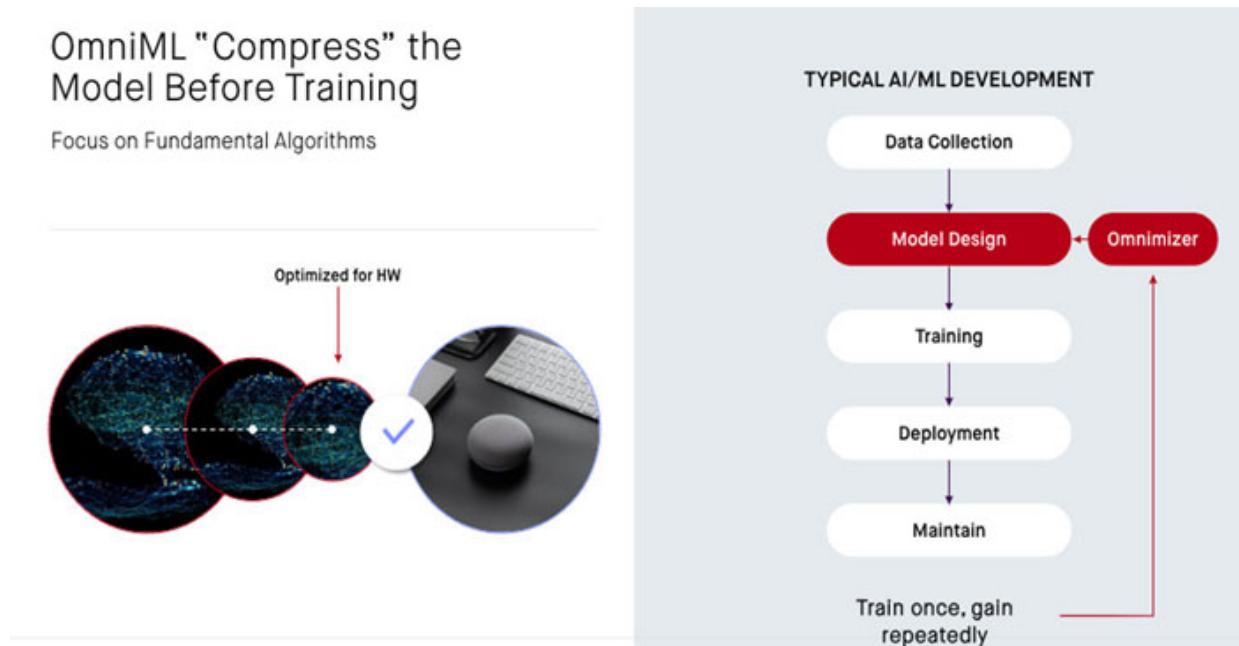


Figure 3.32: A high-level architecture diagram of OmniML platform

Conclusion

In this chapter, we covered major and important TinyML hardware and software platforms. We understood differences among CPUs, GPU, TPU, Raspberry Pi boards and servers at data Centers. We also learned the different types of Raspberry Pi boards, Microcontrollers and their classification based on the architecture, memory size, clock speed, and input/output capabilities and so on. AI accelerators and their advantages for MC boards are studied.

In the TinyML hardware boards, we explored different categories from multiple vendors and their main important features. In TinyML software tools, we went through the platforms to implement the TinyML algorithms on the hardware boards. We have also discussed why ML model compression is needed and the benefits of model compression. In the data engineering and model compression techniques required for TinyML models, current state of the art tools used are covered with their architectures shown at the high level and advantages of model compression techniques are discussed in detail.

Key facts

- A **Central Processing Unit (CPU)** is the primary component of a computer that performs most of the processing tasks. The CPU is responsible for executing instructions from the operating system and applications.
- A **Graphics Processing Unit (GPU)** is a specialized type of processor that is designed to handle the complex calculations required for rendering graphics and video.
- A **Tensor Processing Unit (TPU)** is a specialized type of processor designed specifically for machine learning tasks, such as training and running neural networks.
- TPUs are often used in conjunction with CPUs and GPUs to perform machine learning tasks at scale.
- Servers at data centers are used to store, process, and manage large amounts of data, including data generated by websites, databases, and enterprise applications.
- The Raspberry Pi board contains the CPU, RAM, and other components such as I/O ports, whereas the CPU is simply the central

processing unit that performs computations and executes instructions for the operating system and programs.

- The fundamental distinction between the Raspberry Pi board and the CPU is that the Raspberry Pi board is an entire computer system, whereas the CPU is only one component of the board.
- There are several models of Raspberry Pi boards available, and they can vary in terms of their hardware specifications and capabilities.
- Different types of Raspberry Pi boards are Raspberry Pi 4 Model B, Raspberry Pi 3 Model B+, Raspberry Pi 3 Model A+ and Raspberry Pi Zero W.
- TinyML algorithms can be implemented with different types of hardware and software platforms.
- Major TinyML hardware boards are Arduino (Nano 33 BLE Sense, Nicla Sense ME), Raspberry Pi, Adafruit Circuit Playground Express, Google Edge TPU, NVIDIA Jetson Nano and Syntiant TinyML board and so on.
- Important TinyML software platforms are TensorFlow Lite Micro, uTensor, EloquentML, EON Compiler, STM32Cube.AI and NanoEdge AI Studio, PYNQ (Python productivity for Zynq), SensiML and Edge Impulse and so on.
- Data engineering and model compression frameworks are useful in deploying TinyML algorithms on MCs.

Questions

1. State the major difference between CPU, GPU, TPU and servers at Data Center.
2. List different types of Raspberry Pi boards.
3. In how many ways can we classify the Microcontrollers and list some of them?
4. What are the benefits of TinyML hardware boards for implementing different applications?
5. List TinyML software frameworks used for model development and deployment on microcontrollers.

6. What is the software platform used for deploying ML models on FPGA?
7. Why we need model compression? List tools used for model compression.
8. What are the different types of model compression techniques?
9. List some of the key features of the Syntiant TinyML board and what are the applications we can implement with this board.
10. What are the types of model quantization?
11. What is knowledge distillation technique and how it is useful in model compression?

References

1. **Imagimob AI - Development Platform for Machine Learning on Edge Devices** <https://www.imagimob.com/>
2. <https://github.com/TannerGilbert/Google-Coral-Edge-TPU>
3. <https://realpython.com/python-raspberry-pi/>
4. Osman, A., Abid, U., Gemma, L., Perotto, M. and Brunelli, D., 2022. TinyML Platforms Benchmarking. In International Conference on Applications in Electronics Pervading Industry, Environment and Society (pp. 139-148). Springer, Cham.
5. Ray, P.P., 2021. A review on TinyML: State-of-the-art and prospects. Journal of King Saud University-Computer and Information Sciences.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 4

End-to-End TinyML Deployment Phases

Introduction

End-to-end TinyML deployment is the process of building and deploying a TinyML model on an edge device. It starts with the collection of data and preprocessing it, then moves on to training, evaluating, optimizing, and deploying the model. In this process, there are several steps and technologies, such as data preprocessing techniques, machine learning algorithms development, model optimization methods, and the integration of hardware and software.

It is essential to conduct a thorough analysis of the device's limitations and requirements, as well as the work at hand, before attempting to install a TinyML model on an edge device. This may require selecting an appropriate model architecture and optimization techniques, as well as the proper hardware and software tools, and integrating the model into the application or system you intend to use it in, at the end.

The following steps typically characterize a full deployment of a TinyML model:

1. **Data collection and preprocessing:** The data required to train the TinyML model are acquired and processed at this stage. This could include tasks such as feature engineering, normalization, and data cleaning. This could include deleting any redundant or irrelevant data, normalizing or scaling the data, and dealing with missing or incomplete data.
2. **Model training:** The TinyML model is trained using the prepared data in this phase. This may entail employing a variety of machine learning methods and strategies, to improve the model's performance.
3. **Model validation and testing:** After the model has been trained, it is crucial to validate its performance and precision using a distinct dataset. This may involve testing the model on a different test dataset and measuring parameters such as precision, recall, and accuracy. This ensures that the model is not overfit to the training data and can effectively generalize to new data.
4. **Model optimization and compression:** Once you have trained and tested your model, it is time to optimize it for deployment on a resource-

constrained edge device. Model quantization and pruning procedures are used to minimize the size and complexity of the model.

5. **Model deployment:** After model validation and testing, it is ready for deployment. This may entail integrating the model into an existing application or system, or developing a brand-new application for the model.
6. **Model monitoring and maintenance:** After the model has been deployed, it is essential to perform ongoing monitoring of its performance and to implement any necessary updates or adjustments, in order to increase the model's accuracy and reliability. This may require retraining the model on newly collected data or introducing upgrades to the infrastructure used for deployment.

TensorFlow Lite, PyTorch, and Arduino ML are some of the tools and frameworks that are often used for building and deploying TinyML models. Overall, the goal of end-to-end TinyML deployment is to design and deploy machine learning models that can run well on edge devices and microcontrollers, and do things such as real-time analysis and decision-making without needing to stay connected to the cloud.

Structure

In this chapter, we will talk about the following topics, all of which are relevant to the implementation of end-to-end TinyML model deployment on microcontrollers:

- Understanding Embedded ML (EML)
- Introduction to Edge-impulse and Arduino IDE
- Data collection from multiple sensors
- Data Engineering steps for TinyML
- Model Training in TinyML software platforms
- Model Compression
- Model Conversion
- Inferencing/Prediction of results with test data

Objectives

In this chapter, we will focus our learning on understanding the fundamentals of **Embedded Machine Learning (EML)**, their characteristics, and some sample example systems of EML. Next, we will study the building blocks of EML, pros

and cons of EML and steps involved in running an ML model on Microcontrollers. In the next section, we will go through the details of Edge Impulse and Arduino IDE platforms, their pros and cons, as well as the steps involved in the usage of different hardware boards with these platforms. We will learn about data collection from different sensors, and the data collection steps involved with the previously-mentioned platforms. Lastly, we will go through the data engineering steps, model training with Edge Impulse, optimization techniques involved, and inferencing techniques for model deployment on TinyML hardware platforms.

Understanding Embedded ML

Today, embedded computing systems are quickly making their way into every aspect of human life. They are used in things such as wearable health monitoring systems, wireless surveillance systems, networked systems on the **internet of things (IoT)**, smart appliances for home automation, and antilock braking systems in cars.

An embedded system is a piece of software that runs on a microcontroller. Developers create software for specific devices and functions. This means that the systems have specialized memory and processing needs. While the terms "firmware" and "embedded software" are frequently used interchangeably, "firmware" refers to the low-level software that is firmly connected to the hardware's design. A lot of embedded developers start from scratch when they make embedded software. Other people start with an operating system. A "bare metal" solution is one that does not use an operating system. Bare metal is great for low-end microcontrollers, but it is not the best for advanced features such as connectivity.

When ML models are executed on embedded hardware, this type of machine learning is referred to as **Embedded Machine Learning (EML)**. The EML accomplishes its goals by adhering to the following essential precept: in contrast to the training of machine learning models such as neural networks, which takes place on computing clusters or in the cloud, the execution and inference processes of models are carried out on embedded devices. It turns out that deep learning matrix operations can be properly carried out even on hardware with limited CPU capabilities or even on tiny 16-bit or 32-bit microcontrollers, contrary to the widespread perception that this is not possible.

A typical EML system comprises a small processor or microcontroller, a set of sensors and actuators for obtaining data and conducting actions, and a machine learning algorithm that has been trained to execute a particular task. The

algorithm is frequently pre-trained and tailored to interact with the individual sensors and actuators of the system, as well as for low power consumption and rapid execution. EML systems are used in a wide range of applications, including robotics, IoT devices, and smart homes.

The following are some of the most important features of an EML system:

- **Hardware integration:** As their name implies, EML systems are built right into the hardware of a device or system, to enable instantaneous processing of data.
- **Specialized tasks:** For example, an EML system might be built to recognize images, recognize voice, or do predictive maintenance.
- **Limited resources:** Because of constraints in areas such as memory, computing power, and energy consumption, EML systems must be designed to make the most efficient use of available resources.
- **Offline operation:** Many EML systems can work without being connected to the internet or other external resources. This can be important in places where there is none (or very limited) reliable connectivity.

Here are a few examples of EML systems:

- **Smart surveillance cameras:** These cameras make use of ML algorithms to identify and categorize objects that are within their range of vision. As a result, security personnel are made aware of any potential dangers.
- **Predictive maintenance for industrial equipment:** Algorithms that learn from machine data can examine sensor data from industrial equipment, in order to determine when it is necessary to do maintenance. This therefore assists businesses in reducing downtime and improving productivity.
- **Agricultural sensor networks and drones:** Field sensors can measure soil moisture and temperature, and machine learning algorithms can optimize irrigation and fertilization schedules to increase crop yield. Agricultural drones utilize machine learning algorithms to examine sensor photos and make real-time choices on crop growth and pest detection.
- **Smart home devices:** Numerous IoT devices, such as smart thermostats and security systems use ML algorithms to automatically adapt their behavior, based on the habits and preferences of their users.
- **Autonomous vehicles:** Autonomous vehicles employ ML algorithms to drive, avoid obstacles, and make judgments. ML algorithms analyze sensor data from cameras, LiDAR and radar, to help autonomous vehicles navigate and avoid hazards. Modern cars have sensors and other electronics to collect

data on performance, road conditions, and other aspects. Real-time brake and steering adjustments can also improve vehicle performance and safety using this data. A car's brake system may utilize ML to forecast when the brakes will be needed, based on speed, road conditions, and other factors and adjust brake pressure to improve braking performance.

- **Healthcare devices:** Smartwatches and fitness trackers utilize ML algorithms to assess health data and make customised recommendations. ML can assess data from wearable monitors, insulin pumps, and other medical equipment and make real-time adjustments to optimize treatment. For instance, an insulin pump may employ ML to forecast the patient's insulin needs, based on activity, blood sugar, and other parameters and alter insulin delivery to maintain healthy blood sugar levels.

Several basic blocks are frequently utilized in EML systems. Here are a couple of such examples:

- **Sensors and Hardware:** These devices collect environmental data and transform it into a format that can be processed by the ML system. For instance, an accelerometer and a temperature sensor may monitor acceleration and temperature, respectively. EML systems often require specific hardware for optimal operation. This can comprise of microcontrollers, **Digital Signal Processors (DSPs)**, **Field-Programmable Gate Arrays (FPGA)**, and **Application-Specific Integrated Circuits (ASICs)**.
- **Operating system:** EML systems usually need a lightweight **Operating System (OS)** that works well in environments with limited resources. OSs such as FreeRTOS, VxWorks, and μC/OS are all examples of this type.
- **Communication Interfaces:** EML systems frequently have the requirement to interface with other devices or computer systems, and this communication might take place over a wired or wireless connection. Protocols such as Ethernet, USB, Bluetooth, and Wi-Fi could fall into this category.
- **ML Model Development and Deployment:** Important steps involved are Data preprocessing, model training and evaluation, model deployment.

Some of the pros of EML are as follows:

- **Low latency:** Instead of using a remote server, EML systems make real-time predictions on the device.

- **Increased efficiency:** ML algorithms let devices make better decisions, improving performance.
- **Improved privacy:** EML on devices protects user privacy by processing data locally.
- **Better resource utilization:** EML systems use fewer resources than general-purpose ML platforms, because they only need to fulfill their intended task.
- **Enhanced user experience:** EML lets devices customize user experiences.

Some of the cons of EML are as follows:

- **Limited capabilities:** EML systems may have limited processing power and memory, limiting their performance.
- **Difficult to update:** Since it involves physical access to the device, updating or modifying an EML system might be difficult.
- **Dependence on specific hardware:** EML systems may be hardware-specific and difficult to transfer.
- **Limited flexibility:** EML systems may not be adaptable to new requirements or use cases.

[Figure 4.1](#) features a Venn diagram of TinyML:

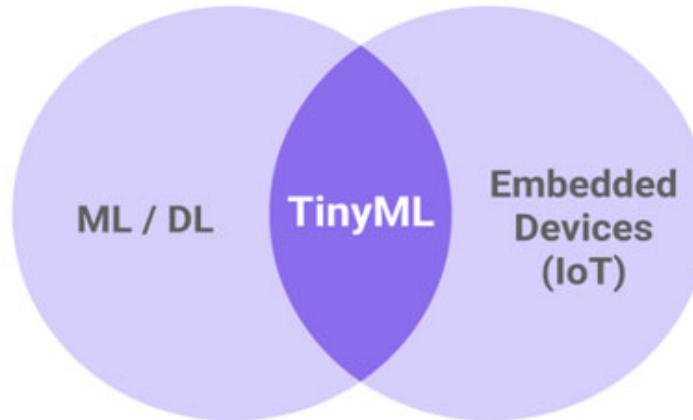


Figure 4.1: TinyML

[Figure 4.2](#) features a Venn diagram of TinyML at scale:

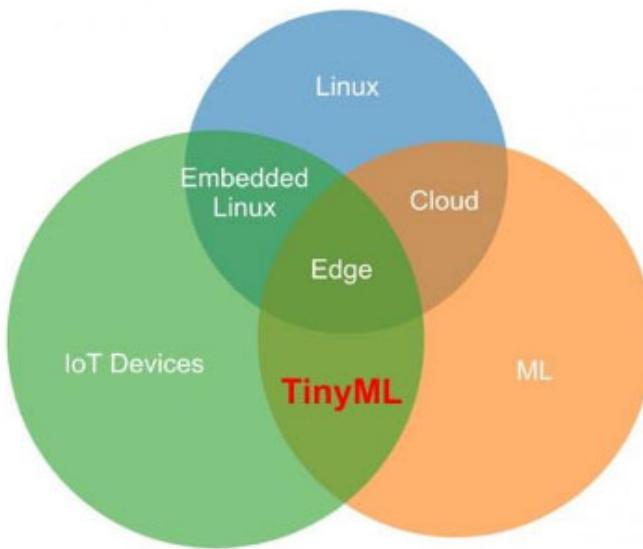


Figure 4.2: TinyML at scale

[Figure 4.3](#) features the advancement of computing technology over time and EML:



Figure 4.3: The advancement of computing technology over time and Embedded Machine Learning

[Figure 4.4](#) features a generic ML life cycle and the steps involved in sorting ML models onto microcontrollers:

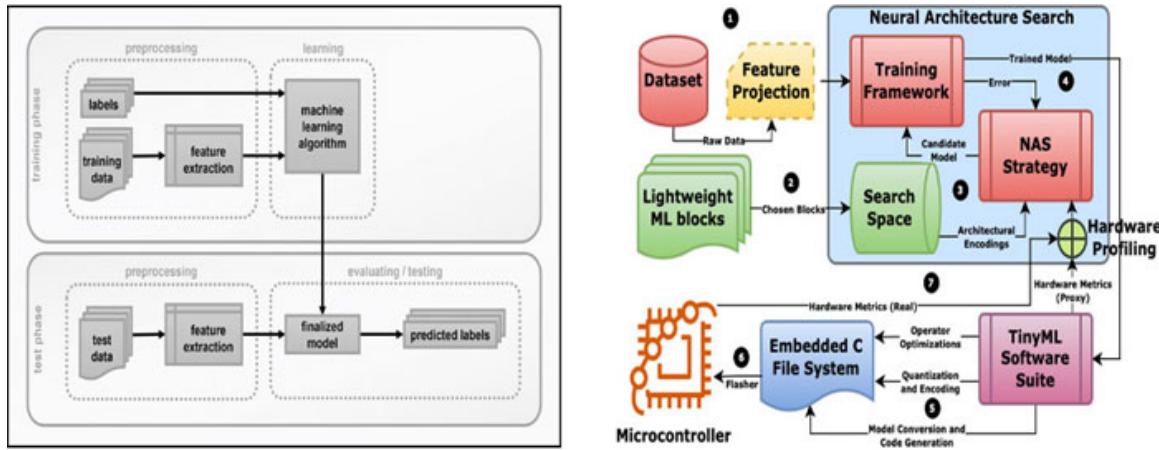


Figure 4.4: A generic ML life cycle and steps involved in porting ML models onto microcontrollers

Introduction to Edge-impulse and Arduino IDE

Let us now learn about Edge-impulse and Arduino IDE respectively.

Edge-impulse

An Edge Impulse is a platform that gives software developers the ability to build and deploy machine learning models on the edge, often known as the edge of networks, by utilizing sensor data from devices that are built on microcontrollers such as Arduino. It offers a user-friendly web-based interface for the construction, training, and deployment of ML models by utilizing sensor data from microcontroller-based devices such as the Arduino.

Edge Impulse offers a variety of development tools that may be utilized to construct machine learning models for edge devices. These models can be employed in many applications. These platforms are comprised of the following:

- **The Edge Impulse Studio:** This is a web-based platform that provides developers with a GUI for the purpose of creating ML models and deploying those models. It contains tools for the collecting of data, the labeling of data, the training and testing of models, and the deployment of models. Refer to [Figure 4.5](#):

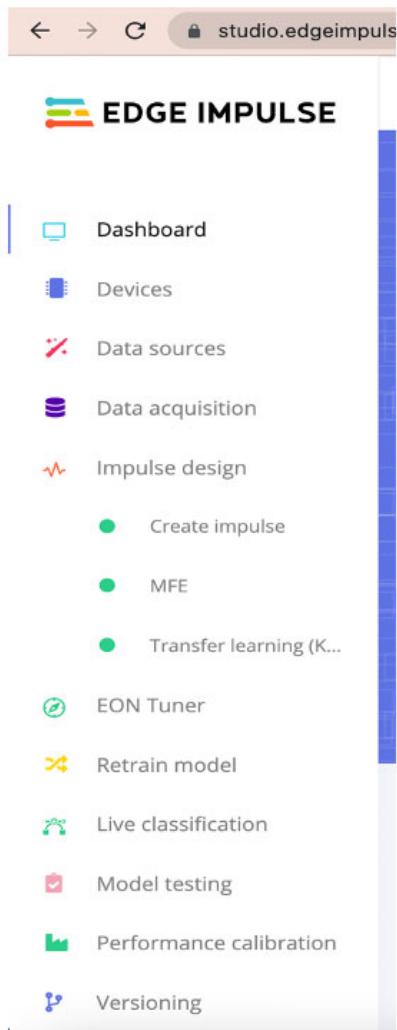
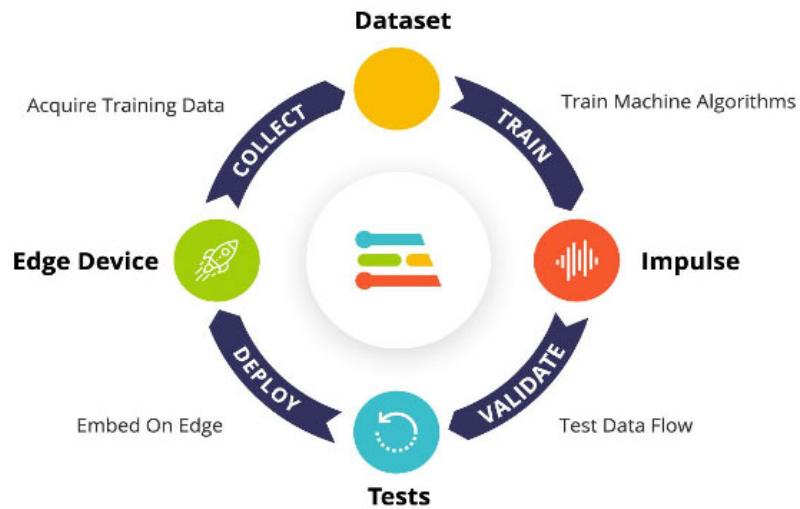


Figure 4.5: ML Development cycle with Edge Impulse and its main modules in the tool

- **The Edge Impulse CLI:** This is a command-line interface that gives developers the ability to construct and deploy machine learning models by utilizing a terminal or command prompt instead of traditional graphical user interfaces. Refer to the following [Figure 4.6](#):

```

C:\ Command Prompt - edge-impulse-run-impulse
Microsoft Windows [Version 10.0.18362.1016]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Rafae>edge-impulse-run-impulse
Edge Impulse impulse runner v1.8.1
[SER] Connecting to COM3
[SER] Serial is connected, trying to read config...
[SER] Retrieved configuration
[SER] Device is running AT command version 1.3.0
[SER] Started inferencing, press CTRL+C to stop...
LSE
Inferencing settings:
    Interval: 0.06 ms.
    Frame size: 12800
    Sample length: 800 ms.
    No. of classes: 2
Starting inferencing, press 'b' to break
Starting inferencing in 2 seconds...
Recording...
Recording done

```

Figure 4.6: Edge Impulse command line tool

Overall, Edge Impulse provides a range of development platforms that enable developers to build and deploy ML models for edge devices in a variety of ways, depending on their preferences and needs.

Edge Impulse currently supports a wide number of development boards, for the experiments. The following boards are used in this book:

- **Arduino Nano RP2040 Connect:** The brain of the board is the the Raspberry Pi RP2040 silicon, a dual-core Arm Cortex M0+ running at 133MHz. It has 264KB of SRAM, and the 16MB of flash memory is off-chip to give you extra storage.
- **Syntiant TinyML boards:** TinyML Board (for example, NDP101 TinyML Board) is a tiny development board with a microphone and accelerometer,

USB host microcontroller and an always-on Neural Decision Processor, featuring ultra-low-power consumption, a fully connected neural network architecture, and fully supported by Edge Impulse.

Some potential advantages of using Edge Impulse include:

- Ease of use, customized models.
- Low latency, energy efficiency.
- Scalability, and integration with other tools.

The steps involved in Edge Impulse usage for any hardware board are as follows:

1. Edge Impulse account creation, as shown in the following [Figure 4.7](#):

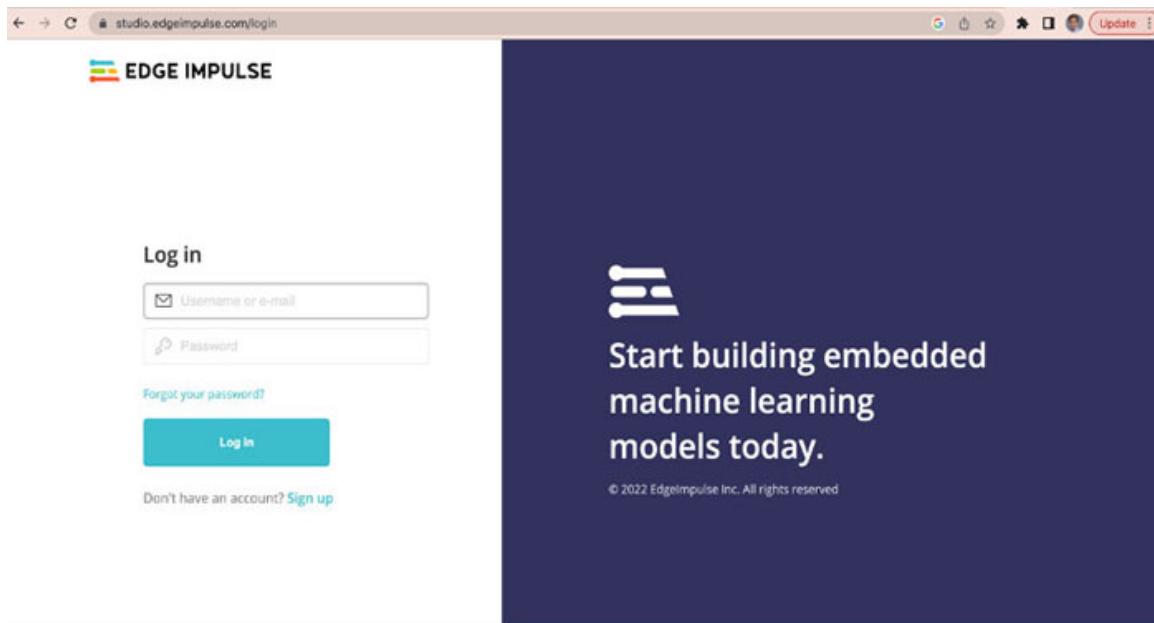


Figure 4.7: Edge Impulse account creation

2. Installing required dependencies for the selected hardware board. Sample dependencies are as follows:
 - a. Installing Node.js and its dependent packages
 - b. GNU Screen
 - c. Command Line Interface
3. Adding the required software components and connecting the hardboard to compute, as shown in the following [Figure 4.8](#):

Your devices

These are devices that are connected to the Edge Impulse remote management API, or have posted data to the ingestion SDK.

NAME	ID	TYPE	SENSORS	RE...	LAST SEEN
 portenta	93:94:AA:E5:88:82	PORTENTA_H7_M7	Built-in microphone, ...	●	Feb 11 2022, 00:03:38
 ti board	A0:B1:C2:D3:E4:F5	TI_LAUNCH_XL	Accelerometer	●	Sep 14 2021, 17:31:59
 tb2	08:68:D7:FE:11:CE	SILABS_TB_SENSE2	Built-in acceleromete...	●	Aug 11 2021, 14:59:22
 test	2E:63:8F:53:C7:4F	NRF52840_DK	Accelerometer, Micro...	●	Aug 10 2021, 20:11:50

Dashboard

Devices

Data acquisition

Impulse design

Create impulse

EON Tuner

Retrain model

Live classification

Model testing

Versioning

Deployment

GETTING STARTED

Documentation

Forums

Your devices

These are devices that are connected to the Edge Impulse remote management API, or have posted data to the ingestion SDK.

NAME	ID	TYPE	SENSORS	RE...	LAST SEEN
 wiopm	30:83:FF:19:13:39	SEEED_WIO_TE..	Built-in accelerom...	●	Today, 12:01:46
 ardupiuno	35:30:35:39:33:30	RASPBERRY_PI_...	Ultrasonic ranger, ...	●	Today, 11:48:59

© 2022 EdgeImpulse Inc. All rights reserved

Figure 4.8: Connecting different hardware boards in Edge Impulse

4. Installing the required firmware and keys for the hardware board, as shown in the following [Figure 4.9](#):

NAME	API KEY	ROLE	CREATED	DEVELOPMENT KEY
mouna-project-1-key	ei_65ecfbf99398e94d15d810775e8db7b2ede944...	Admin	Dec 10 2022, 16:26:09	

NAME	HMAC KEY	CREATED	DEVELOPMENT KEY
mouna-project-1-key	c2ba1c0ff0401b448b23915ce1d4a5982	Dec 10 2022, 16:26:09	

Figure 4.9: Adding keys for hardware boards in Edge Impulse

5. Project name creation.
6. Verify if the hardware board is connected or not.
7. Using sensors to collect the required data (this can be from multiple options), as shown in the following [Figure 4.10](#):

COLLECTED DATA	NAME	CLASS	CREATED
ok people	ok people	Today 10:04:55	1
ok people	ok people	Today 10:04:49	1
ok people	ok people	Today 10:04:49	1
ok people	ok people	Today 10:04:47	1
ok people	ok people	Today 10:04:46	1
ok people	ok people	Today 10:04:46	1
ok people	ok people	Today 10:04:45	1
ok people	ok people	Today 10:04:45	1
ok people	ok people	Today 10:04:44	1
ok people	ok people	Today 10:04:44	1

Figure 4.10: Different data collection options in Edge Impulse and collected audio data samples in Edge Impulse

8. Once data is collected, it is necessary to build, augment, data balance and label the final Dataset.
 9. Designing a model architecture and carrying out the required configurations.
- Figure 4.11** features Model Training (configuring the model parameters):

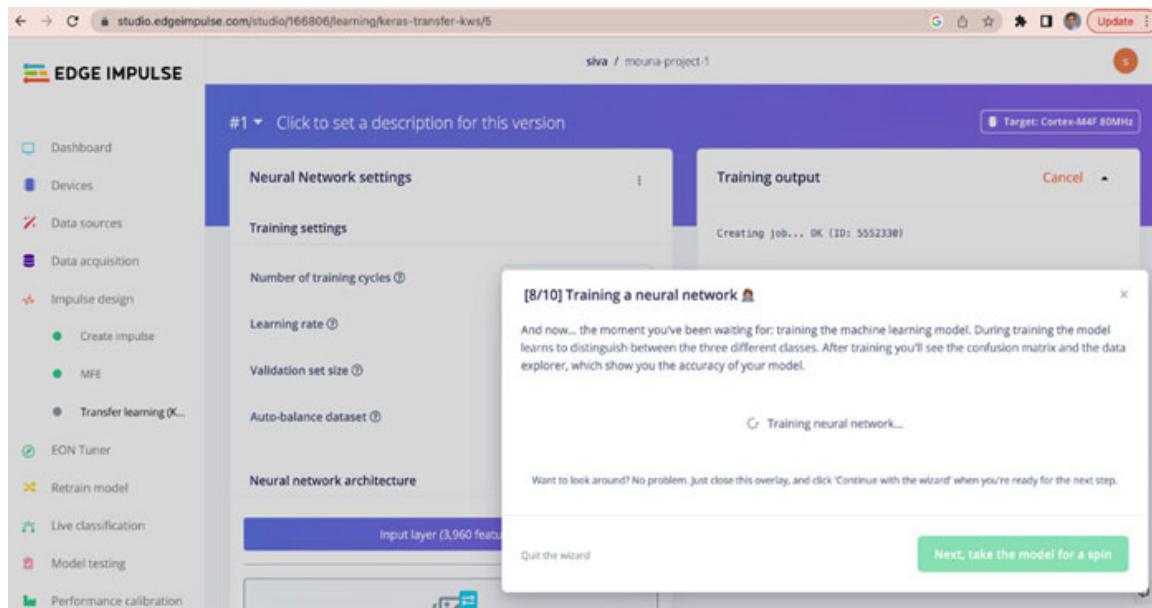


Figure 4.11: ML model training steps in Edge Impulse

10. Model Testing and Validation, as shown in the following **Figure 4.12**:

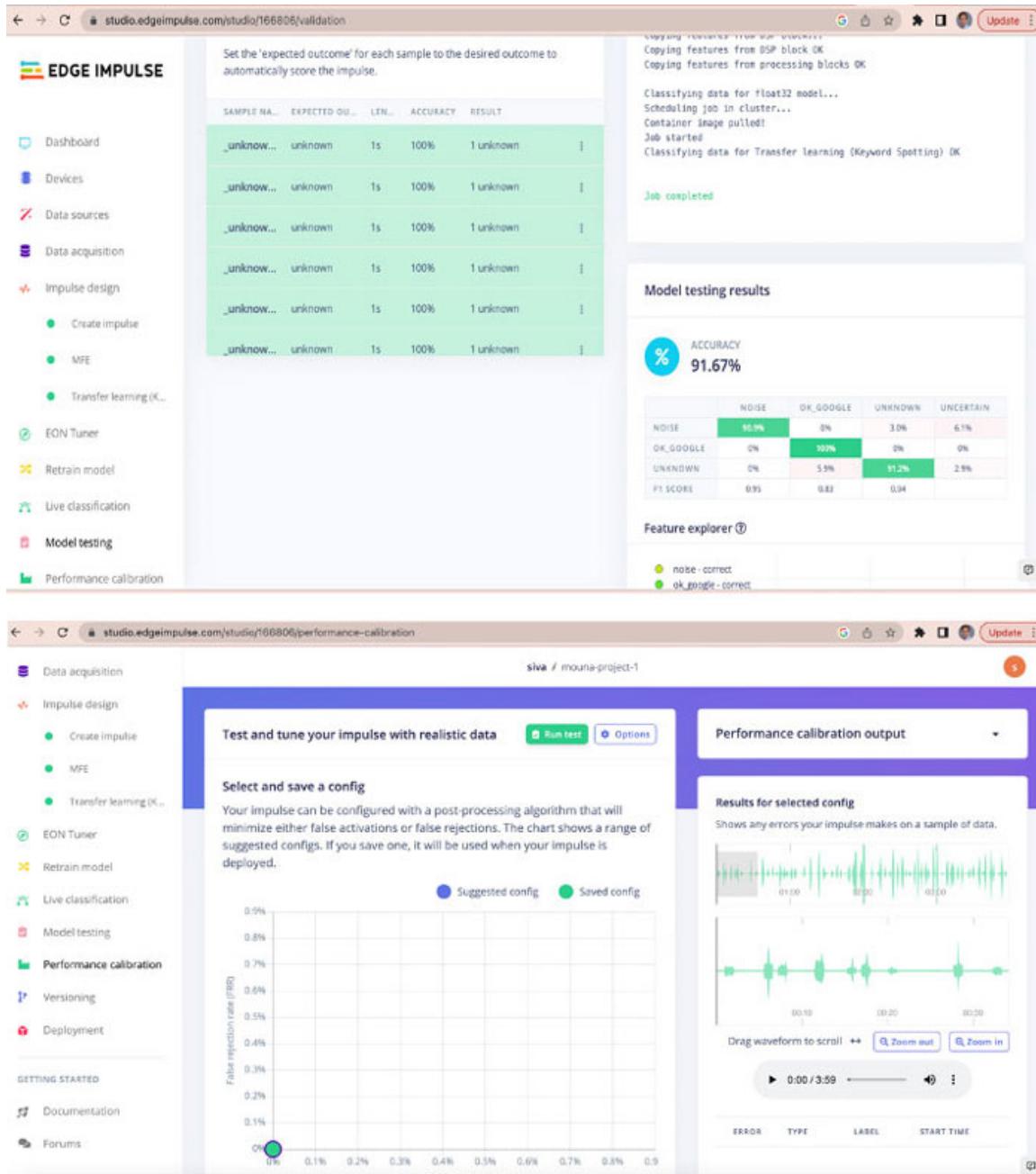


Figure 4.12: ML model testing in Edge Impulse

11. Deployment of the model to hardware board/computer system. This can be done by carrying out the following steps:
 - a. Model conversion and exporting the model as binary file.
 - b. Running inference on the hardware board.
 - i. Flashing the binary file and do the inference on the target hardware board/system.

Refer to the following *Figure 4.13 (a)* to *(e)*:

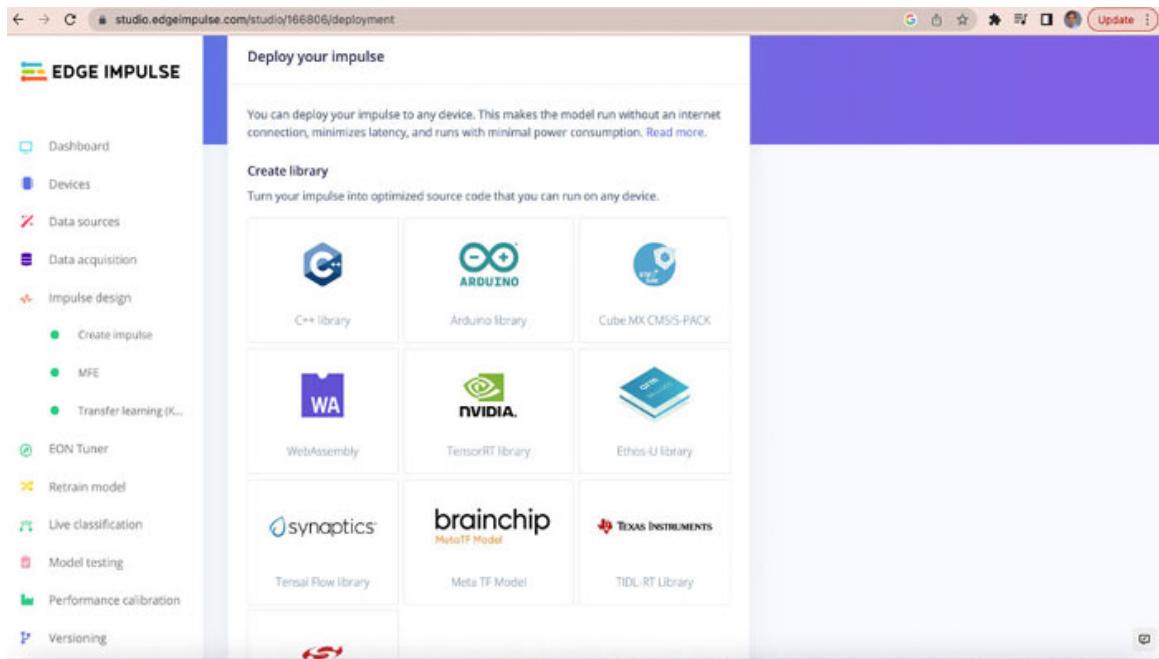


Figure 4.13 (a): Different hardware libraries available in Edge Impulse

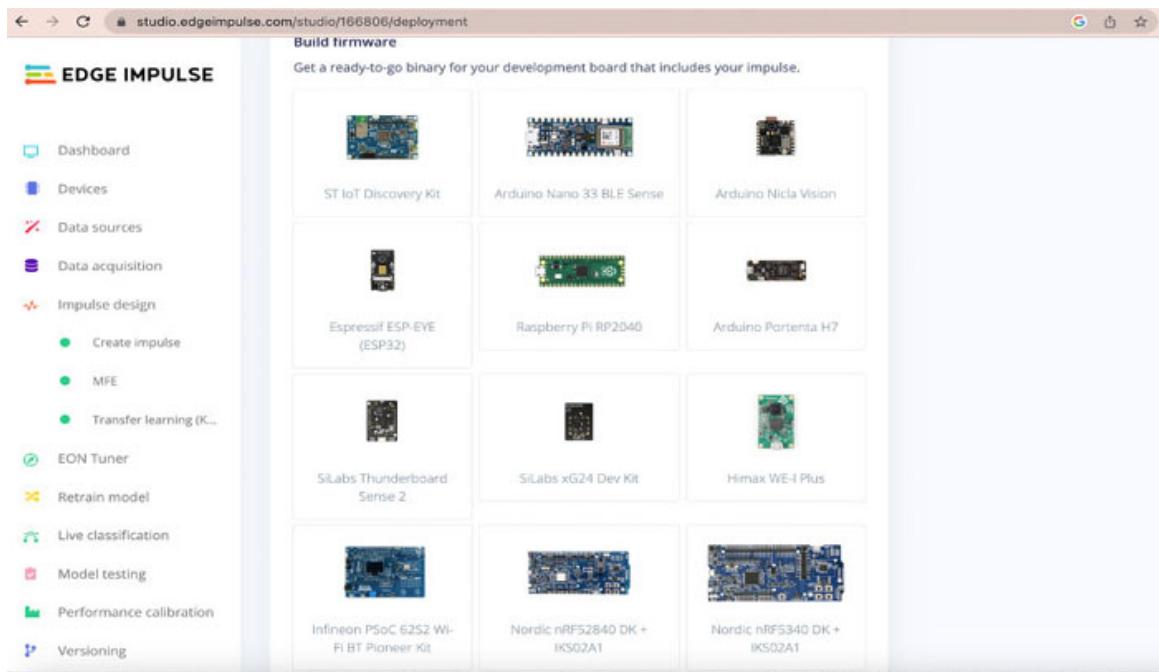


Figure 4.13 (b): Different Hardware Platforms available for the selection

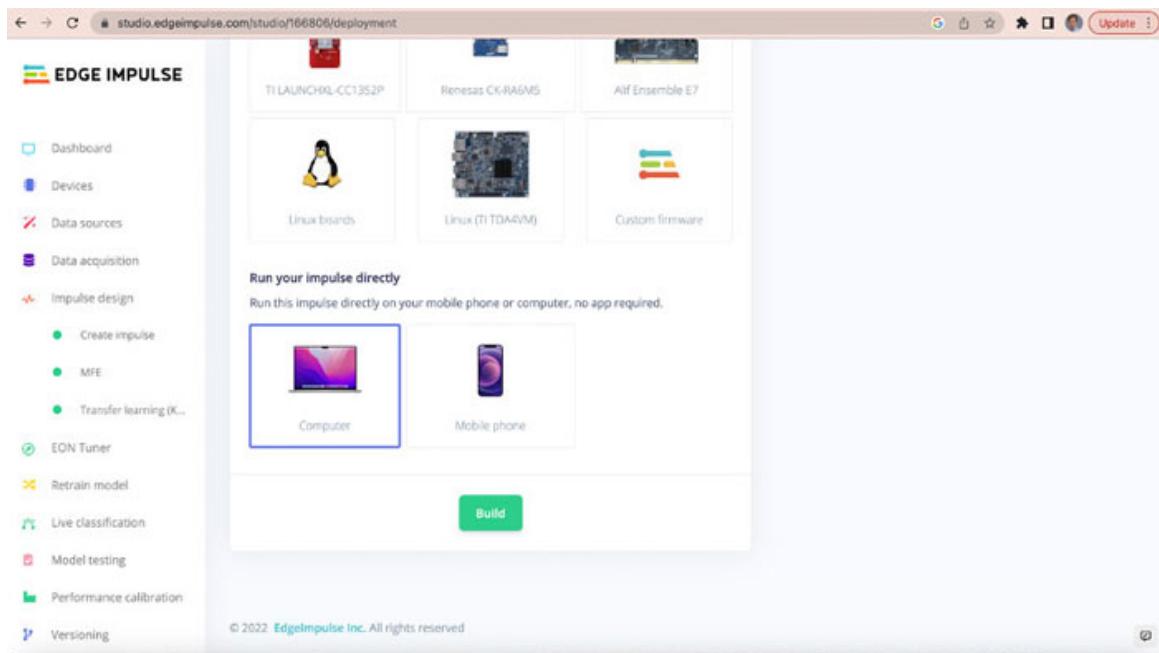


Figure 4.13 (c): Available runtime target deployment device type

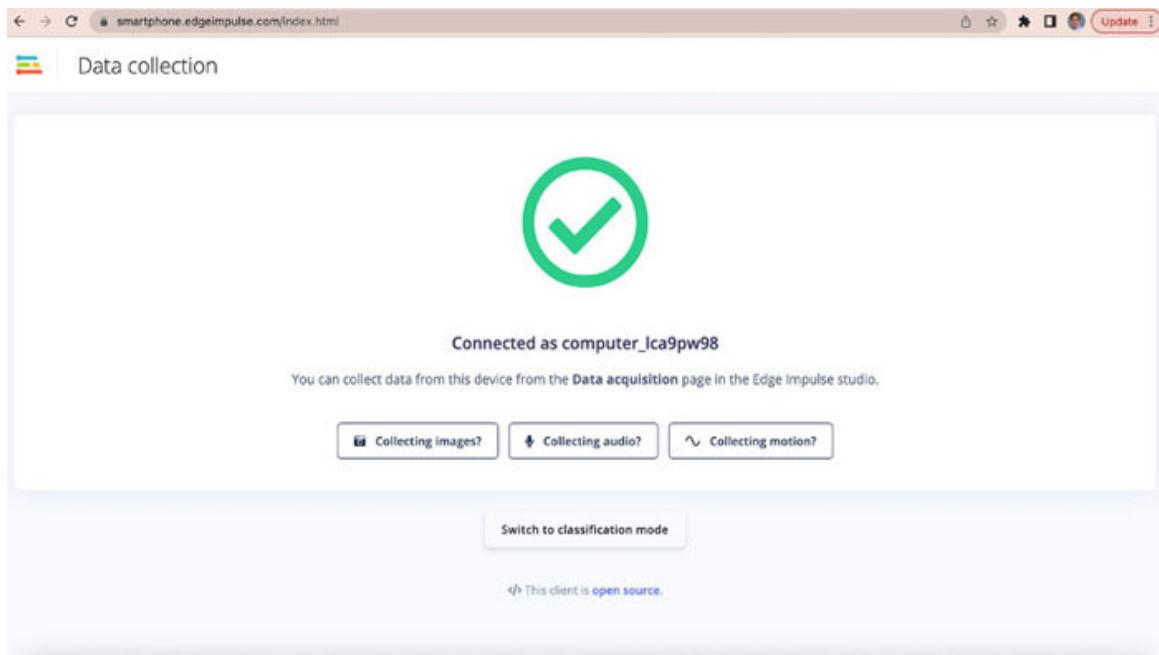


Figure 4.13 (d): Sample selected target device

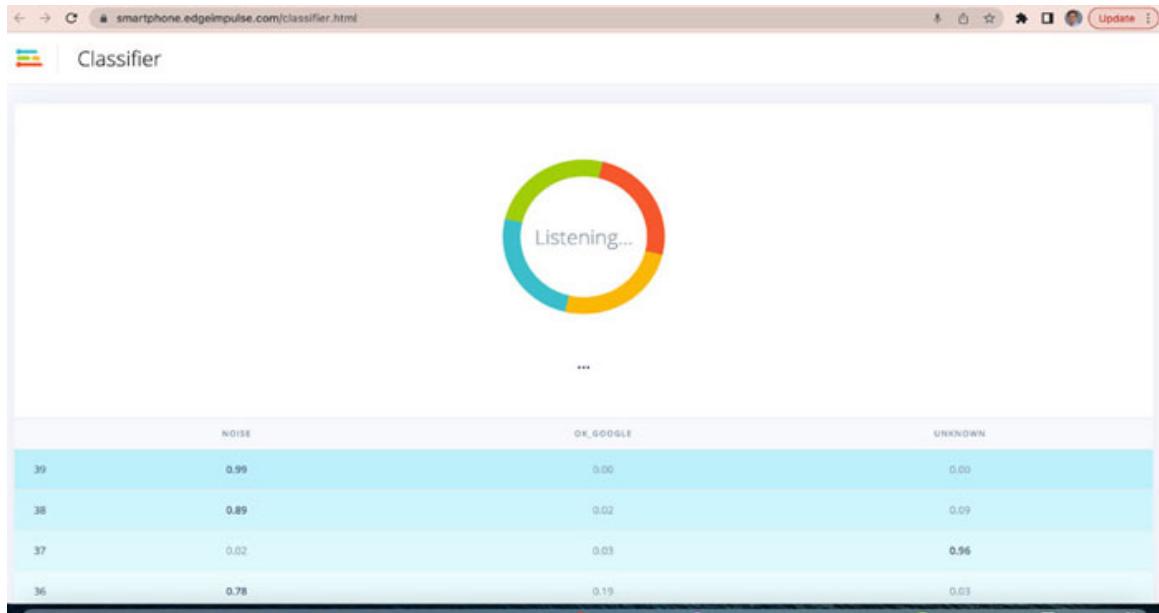


Figure 4.13 (e): Sample Real Time experiments on the target device

Figure 4.13 (a) to (e): ML model deployment and live testing in Edge Impulse

Arduino Integrated Development Environment (IDE)

Code for Arduino boards can be written, uploaded, and debugged with the help of the Arduino **Integrated Development Environment (IDE)**. The Arduino IDE supports the C++ programming language, is cross-platform (Windows, macOS, and Linux), and is also easy to use.

Arduino is a platform for building electronic projects that is made up of both hardware and software that anyone can use for free. The Arduino IDE is a piece of software that lets you write and upload code to an Arduino board in an easy-to-use way. It has a text editor for writing code, a compiler for turning code into a binary file that can be run on the Arduino board, and a debugger for fixing problems in code.

To use the Arduino IDE, you need to have a USB cable connecting from an Arduino board to your computer. Then, you can open the Arduino IDE, use the built-in code editor to write your code, and upload it to the Arduino board. The Arduino board has a microcontroller that runs the code. This lets you control the board and interact with the real world through sensors, motors, and LEDs, among other input and output devices.

There are many libraries and tools in the Arduino IDE that make it easy to write code for the Arduino board. It also has a serial monitor built in, which lets you see what your code does in real time.

Overall, the Arduino IDE is a powerful and flexible tool that lets developers and hobbyists use Arduino boards and other microcontroller-based platforms to build a wide range of projects. The following [Figure 4.14](#) features Arduino hardware as the code editor:



Figure 4.14: Arduino Hardware board and Code editor

It is easy to work and run codes with Arduino IDE. The most up-to-date version of Arduino IDE can be found at the website <https://www.arduino.cc/en/software>. Multiple versions are available for various platforms, including Windows, Mac OS X, and Linux. Additionally, there are now two versions of the Arduino IDE available: Arduino IDE 1.x and Arduino IDE 2.x. This guide will focus on the Classic 1.X release. Both of them essentially provide the same functionality, but have distinct user interfaces and other features such as auto code completion. Follow the given steps:

1. Open the link provided previously in a new browser page, to download the appropriate installer for your computer's operating system.
2. When the download is finished, open the file that ends in .exe
3. To proceed, you will need to click "**Next**" once you have read and accepted the license agreement and decided whether the IDE should be installed for all users.
4. If you wish to alter the default installation location of the IDE, you will need to click the "**Install**" button first, and then select the directory in which you want the IDE to be installed.

After the installation has been completed by the installer, select "**close**" from the menu, as shown in the following [Figure 4.15](#):

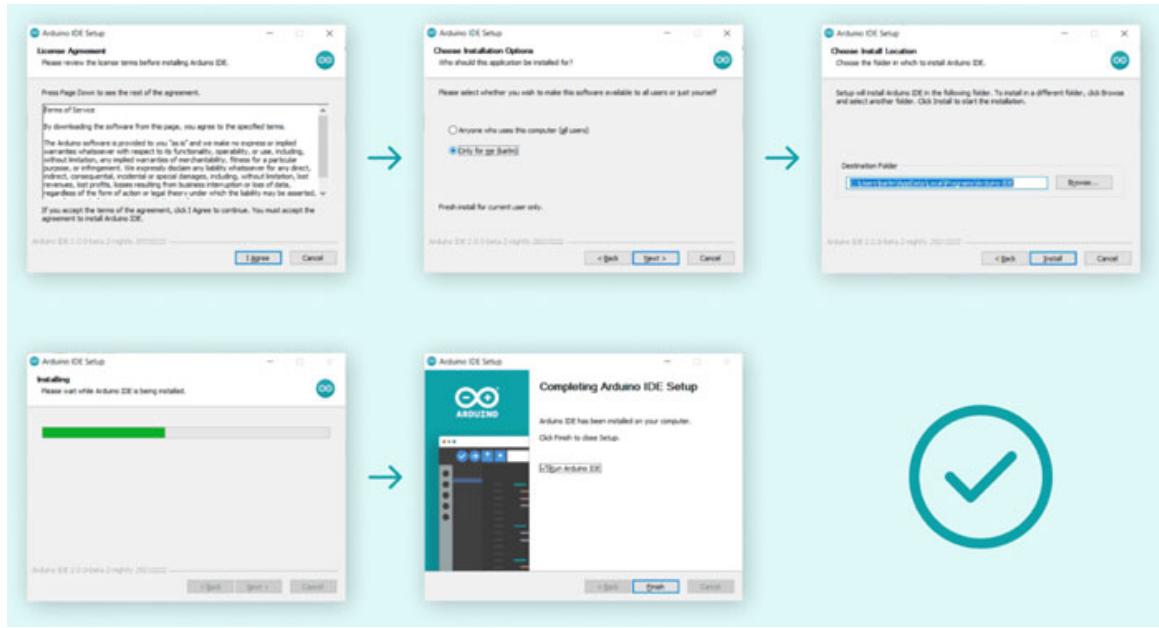


Figure 4.15: Arduino IDE installation steps

Arduino Driver Installation

When getting started with Arduino Software, one of the most typical questions that everyone has, is about the installation of the board driver. But the fact of the matter is that you do not need to be concerned about it. As soon as you connect the board to the computer, the installer will load all the necessary files onto the computer and will install the drivers immediately. For installation, follow the given steps:

- 1. Establishing a Connection Between the Arduino Board and the Computer:** Simply attach the proper cable to the Arduino board, and then attach the other end of the cable to the USB port on your personal computer, as shown in the following [Figure 4.16](#). This will allow you to connect the Arduino board to the computer. As soon as power is applied to the board, the power LED will begin to glow. The system will handle the installation of the driver for the board automatically. Please refer to the following figure:



Figure 4.16: Connecting Arduino board to computer

2. Introduction to the Arduino Software (IDE) and Choosing the Appropriate Board: After the installation of the Arduino IDE, you can access it by either double-clicking the Arduino IDE shortcut on your Desktop or by selecting Arduino IDE from the Start Menu. Right click to launch the Arduino software development environment. [Figure 4.17](#) depicts the environment used by the Arduino IDE:



Figure 4.17: Arduino IDE components

3. Choosing the right board is our next step. The compiler makes heavy use of the board selection in the compile instructions, so getting it right beforehand is crucial. Select your board by going to **Tools | Board | Arduino Nano Every | Arduino megaAVR Boards | Arduino Nano Every**. Refer to the following [Figure 4.18](#):

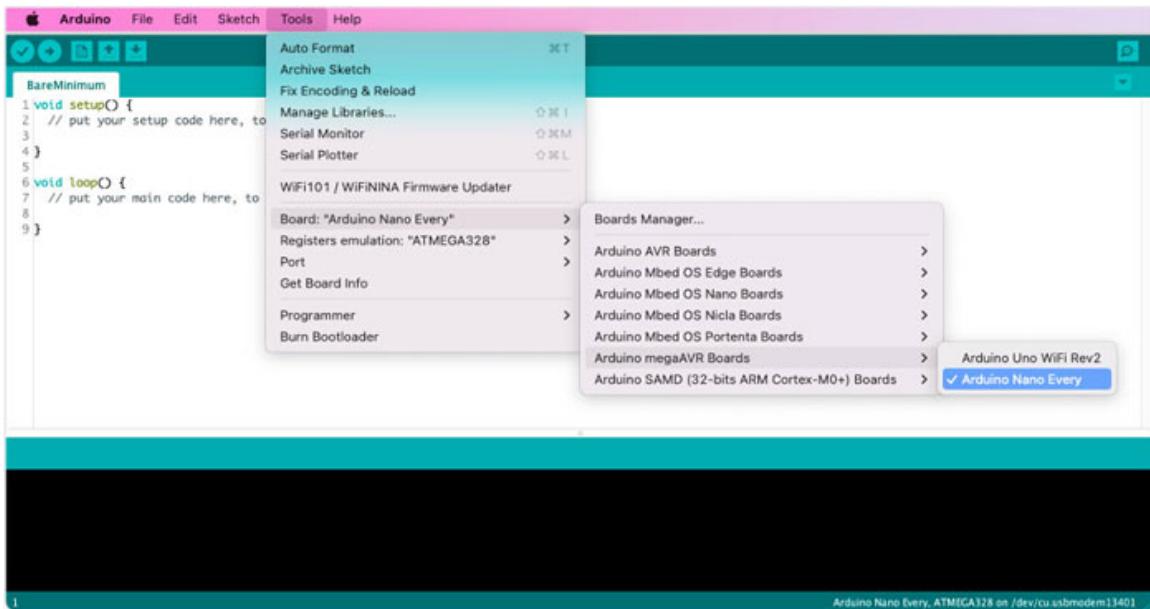


Figure 4.18: Specific hardware board selection in Arduino IDE

4. **Choose the Serial Port on the Arduino:** As with any serial device, the Arduino board's serial port must be set correctly. If you do not do that, you will not be able to program the board. Select the appropriate COM port by going to **Tools** and then **Port**. Disconnect the Arduino board from the USB port and revisit the menu if you are having trouble deciding which COM port to use. The now-missing entry for the COM port will turn out to be the real deal. It is recommended that you use the same USB port that was previously used to connect the Board. Refer to [Figure 4.19](#):

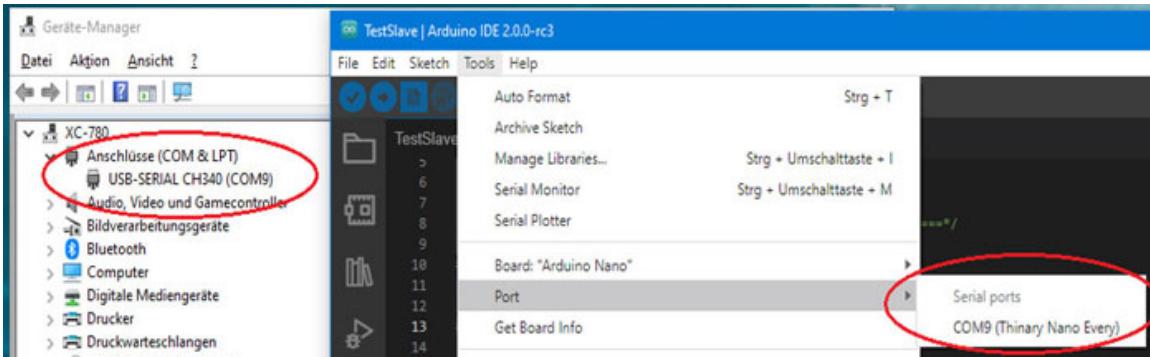


Figure 4.19: Specific serial port selection in Arduino IDE

5. **Codes for Arduino Designed as an Example:** To get started, we will use the LED Blink sample code that comes pre-installed on every Arduino IDE. The Blink example can be accessed through the menus **File** | **Examples** | **Basics** | **Blink**. Refer to the following [Figure 4.20](#):

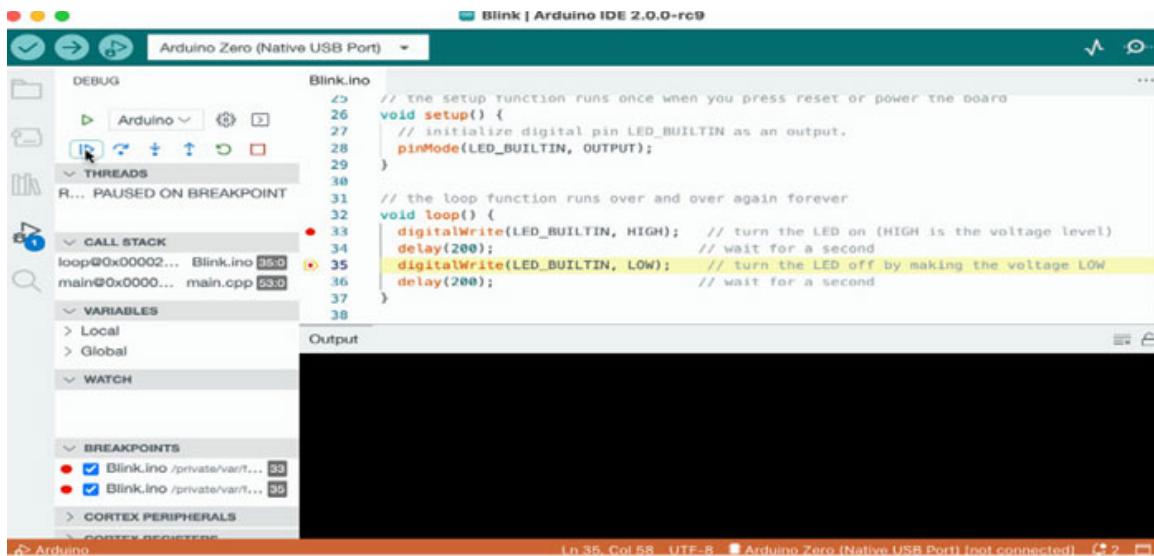


Figure 4.20: Sample code execution in Arduino IDE

6. Arduino Code Uploading: Either click the upload button located in the quick action bar or navigate to the **Sketch** menu and select **Upload** to upload the code. This code is needed to verify if the Arduino has been properly connected, and the appropriate board and port have been chosen. You may also use the shortcut on your keyboard by pressing **Ctrl + U** simultaneously. If the upload was successful, the status bar will indicate as much with a message reading "**Done uploading**" and the LED will begin to blink. Refer to the following [Figure 4.21](#):

```
Done uploading.
Reading | ****| 100% 0.22s

avrduke: verifying ...
avrduke: 924 bytes of flash verified

avrduke done. Thank you.
```

Figure 4.21: Code uploading in Arduino IDE

The advantages of the Arduino IDE tool are as follows:

- It is free and open-source, which makes it easy for anyone who is interested in using it, to get their hands on it.
- It is user-friendly and has an interface that is straightforward to use. Because of this, it is appropriate for beginners who are just starting out in the fields

of programming and electronics.

- It features a sizable and lively community of users and developers who collaborate through online forums and communities to exchange information, resources, and assistance with one another.
- It is compatible with a wide array of Arduino boards and hardware platforms, in addition to a number of programming languages, such as C++ and Python.
- It has a built-in code editor that simplifies and streamlines the programming process by providing features such as syntax highlighting, error checking, and other useful tools and functions.
- It gives you access to a range of libraries and example sketches, as well as a handy method for uploading and debugging code on Arduino boards. This can help you get started with programming Arduinos.
- In order to facilitate debugging and testing, it connects with additional tools and applications such as serial monitors, oscilloscopes, and visualization tools.
- **Extensive documentation:** The Arduino website features extensive documentation for the Arduino IDE as well as the Arduino programming language. As a result, it is simple to learn how to use the tool and comprehend how the code operates.

The disadvantages of the Arduino IDE tool are as follows:

- It is not as feature-rich as other IDEs, such as Eclipse or Visual Studio, which may make it less suitable for more complex projects.
- It does not have as many debugging and profiling tools as other IDEs, which may make it more difficult to troubleshoot issues in your code.
- It is not as well-suited for developing large, modular software projects, as it does not have built-in support for project management and version control.
- It does not have a strong support for programming languages other than C++, although it is possible to use other languages with the Arduino IDE by installing additional software or libraries.
- It may be more difficult to use the Arduino IDE with more advanced microcontrollers or boards that have additional features or peripherals, as the built-in libraries and examples may not fully support these devices.

Overall, Edge Impulse and the Arduino IDE are powerful tools that can be used together to build and deploy machine learning models on microcontroller-based devices.

Data collection from multiple sensors

In a TinyML implementation, data collection from multiple sensors typically involves the following steps:

- 1. Determine the sensors that will be utilized in the data collection process.** The particular requirements of the TinyML application, as well as the kind of information that needs to be gathered, can guide the selection of the appropriate sensors. Accelerometers, gyroscopes, temperature sensors, and pressure sensors are all examples of typical types of sensors that can be used in TinyML applications.
- 2. Adjust the settings on the sensors so that they can gather data at the frequency and resolution you require.** Performing this step may require modifying the settings of the sensor, such as the sample rate or the sensitivity, in order to guarantee that the data being collected is correct and applicable. Establish a connection between the TinyML device or platform and the sensors, so that the data may be processed. To accomplish this, the sensors need to be hardwired into the device, or wireless communication methods such as Bluetooth or Wi-Fi might need to be used instead.
- 3. Start accumulating data from the various sensors.** It is possible that data will be collected continuously or at predetermined intervals, but this will be determined by the requirements of the application. It is essential to make certain that the facts that are being collected are accurate depictions of the environments in which the TinyML device will be utilized. Keep the gathered information in a format that can be quickly accessible and manipulated by the TinyML algorithms, so that it can be used. Performing this step can require storing the data to a local storage device or uploading it to a storage service that is hosted in the cloud.
- 4. Perform an analysis on the gathered data to search for any recurring patterns or long-term trends that may be of use to the TinyML application.** To accomplish this goal, it may be necessary to utilize data visualization tools or statistical analysis methods in order to gain an understanding of the links that exist between the various variables. Conduct performance evaluations of the TinyML algorithms by making use of the data that has been gathered. This will guarantee that the algorithms are able to accurately categorize or anticipate the results that are required.

Data collection from an Arduino board

To collect data from an Arduino board, follow the given steps:

1. Make the connection between the Arduino board and the sensors by using the proper cables and wiring. Be sure to follow the exact instructions for each sensor, since the wiring and connection may be different for each type and model of sensor. This is why it is important to read and follow the instructions carefully.
2. Install the required libraries for the sensors that you are utilizing. It is possible that to interface with some sensors, you will need to first install certain additional libraries.
 - a. Create an initialization sketch for the sensors on your Arduino (program). In most cases, this necessitates the creation of objects for the sensors as well as the specification of the pins to which the sensors are linked.
3. You should initialize the sensors in the setup method of your Arduino sketch by calling the relevant initialization routines for each sensor. Read the data from the sensors, using the functions that are designed specifically for each type of sensor in the loop function of your Arduino program.
 - a. Keep the data from the sensors in variables or arrays, depending on what you require. Make appropriate use of the data collected by the sensors in your sketch. You may, for instance, wish to display the sensor data on an LCD screen, transmit it to a computer via a serial connection, or utilize it in such a way that it controls other devices.
4. It is crucial to note that the particular methods for gathering data from many sensors will depend on the specific sensors you are utilizing, as well as the unique aims of your project. This is something that you should keep in mind.

Data collection from Syntiant board

You will need to follow the given steps to use a Syntiant TinyML board to collect data from several different sensors:

1. Following the instructions in the literature that was provided by Syntiant, connect the sensors to the TinyML board. Each sensor will have a unique way of connection; thus, it is imperative that you pay close attention to the provided instructions.
2. On the TinyML board, the required software and library components should be installed. In most cases, this will necessitate the installation of a

development environment, such as Arduino or Python, as well as any libraries or drivers that are necessary for the sensors.

3. To gather data from the sensors, you will need to write the code. In order to accomplish this, you will need to initialize the sensors and configure them so that they can communicate with the TinyML board. After that, you will have to establish the method of data collection, for which will need you to take into account any filtering or processing that must be done on the sensor data.
4. Conduct tests on the data gathering procedure to validate that it is functioning appropriately. This might require running the code and checking to make sure that the data being collected is correct and reliable. You have the option of either storing the data obtained on the TinyML board itself or transmitting it to another device in order to undergo additional processing or analysis. This may need the setting up of communication protocols such as Bluetooth or Wi-Fi, as well as the writing of code to handle the data transmission process.

Data engineering steps for TinyML

Data engineering refers to the practice of preparing data for analysis and modeling. It includes the following aspects.

Cleaning

Important to TinyML's data engineering is data cleaning, which entails discovering and repairing or deleting data flaws and inconsistencies. This can include identifying and addressing missing values, outliers, and errors in the data, as well as standardizing and formatting the data to ensure that it is consistent and usable. In addition to these methods, it is essential to thoroughly examine the data and comprehend its structure, quality, and constraints. This may involve visually exploring the data, utilizing summary statistics, and testing data-related hypotheses. By adhering to these procedures, data engineers may guarantee that the data is clean, consistent, and ready for use in TinyML applications.

The following are typical data cleansing steps:

- **Identify and correct errors:** Examine the data for problems, such as typos or wrong values, and repair them as needed.
- **Handle missing values:** Determine how to deal with missing data values. This may entail replacing missing values with a default value, removing

rows with missing values, or inputting values based on the mean or median of the remaining data.

- **Normalize and standardize the data:** Normalize and standardize the data so that it has a mean of 0 and a standard deviation of 1. This can assist in enhancing the efficiency of machine learning algorithms.
- **Remove unnecessary columns:** Remove any columns that are unnecessary for analysis or modeling.
- **Check for outliers:** Examine the data for outliers and determine whether to eliminate or retain them. Outliers can have a significant effect on the output of a ML model. Thus, they must be handled with care.
- **Convert data types:** Adjust the data type as needed, to make sure it can be read by the ML programs.

If you follow these guidelines, your data will be in great shape for your TinyML project's analysis and modeling.

It is essential to keep in mind that the process of data engineering consists of a variety of steps, one of which is data cleansing. Other processes include feature engineering (the process of producing new features based on previously collected data), data transformation (the process of altering the data so that it is better suited for modeling), and data integration (combining data from multiple sources).

Organizing

In general, the purpose of data organization within the framework of TinyML, is to structure the data in a manner that makes it possible to conduct analysis and modeling in an efficient and effective manner. It is much simpler to work with and understand data that has been correctly organized, and this can contribute to an improvement in the accuracy and performance of machine learning models.

Transformation

TinyML relies heavily on data transformation to prepare the data for training and assessing machine learning models.

To prepare information for use in a ML model, data must be transformed from its original format or structure. This may necessitate transforming the information from its original format into another one, such as a **Comma-Separated Values (CSV)** file from a raw text file or an image from a numerical array.

For any given ML problem and set of input data, there exists a wide variety of possible data transformations to employ. Instances of frequently used data

transformations are as follows:

- **Normalization:** Normalization means adjusting the data so that its mean is zero and its standard deviation is one. The input data can be made more accessible to machine learning algorithms by means of normalization, which helps to standardize the scale of the input data.
- **Encoding:** To do this, we must first transform categorical information (information that can be placed into a finite number of categories) into a numeric format. One-hot encoding and label encoding are only two examples of the many methods available for representing and transmitting category information.
- **Feature extraction:** This is accomplished by separating out useful information from unprocessed data, such as the edges of an image or the spectral characteristics of an audio wave. Data dimensionality can be reduced through feature extraction, making it more manageable for machine learning algorithms.
- **Imputation:** Estimates or replacements must be made for any information that is either absent or incomplete. As a result of imputation, the machine learning model is less likely to suffer from insufficient data.
- **Data deduplication:** For this purpose, it is necessary to search for and eliminate data repetition. This is vital in order to prevent overfitting and other model problems, by ensuring that the data is genuine and representative.

Model Training in TinyML software platforms

Multiple frameworks and tools (such as TensorFlow Lite and Edge Impulse) can be used to develop TinyML applications.

EON Compiler (Edge Impulse)

EON Compiler is a tool developed by Edge Impulse that allows users to compile and deploy machine learning models for edge devices. One of the main features of EON Compiler is its ability to optimize machine learning models for a specific target device. It does this by analyzing the model and the target device's capabilities, and then generating code that is optimized for that specific device. This allows developers to build TinyML applications that are both accurate and efficient, while also being able to run on devices with limited resources.

The following *Figure 4.22 (a)* to *(c)* features EON compiler and ML model training in Edge Impulse:

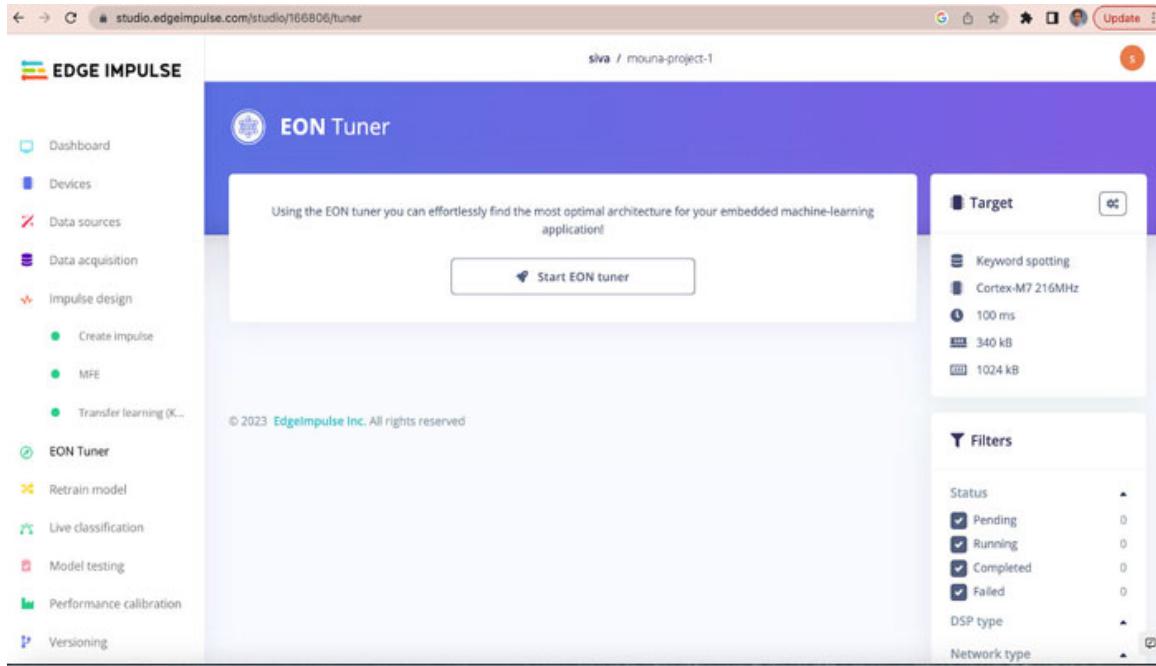


Figure 4.22 (a): Edge Impulse EON tuner for Optimal Architecture Selection

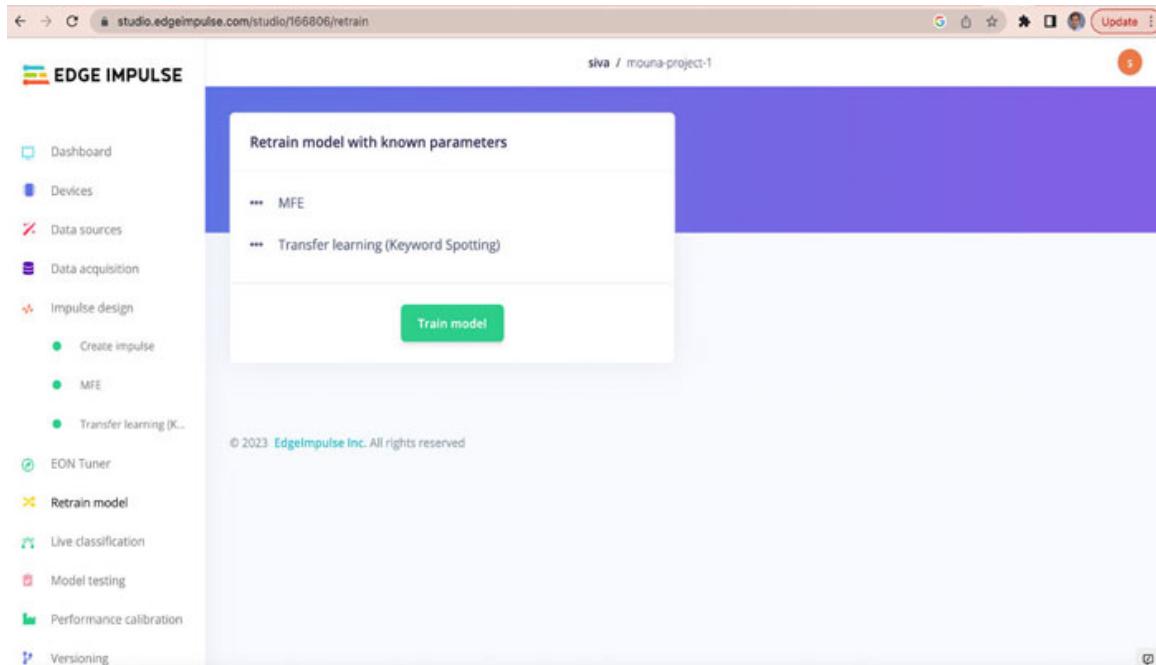


Figure 4.22 (b): Model retraining with Edge Impulse



Available optimizations for NN Classifier

Quantized (int8) ★	RAM USAGE 1.5K	CONFUSION MATRIX
Currently selected	LATENCY 1 ms	<small>?</small>
This optimization is recommended for best performance.	ROM USAGE 14.3K	
	ACCURACY 89.91%	
Unoptimized (float32)	RAM USAGE 1.4K	CONFUSION MATRIX
Click to select	LATENCY 1 ms	<small>?</small>
	ROM USAGE 14.4K	
	ACCURACY 89.87%	

Estimate for Cortex-M4F 80MHz (ST IoT Discovery Kit)

Build

Figure 4.22 (c): Neural Network Model Classifier optimization selection in EON Compiler

Figure 4.22 (a) to (c): EON compiler and ML model training in Edge Impulse

Model Compression

Let us now learn about Model Compression and its different aspects.

Pruning

Pruning is a method that is utilized in model compression for the implementation of TinyML. This method entails deleting weights and connections from a neural network model that are deemed to be not required. It is predicated on the concept that a neural network model typically has many redundant or unnecessary connections and weights, which do not significantly contribute to the performance of the model. By removing some of these connections and weights from the model, it is possible to make it more efficient and make it smaller. This is especially helpful for deploying the model on small devices with limited resources, such as those used in TinyML applications. The complexity of the model should be simplified to keep or even improve its performance, and this is the objective of the pruning process.

There are several distinct methods for pruning, but one way that is frequently used is to locate the connections and weights within the model that have the least amount of significance and then eliminate those components. This can be accomplished by hand, or it can be done automatically using algorithms that look for and get rid of the connections and weights that are not as critical. **Structured pruning** is another method that can be utilized, and this technique entails erasing entire layers or groups of neurons from the model.

It is necessary to be careful not to eliminate too many connections or weights when pruning a neural network model because doing so can have a detrimental influence on the performance of the model. Pruning is a method that can be successful in reducing the size and complexity of a neural network model. When determining which connections and weights to prune, it is essential to do a thorough analysis of the trade-offs between reducing the size of the model and improving its performance.

There are a few different sorts of pruning strategies that can be utilized in order to compress a model in preparation for its implementation of TinyML. These techniques include **weight pruning, unit pruning, and low-rank factorization**, and they are explained as follows:

- The process of **weight pruning** entails removing the weights from the model that have the least amount of magnitude. This can be accomplished by determining a cutoff point for the weights and then setting all weights that fall below that point, to zero. It is possible to execute weight pruning in an iterative manner, with the objective of gradually reducing the number of weights in the model, while attempting to have as little of an effect as possible on its performance.
- The process of eliminating whole units or neurons from a model is referred to as "**unit pruning**." This can be accomplished by determining which units

have a negligible influence on the outcome of the model and then deleting those units.

- A type of pruning called **low-rank factorization** involves decomposing the weight matrices of the model into the product of two lower-dimensional matrices. This is done as part of the process of pruning. This can result in a large reduction in the model's total number of parameters, which in turn can increase the model's computing performance.

To further minimize the size of the model and enhance its performance on devices with limited resources, pruning can be paired with additional approaches for model compression, such as quantization and model distillation.

Knowledge distillation

Knowledge distillation is a method for model compression, that entails teaching a more compact model, sometimes known as the "student model," to behave in the same way as a more extensive model, that has already been pre-trained (the "teacher" model). The purpose of knowledge distillation is to transfer the knowledge that is encoded in the teacher model to the student model, so that the student model can perform similarly to the teacher model, while having a smaller model size and faster inference speed. This is accomplished through the process of knowledge transfer.

The following is an outline of the broad steps involved in the process of training a student model to use knowledge distillation:

1. Begin with a teacher model that has already been pre-trained, and a dataset.
2. Train the student model on the dataset by employing a mix of supervised learning (by making use of the labels provided by the ground truth) and unsupervised learning (using the output of the teacher model as a "soft" target).
3. The student model is trained to mimic the behavior of the teacher model by minimizing the difference between the teacher's output and the student's output. This is typically done with a loss function that combines both the supervised loss (based on the ground truth labels) and the unsupervised loss. In this way, the student model is trained to mimic the behavior of the teacher model (based on the output of the teacher model).

One of the most important advantages of knowledge distillation is that it makes it possible for a student model to gain knowledge from a teacher model, despite the fact that the teacher model may be considerably larger and more complicated than

the student model. This can be especially helpful for applications utilizing TinyML (small machine learning), in which both the size of the model and the computational resources available are restricted.

It is important to keep in mind that the distillation of one's knowledge is simply one of the various methods that can be utilized for the process of model compression. Pruning is also known as the elimination of superfluous or redundant parameters from a model. Quantization is also known as the reduction of the precision of model parameters. Lastly, model distillation refers to some other strategies (compressing a model by training a smaller model to mimic the behavior of a larger model).

Model compression can be accomplished with Edge Impulse in one approach. Edge Impulse provides tools for model compression and optimization, as well as the ability to train and refine machine learning models right on your device.

To use Edge Impulse for model compression, you can follow these steps:

1. **Collect and label a dataset:** First, you need to collect and label a dataset to train your ML model on. This can be done using the Edge Impulse Studio, which allows you to collect data from sensors and label it for training.
2. **Train a ML model:** Next, you can use the Edge Impulse Studio to train a machine learning model on your dataset. The platform provides a range of pre-built machine learning models that you can use, or you can build your own custom model using TensorFlow or Keras.
3. **Optimize and compress the model:** Once you have trained a machine learning model, you can use the Edge Impulse Studio to optimize and compress the model for deployment on your edge device. This can be done using techniques such as quantization and pruning, which reduce the model size and complexity without significantly impacting performance.

Model conversion

Let us now learn about Model conversion and its different aspects.

Quantization

Quantization is the process of decreasing the precision of a neural network model's weights and activations, to decrease the model's memory footprint and computational complexity. This is accomplished by a technique known as "quantization."

There are two main types of quantization:

- **Weight quantization:** It involves reducing the precision of the weights of the model, typically from 32-bit floating point numbers to 8-bit integers or even lower-precision fixed-point numbers.
- **Activation quantization:** It involves reducing the precision of the activations of the model, typically from 32-bit floating point numbers to 8-bit integers or lower-precision fixed-point numbers.

There are several techniques that can be used to perform quantization, including **dynamic fixed-point quantization**, **static fixed-point quantization**, and **quantization-aware training**, and they are explained as follows:

- Dynamic fixed-point quantization involves converting the weights and activations to fixed-point numbers at runtime.
- Static fixed-point quantization involves converting the weights and activations to fixed-point numbers ahead of time and storing them in the model.
- Quantization-aware training involves training the model with quantization in mind, using techniques such as fake quantization to mimic the effects of quantization during training.

Edge Impulse includes several features for model compression, including:

- **Automated model pruning:** Edge Impulse can automatically identify and remove unnecessary weights and connections from a model, resulting in a smaller and more efficient model.
- **Quantization:** Edge Impulse can convert a model's floating-point weights and activations to fixed-point values, which requires fewer bits to represent and can result in a more efficient model.
- **Weight sharing:** Edge Impulse can identify shared weights within a model and consolidate them into a single weight, reducing the number of parameters in the model.
- **Low-precision inference:** Edge Impulse can reduce the precision of the weights and activations in a model during inference, while maintaining accuracy.

By using these techniques, it is possible to significantly reduce the size and computational complexity of a machine learning model, making it more suitable for deployment on edge devices.

Figure 4.23 features the ML model compression in Edge Impulse:

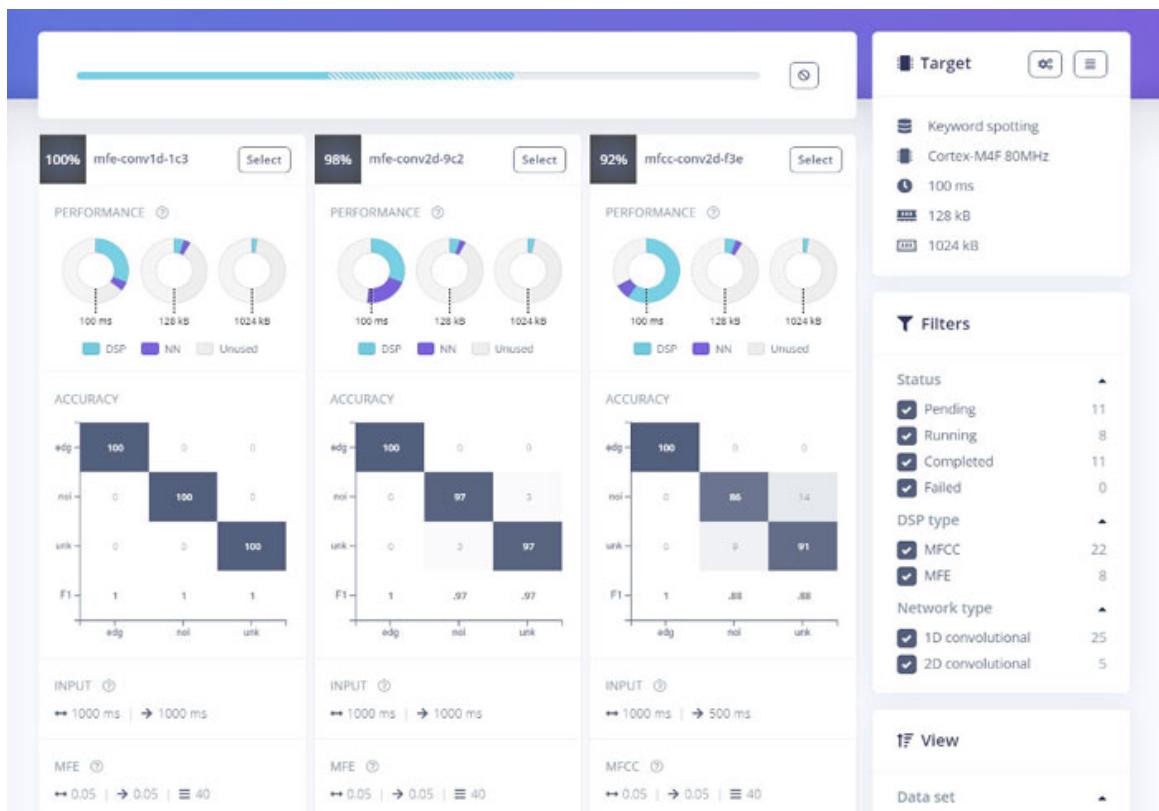


Figure 4.23: ML model compression in Edge Impulse

Inferencing/Prediction of results with test data

The practice of utilizing a ML model that has been trained to make predictions on new data that has not been seen before, is referred to as inferencing in TinyML. This method can alternatively be referred to as "inference" or "prediction," depending on the context.

To use TinyML's inferencing capabilities, you will need to have a ML model that has been trained, as well as a collection of test data that you want to use to evaluate the performance of the model. The data used for testing should not be the same kind of data that the model was trained on, but it should be indicative of the kind of data that the model will be exposed to, when it is used in the real world.

To execute inferencing, you will first run the test data through the trained model, and then you will make predictions about the data, based on the output of the model. For instance, if you have trained a model to categorize photographs of various species of animals, you can use inferencing to determine the species of animal depicted in a new image that the model has never seen before. This allows you to make accurate predictions about what the model is seeing.

It is essential to keep in mind that the precision of the model's forecasts will be influenced both by the quality of the data used for training the model and by the structure of the model itself. The outputs of the inferencing may be erroneous if the model was not trained on a diverse and representative set of data, or if the model is not well-suited to the task it is being utilized for.

It is essential to keep in mind that the purpose of inference is not to provide more training to the model; rather, it is to assess how well the model works with fresh information. This can help you understand how effectively the model generalizes to data, that has not yet been seen, and it can also give you an indication of how well the model might perform in conditions that are more representative of the actual world.

In the context of microcontrollers, inferencing is a method that may be used to speculate on the actions that will be taken by a system or process, based on the information that has been gathered from various sensors and inputs. For instance, a ML model that has been trained on the data that has been gathered by a temperature sensor, may be used to make predictions regarding the temperature of a room, based on new input data that has been acquired by the sensor.

It is necessary to first load the learned ML model into the memory of the microcontroller before attempting to perform inferencing using the microcontroller. After that, the processing capacity of the microcontroller can be used to apply the model to the test data, which will result in the generation of an inference or prediction. Embedded software makes it possible to automate this process, which enables the microcontroller to continuously gather and evaluate data and to generate predictions in real time.

Figure 4.24 features ML model testing in Edge Impulse:

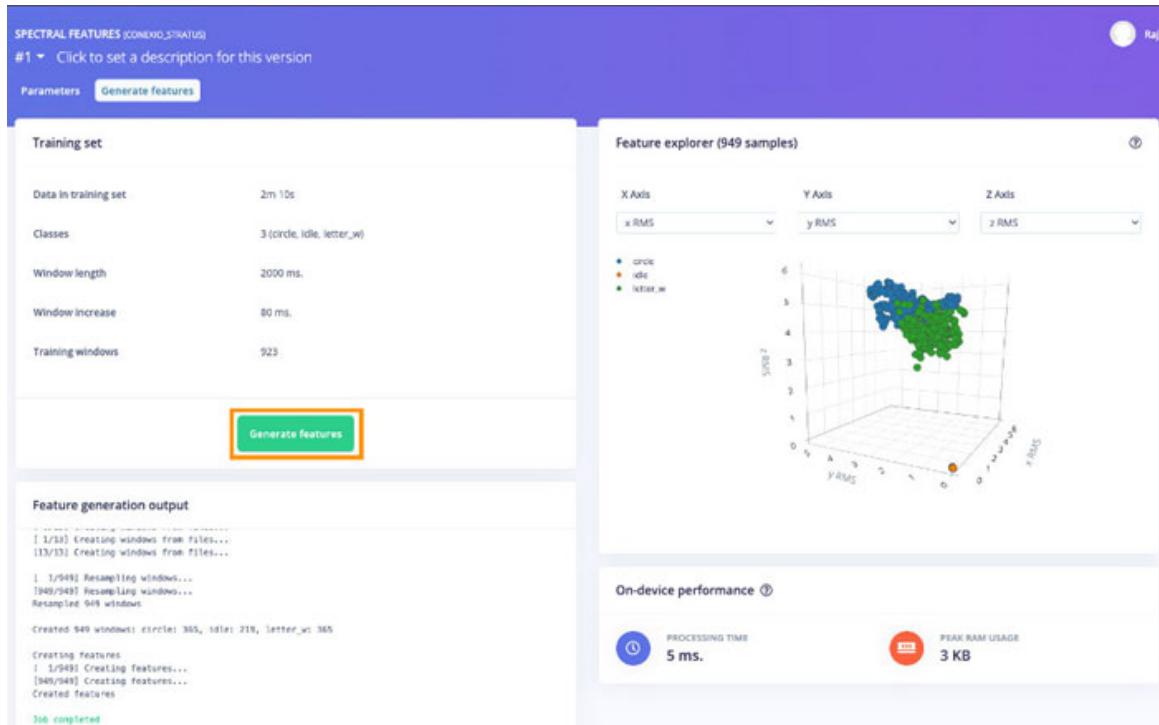


Figure 4.24: ML model testing in Edge Impulse

Model Deployment in TinyML Hardware board

It is essential to keep in mind that deploying a ML model on a TinyML hardware board can be a challenging task, and it may be necessary to go through a process of trial and error, to get the model to operate efficiently on the device. Despite this, it is possible to deploy a ML model on a TinyML hardware board and carry out machine learning activities at the edge of the network if adequate planning and attention to detail are exercised.

The steps required to deploy a machine learning model in a microcontroller, such as an Arduino or Raspberry Pi or any other hardware board, include the following steps:

1. **Converting the model:** Once the ML model has been trained, the next step is to change it into a format that the microcontroller can understand and use. In this step, you might need to use a program such as TensorFlow Lite or ONNX to transform the model into a format that the processor of the microcontroller will be able to understand and run.
2. **Integration:** Once the model is in a format that is compatible, you will be able to incorporate it into the code for your microcontroller. In order to load and run the model, it may be necessary to make use of a library or

Application Programming Interface (API) that is offered by the operating system of the microcontroller.

3. **Testing:** As a last step, you will need to perform tests on the model using the microcontroller to validate that it operates as intended. As part of this process, the performance of the model may be evaluated, based on data collected in real time by sensors or other inputs.

TensorFlow Lite and PyTorch Micro are only two examples of the many tools and frameworks that are at your disposal, to assist you with the deployment of models on microcontrollers. It is also possible to deploy models by utilizing custom code, although doing so may need additional work and knowledge.

Approaches to deploying machine learning models on microcontrollers include the following:

- **Using pre-trained models:** It is possible to use pre-trained models that have already been trained on a larger dataset, and then fine-tune them for the specific application. This can help reduce the computational resources required for training and allow the model to be deployed on a microcontroller.
- **Quantization and Model pruning:** Discussed in the previous sections.
- **Hardware acceleration:** Some microcontrollers have hardware acceleration capabilities that can be used to accelerate the execution of ML algorithms.

Overall, deploying ML models on microcontrollers requires careful optimization and design, to ensure that the model fits within the available resources and can run in real-time. Security and privacy concerns can be addressed during the deployment of the TinyML model. As the TinyML systems may handle sensitive data or be deployed in sensitive environments, it is important to consider security measures such as encryption and secure communication protocols.

[Figure 4.25](#) features the ML model deployment in Edge Impulse:

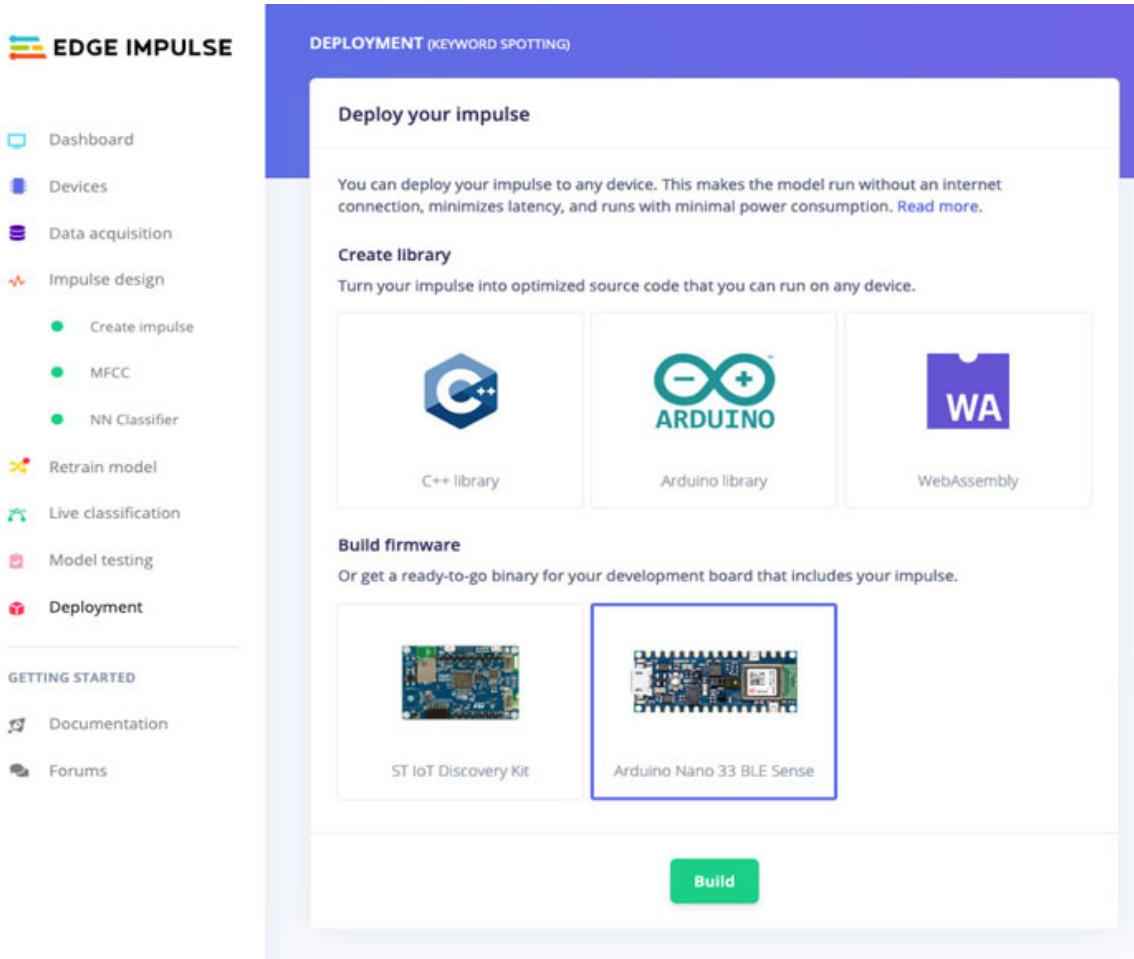


Figure 4.25: ML model deployment in Edge Impulse

Conclusion

In this chapter, we understood EML, its characteristics, different types, and sample example systems. We learned in detail about the Edge Impulse and Arduino IDE tools usage, as well as the different components in it. We studied different approaches in data collection steps from sensors and TinyML tools. In data engineering, we focused more on the topics such as data cleaning, and transformations required on the data. Model training using EON compiler was also studied. Model compression, especially pruning and knowledge distillation topics were focused on, and model conversion was understood in detail with the help of different quantization techniques. Lastly, we covered Model inferencing and deployment techniques on different hardware boards, with the help of Edge Impulse Studio.

Key facts

- **Embedded Machine Learning (EML)** refers to the use of ML techniques and algorithms in devices and systems that are embedded in the physical world, such as sensors, drones, robots, and other **Internet of Things (IoT)** devices.
- EML systems are designed to operate in real-time and often have limited computing resources, such as memory and processing power. The goal of EML is to enable these devices to learn and adapt to their environments and make decisions without the need for a central server or external processing.
- Some of the key building blocks utilized in EML systems are sensors and hardware, such as **Digital Signal Processors (DSPs)**, **Field-Programmable Gate Arrays (FPGA)**, and **Application-Specific Integrated Circuits (ASICs)**; Operating Systems (FreeRTOS, VxWorks, and μC/OS), and Communication Interfaces.
- When deploying a TinyML model, you will need to consider the hardware constraints of the target device. This includes the available memory, processing power, and power consumption.
- Edge-impulse and Arduino IDE, along with their building blocks are explained in detail, for performing different TinyML operations.
- In order to train a machine learning model, you will need a large amount of labeled data. When deploying a TinyML model, it is important to consider how you will collect and label the data, as well as how you will store it on the device.
- In data engineering, data cleaning, organization and data transformations are very important steps for the proper ML model training, in order to achieve good performance for TinyML applications.
- In ML model conversion, different quantization techniques such as weight quantization, activation quantization, dynamic fixed-point quantization, static fixed-point quantization, and quantization-aware training, are useful techniques.
- For deployment of ML trained models, Edge Impulse supports many hardware boards such as Syntiant TinyML, Arduino Nano 33 BLE sense and so on.
- After deploying your TinyML model, it is important to evaluate its performance in order to ensure that it is meeting your desired accuracy and efficiency goals. This may involve collecting additional data and fine-tuning the model.

Questions

1. What is Embedded Machine Learning?
2. List key building blocks and give some example systems of EML.
3. Discuss the pros and cons of EML systems?
4. How will the model be deployed to the target device, and what tools and frameworks will be used for this process?
5. What are components of Edge Impulse Tool?
6. In which versions is Arduino IDE available?
7. List some of the pros and cons of Arduino IDE.
8. Discuss the effect of outliers in data cleaning.
9. What is encoding? List the types of encoding.
10. How will the model be trained and fine-tuned? Will it be done on the device itself or on a separate platform?
11. Discuss pruning and knowledge distillation and how they are helpful in TinyML model compression?
12. What is low-rank factorization?
13. How will security and privacy concerns be addressed during the deployment of the TinyML model?

References

1. Schizas, N., Karras, A., Karras, C. and Sioutas, S., 2022. TinyML for Ultra-Low Power AI and Large Scale IoT Deployments: A Systematic Review. *Future Internet*, 14(12), p.363.
2. Saha, S.S., Sandha, S.S. and Srivastava, M., 2022. Machine Learning for Microcontroller-Class Hardware--A Review. arXiv preprint arXiv:2205.14550.
3. <https://docs.edgeimpulse.com/docs/>
4. <https://docs.arduino.cc/>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 5

Real World Use Cases

Introduction

By now, the reader should have got a good idea of what **Artificial Intelligence (AI)** is and its deployment in TinyML form. Before we proceed to show how to implement an example of AI deployment from concept to lab testing, it may be a good time to see what people can achieve with this technology. In [Chapter 1, Introduction to AI](#), we looked briefly at where TinyML is being used. The AI revolution has just started and so it is difficult to say how many places it could be used in, in the future. As a matter of fact, the purpose of this book is to give its reader tools, so that they can transform the world through AI. To inspire readers, we will give some examples where AI has been successfully demonstrated and deployed.

Structure

In this chapter, the following topics will be covered:

- Smart agriculture
 - Agriculture video analytics
 - Crop intruder detection
 - Crop yield prediction and improvement
 - Agribots
 - Insect detection and pesticide reduction
 - Weedicide elimination
 - Acoustic insect detection
 - Animal husbandry
- Smart appliances
 - Vision AI for appliances

- Audio AI for appliances
- Sensor based AI for appliances
- Smart cities
 - Safe and secure city
 - City maintenance
 - Parking enforcement
 - Traffic management
 - Maintaining bridges
 - Nonsmoking enforcement
- Smart health
 - Cataract detection
 - Fall detection
 - Cough detection
 - Boxing moves detection
 - Mosquito detection
 - Snoring and sleep apnea detection
- Smart home
 - Person detection at the door
 - Glassbreak detection
 - Smart baby monitor
 - Voice recognition for home automation
- Smart industry
 - Railway track defect detection
 - Telecom towers defect detection
 - Defect detection in components
- Smart automotive
 - Drowsy driver alert
 - Advance collision detection

Objectives

In this chapter, we will provide numerous examples where AI has already been considered or could be considered. By no means is this an exhaustive list. Sometimes, it is difficult to classify a solution as it may belong to multiple categories and in some cases, it is starting its own category.

Smart agriculture

Agriculture is a labor-intensive job. It requires hard manual work in an open field. Throughout the world, as educated workforce prefer to work in cities in the comfort of air conditioning, the agriculture industry is often neglected. Since the green revolution, chemicals have been abused for massive usage as pesticides and insecticides. Now, intelligent technology is expected to reduce the use of pesticides and weedicides. A short video on YouTube (<https://www.youtube.com/watch?v=Qmla9NLFBvU>) gives a glimpse of how agriculture is using technology to increase production. The following examples show how AI can further improve the effectiveness of the technology.

Agriculture video analytics

With wide usage of cameras in cell phones, the camera technology has evolved tremendously in the last decade. The small size and low cost of the camera makes it possible to use them in drones. The resolution and picture quality has been ever improving. Video analysis has been used in agriculture for many different reasons. [Figure 5.1](#) shows a picture of a drone specially designed for agriculture:



Figure 5.1: Agriculture monitoring drone technology. **Source:** <https://www.dslrpros.com/agricultural-monitoring-drone.html>

Figure 5.2 shows an example of a picture taken by a drone. The analytics embedded in the drones avoid streaming of the live video. When a section of the field looks anomalous, the drone takes pictures and tags it with the location of the field. This reduces manual work by orders of magnitude. *Nilg.ai* is pioneering in bringing AI to the agriculture field.



Figure 5.2: An image taken by a drone which can differentiate between a healthy and unhealthy plant

Crop intruder detection

With video monitoring, it can be identified who is breaching the crop. Human intruders or even wild animals can be identified using image recognition algorithms. Twenty20 solutions is pioneering in this field (<https://www.twenty20solutions.com>).

Crop yield prediction and improvement

Once crops are monitored, its yield can be correlated to environmental conditions. Intelligent sensors capturing in-ground moisture, fertilizer and natural nutrients are readily available and are distributed throughout the farm. Once significant data is captured, it can be used to predict the crop yield. The data can be used to optimize some variables under control to improve the yield.

Agribots

Worker shortage is tackled by AI powered self-driving smart tractors and agriculture specialized robots termed as agribots. **VineScout** is a pioneer in providing one of the agribots.

Figure 5.3 features a robot utilizing AI for self-navigation:



Figure 5.3: Self driving agribot monitoring the farms. Source: <http://vinescout.eu/web/>

Insect detection and pesticide reduction

Drones can inspect the crop for pesticide infected areas. By analyzing the infected area, the type of pest can be inferred. Vision based AI can further verify the pests in different parts of the field. By targeting the infected area, many agri-AI companies claim to reduce pesticide by 90%.

Weedicides elimination

Unintended plants fight for the resources and can reduce the yield, and so it is important to remove the unwanted plants in the field. Weedicides which attack certain types of plants have been used exhaustively over the last several decades. The broad-spectrum herbicide glyphosate (common trade name “Roundup”) was first sold to farmers in 1974. Since then, **Glyphosate Based Herbicides (BGHs)** have seen 100x growth in their usage. Now it is considered as probably carcinogenic to humans. Whether or not weedicides are carcinogenic, their uncontrolled usage is making a huge impact on the environment. Now weeds are visually detected using cameras and they are either mechanically removed or removed by laser. This new method eliminates the use of weedicides altogether (<https://www.youtube.com/watch?v=3O42e4-u7hM>)

Figure 5.4 shows the images where weeds are detected and their locations are automatically marked:



Figure 5.4: Vision based AI detecting the weeds. Source: <https://www.frontiersin.org/articles/10.3389/fpls.2021.732968/full>

Acoustic insect detection

AI using audio signals can be used in agriculture to detect insects with their chirping sounds and humming sound when they fly. The insects which are too small to be spotted by the visual AI, can be detected by characteristic acoustic sound they may produce, which could be their chirping sound or buzzing sound.

[Figure 5.5](#) shows a picture of destructive pest which can be detected by their buzzing sounds:



Figure 5.5: Picture of a destructive insect which can be detected by its buzzing sound. Source: <https://www.ledfordspestcontrol.com/blog/pest-facts/10-destructive-garden-pests/>

Animal husbandry

Different types of sensors can be combined with the AI to make them smart. One such example is to analyze farming animals by motion sensors. By monitoring their movements, their health can be inferred. If they are lethargic, then the sensors will infer that the animal is sick. The accuracy can further be increased by adding temperature and other bio sensors in the mix. [Figure 5.6](#) shows a sensor which is clipped into the cow's ear for sensing its health:



Figure 5.6: A sensor is clipped into the cow's ear for sensing its health. **Source:** <https://www.cowmanager.com/cow-management/modules/health/>

Smart appliances

We interact daily with appliances such as the washing and drying machine, dish washer, microwave, oven-range, refrigerator, air conditioners and so on. The appliances have come a long way and are much more efficient and user friendly than ever before. Now, these can be made more efficient and friendly with AI. Not only will the appliances be convenient to use in the future but they will be safer as well.

Vision AI for appliances

Imagine what the appliances could do if they could see! If a microwave oven can see what kind of food is kept inside it, it can recommend the proper setting. It can use infrared cameras to map the temperature of the food, so

that it can adjust its turning and intensity to produce evenly heated food at the right temperature.

Figure 5.7 shows an example where a microwave oven is fitted with an internal camera. The camera can see the food and then can adjust the cooking for perfecting the food.

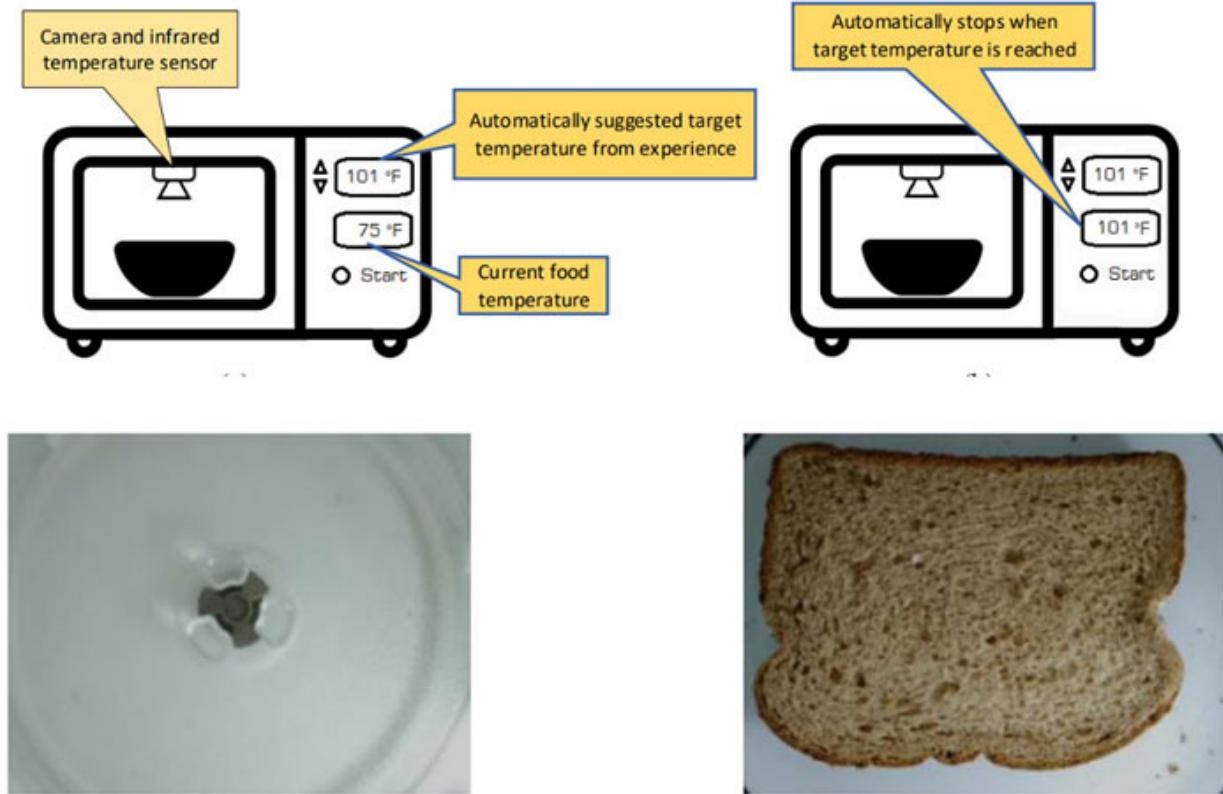


Figure 5.7: Camera sees the turntable. It will not cook if an accidental cook button is pressed. By analyzing the picture of the food, the cooking can be controlled for perfection. Source: https://pdfs.semanticscholar.org/e3c9/634f6917a687b50566e58f8e95ce53e72d0f.pdf?_ga=2.36848040.988300366.1666041096-2016201435.1666041096

A fridge can look inside it and detect if a certain item is about to reach past its expiry date. The odors can be analyzed and the user can be alerted about the rotting material.

Figure 5.8 illustrates an example where the fridge determines what kind of raw foods are present inside it and then it can suggest recipes which can be made by available raw food:



Figure 5.8: Fridge recommends recipes based on the ingredients available in the fridge. *Source:* <https://news.samsung.com/us/new-food-ai-looks-inside-fridge-help-find-perfect-things-cook-already/>

By analyzing human presence, the air conditioning can be managed room to room. This can reduce the burden on the air conditioner so that it will last longer. Not to mention, it will save energy which will further save the planet.

Audio AI for appliances

Home appliances can use both voice and non-voice AI. The voice command can start or stop a cycle. In an ultimate user interface, we need to just say what we want, the same way we would interact with a chef or household help. We can “tell” the oven that we want to bake a cake of a certain kind. After that, we do not have to control the temperature and time. The intelligent oven will look at the cake and optimize control of temperature and time.

Figure 5.9 features a voice controlled smart range:



Figure 5.9: A voice controlled smart range which can set its settings based on the voice commands saving time of the user. **Source:** <https://www.youtube.com/watch?v=YQr0oulqCuQ>

The non-voice audio events can be detected by the appliances to detect their own failure mechanism for early warning.

Sensors based AI for appliances

There are many types of sensors which most appliances use, and there are many more sensors which are used in the industrial applications, which can be used in appliances as the cost of the sensors is pretty affordable . The sensors could be **Inertial Measurement Units (IMU)** which determine acceleration and angular motion, gas detection sensors, temperature, pressure, moisture detection and so on. Sometimes, it makes sense to combine data from multiple sensors to make better predictions, and this is termed as sensor fusion.

Smart cities

Farming is getting more automated, and more and more people are living in cities. With limited resources, life in cities must be managed by authorities. AI can help make cities more convenient to be in and safer.

Safe and secure city

Today, video surveillance footage is used in almost all criminal cases. With widespread video surveillance coverage, AI can add a new dimension to them. For important locations, a wall of video feed is monitored with few individuals. It is not practical for one person to pay attention continuously to one video feed, let alone multiple video feeds. AI can assist in real time tracking and alerting when an abnormal incident is happening.

With reliance on video monitoring, stores are opening with no checkout, thus saving shoppers their precious time. With deep AI, subtle intents can be predicted before an incident even happens. With some way of letting the perpetrators know that they are being watched, the incidence could be avoided.

License plate reading can track vehicles and be alerted if they are driving dangerously or have an imminent danger from other drivers.

City maintenance

A company called **RoadBotics**, has developed a technology using artificial intelligence to analyze road conditions and then recommend repairs. By analyzing all the repairs, cities can plan better and prioritize the maintenance.

Figure 5.10 shows where roads can be monitored by unmanned cars and drones. The location of needed repairs and urgency of the repair needs is documented automatically, saving money for the cities so that they can spend the money in actual repair.



Figure 5.10: A picture taken by a visual AI system tags the problem and maps it on the map. **Source:** RoadBotics - Make Data-Driven Decisions

Parking enforcement systems

For equitable parking, cities control the parking systems. With traffic police, the tracking system is quite costly. With license plate recognition technology, parking can be enforced for outstayed hours and illegal parking.

Traffic management

AI technology is good at analyzing road traffic. There are several algorithms which can count things even if they are very close together. With real time detection of traffic and root cause of the issues, road signals can be controlled to reduce the traffic jams. *Figure 5.11* shows a traffic analyzing system. The traffic signs can be managed to accommodate pedestrians and cyclists. *Figure 5.11* illustrates this:



Figure 5.11: For safety of pedestrians and cyclists, cameras can determine who is on the road.

Source: <https://studio.edgeimpulse.com/public/108632/latest/dsp/image/3>

Maintaining bridges

Bridges can be monitored using IMU. The vibrations can be monitored by smart vibration detectors when heavy vehicles pass over them. If there is any deviation or anomaly, then alarms can be generated.

Figure 5.12 shows a bridge which can be fitted with IMU sensors for monitoring vibrations. Campbell Scientific claims to use their technology for monitoring bridges and other structures.



Figure 5.12: Campbell Scientific claims to monitor bridges and other structures using IMUs. **Source:** <https://www.campbellsci.com/bridge-monitoring>

Non-Smoking enforcement

Despite sincere warning and nonsmoking signs, people still smoke in public places because enforcement is not easy. With smart smoke detectors and vision AI, people can be caught smoking in forbidden places, for example, close to schools, hospitals and so on. Refer to [Figure 5.13](#):



Figure 5.13: Smart smoke detectors along with vision-based AI can monitor people who do not follow the public rules. Source: <https://philnews.ph/2017/12/14/man-spotted-smoking-front-no-smoking-signage/>

Smart health

There are numerous wearables which are aimed at monitoring our health and they can intervene, in order to thwart an imminent threat. For example, the heart rate monitor can determine if the heart rate is becoming abnormal and can predict if there is a high chance of a heart attack. Similarly, epilepsy can be predicted in advance, giving the user some time to react and reach a safe place.

A small ring made by Aura can monitor users' sleep and provide recommendations on how sleep can be better. **Figure 5.14** shows the Aura ring and sleep analysis which can be viewed in a Bluetooth paired smartphone:

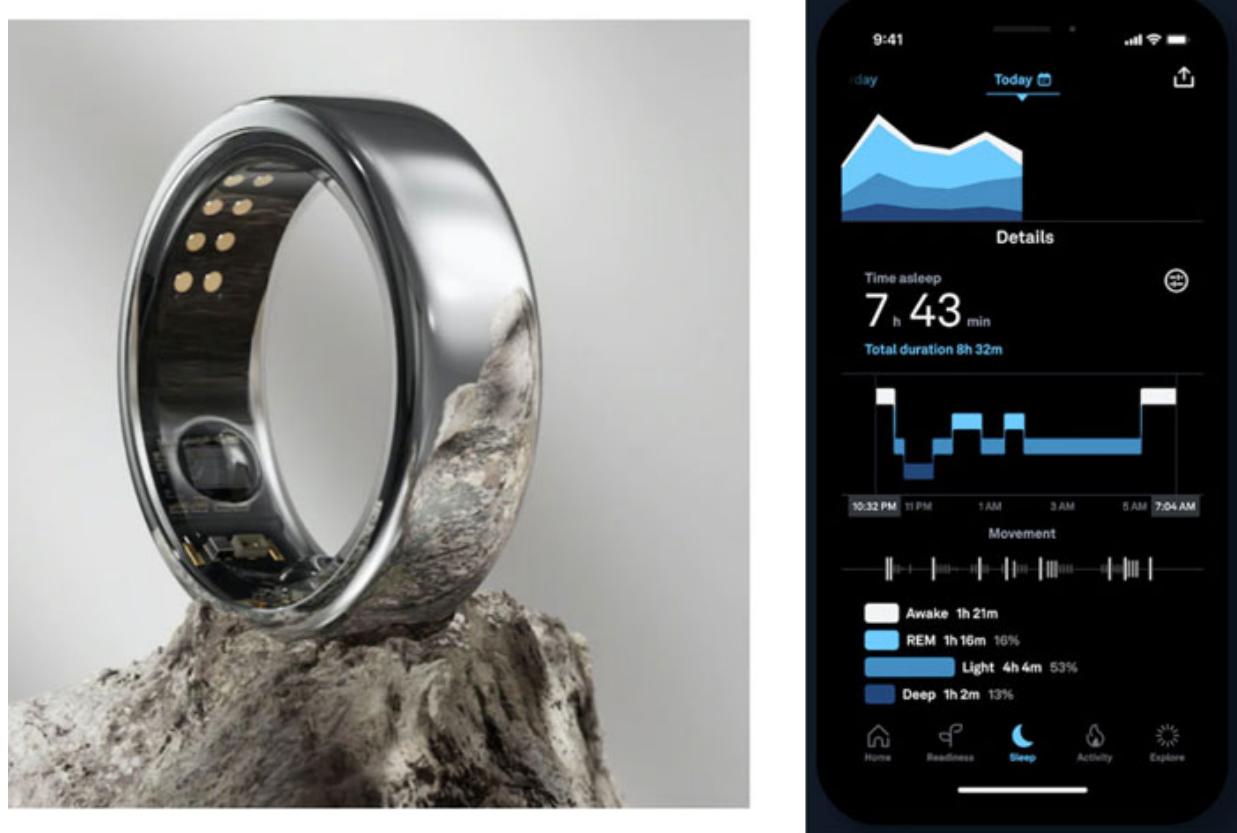


Figure 5.14: Aura ring (left) and Sleep analysis (right). Source Ouraring.com

Cataract detection

Picture taken of eyes from a typical cell phone can be analyzed by AI and cataract can be diagnosed. For rural areas this could be a boon for elderly population.

Figure 5.15 shows Data set provided by Kaggle. Vision AI can predict the presence of cataract in the eye:

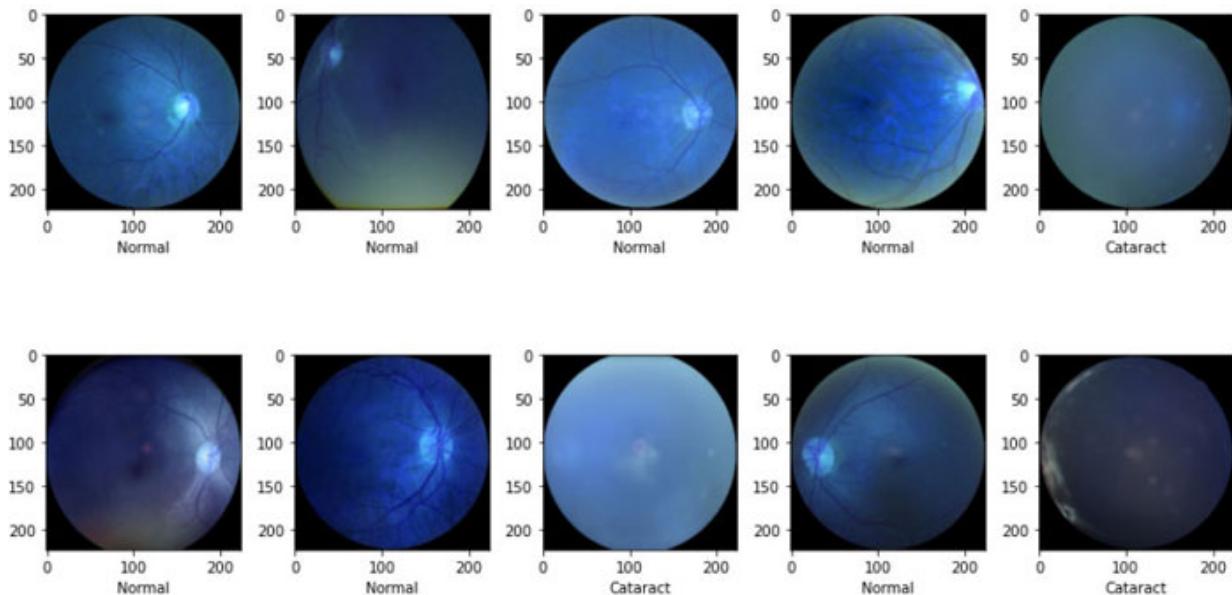


Figure 5.15: Cataract can be detected from the vision AI. Source: Cataract Prediction | Kaggle

Fall detection

Falling is a common problem for elderly. Fall can be detected with IMU built in the specialized smart watch, such as the Kanega watch (<https://www.unaliwear.com/why-kanega-watch/>). The watch can send alerts for paramedics to arrive without someone calling them explicitly. This can be a lifesaver for elderly who live by themselves.

Figure 5.16 is showing a scenario where fall detection can be a lifesaver.



Figure 5.16: The picture depicts an elderly woman who has fallen and no one is around. A wearable device can detect the fall and alert loved ones or senior care professionals. **Source:** <https://www.homechoicehomecare.com/injury-prevention-safety/preventing-falls-home-care-elderly/>

Cough detection

By listening to the coughing sound, pneumonia can be detected. By placing these detectors in hospitals, schools, public transportation systems, the pandemic can be monitored and policies can be changed over time, depending on the severity of the pandemic.

Figure 5.17 shows a cough detection system which can process the cough sound in real life and can alert the officials:

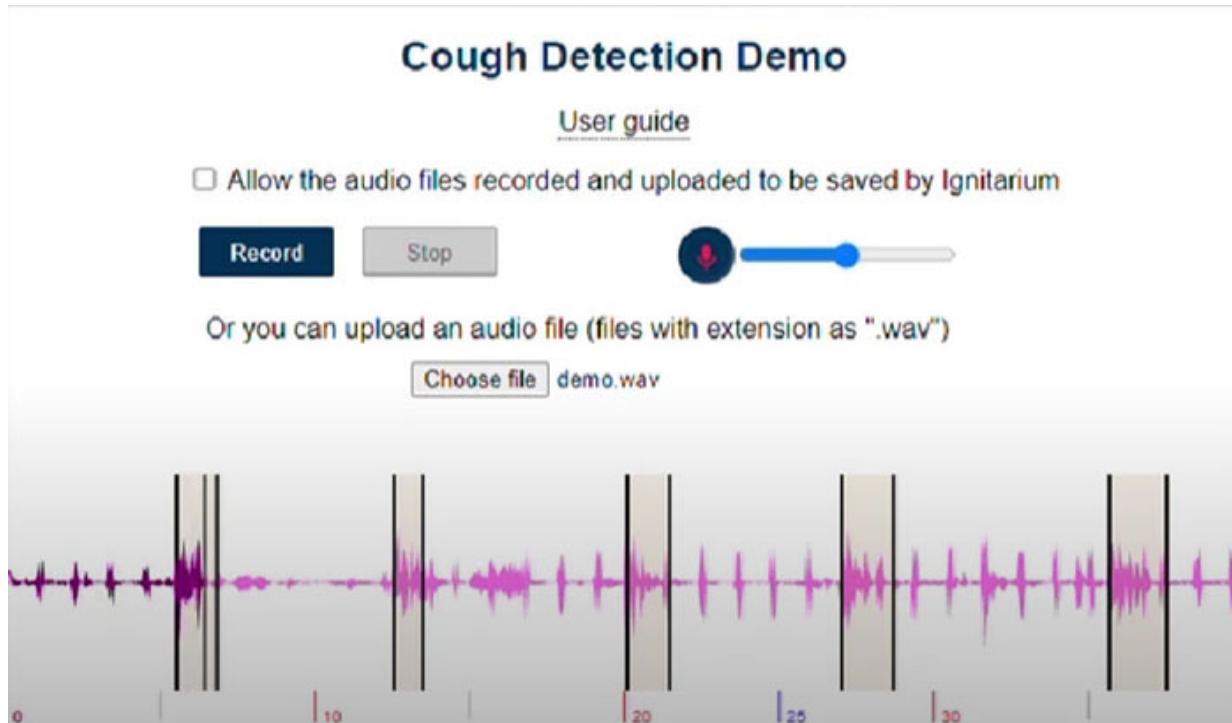


Figure 5.17: Cough detection system. The wav file is marked with boundaries where caught was detected. **Source:** <https://www.youtube.com/watch?v=5IuoIGHmUrk>

Boxing Moves Detector

By wearing IMU sensors at different parts of the hands, hand movements can be monitored. The models will be trained by the coaches and professionals. Once students wear the same sensors, their movements can be monitored and suggestions can be made.

Figure 5.18 shows various boxing moves which can be detected by a sensor worn. An open-source project is created by Edge Impulse can be found here: <https://studio.edgeimpulse.com/public/86666/latest>

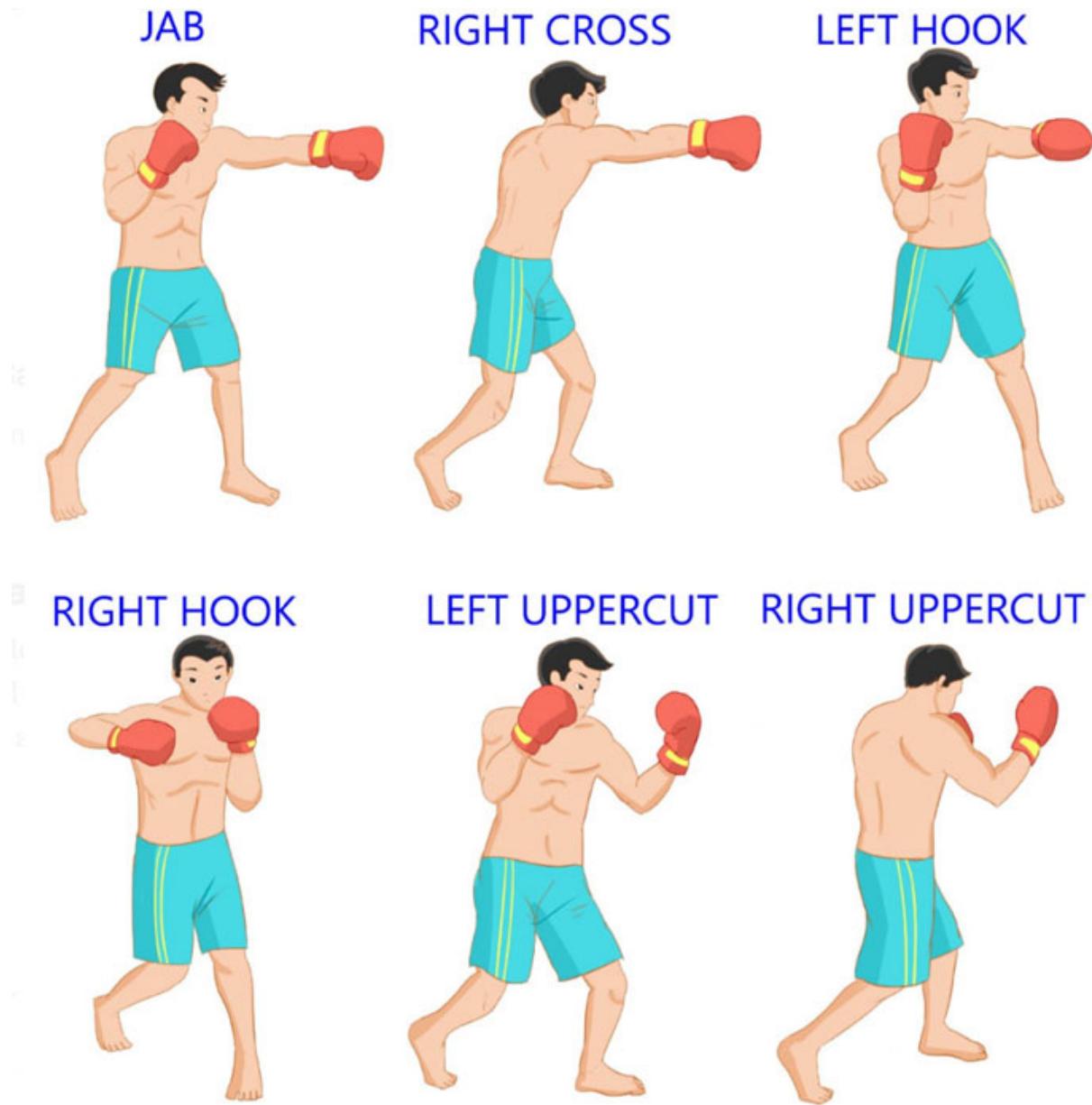


Figure 5.18: Various boxing moves can be monitored by IMU sensors worn by the student. **Source:** <https://medium.com/@chadwick.balloo/boxing-punches-one-through-six-169b3ba2a7a0>

Mosquito detection

From the buzzing sound of mosquitos, their species can be detected. Specific mosquito species carry transmit specific disease, for example, yellow fever, dengue, chikungunya virus, Zika virus and others. Detection of specific species can be helpful in early detection and elimination of such mosquitos.

A project done by Nari Johnson, Rose Hong, and Joyce Tian shows how easy it is to implement such a solution. [Figure 5.19](#) shows pictures of different species of mosquitoes which can be detected by their humming sounds:



Female of *Culex pipiens* and a tiger mosquito (*Aedes albopictus*).

Figure 5.19: Different species of mosquitoes can be detected using AI by analyzing their humming sound. Source: <https://www.irta.cat/en/a-smart-trap-classifies-mosquitoes-by-sex-and-genus-based-on-their-buzzing-sound/>

Snoring and sleep apnea detection

Though snoring may be considered harmless, sleep apnea could have significant effects on the health. Sleep apnea is a sleep disorder where breathing stops and starts when a person is sleeping. Though it is not fatal because the patient wakes up gasping for air, it has other side effects, for example, insomnia, headache, irritability. It mostly gets unnoticed as the patient and family members are asleep. A snoring detector can not only catch the snoring and sleep apnea, but it can also even quantify how bad the situation is.

[Figure 5.20](#) shows the origin of sleep apnea which can be detected by analyzing snoring sound:

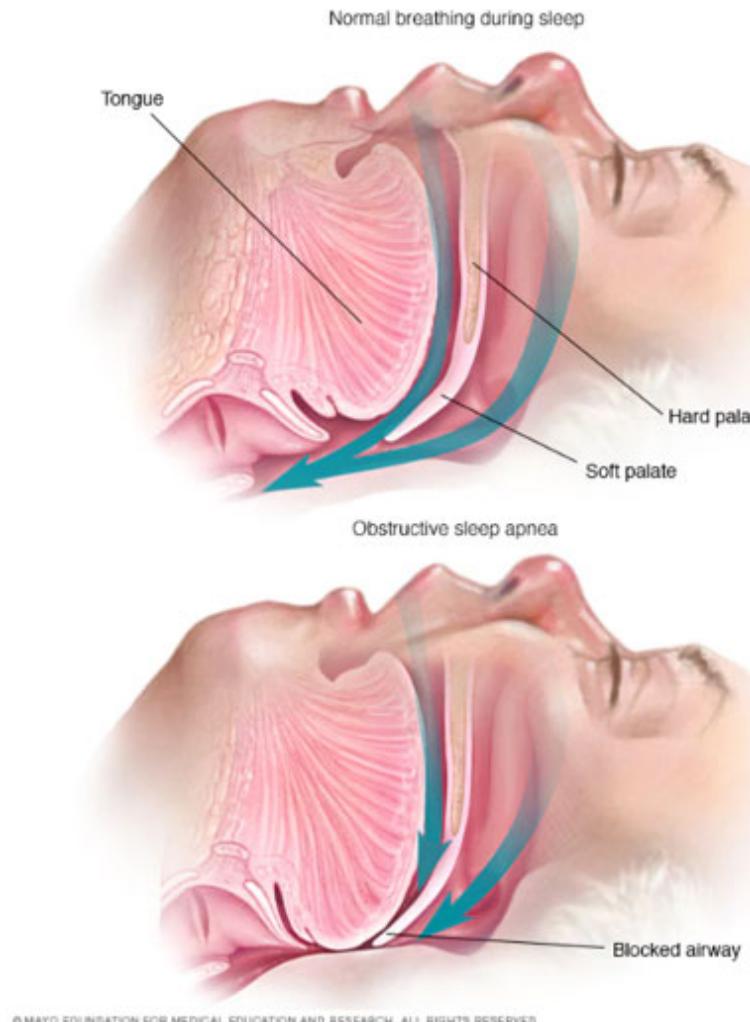


Figure 5.20: Sleep apnea is caused by blocked airways. The snoring sound and sudden stop and then gasping sound made by the patient, the condition can be monitored using AI. **Source:** <https://www.mayoclinic.org/diseases-conditions/sleep-apnea/symptoms-causes/syc-20377631#dialogId23884085>

Smart home

AI can also be used in the home. Let us explore the various scenarios.

Person detection at the door

Door cams with integrated AI can easily recognize if some person is present at the door. These can even recognize if the person is present or not. This adds added safety while answering the door. An example of such a project is available at Edge Impulse.

Figure 5.21 is showing person detection using visual AI:

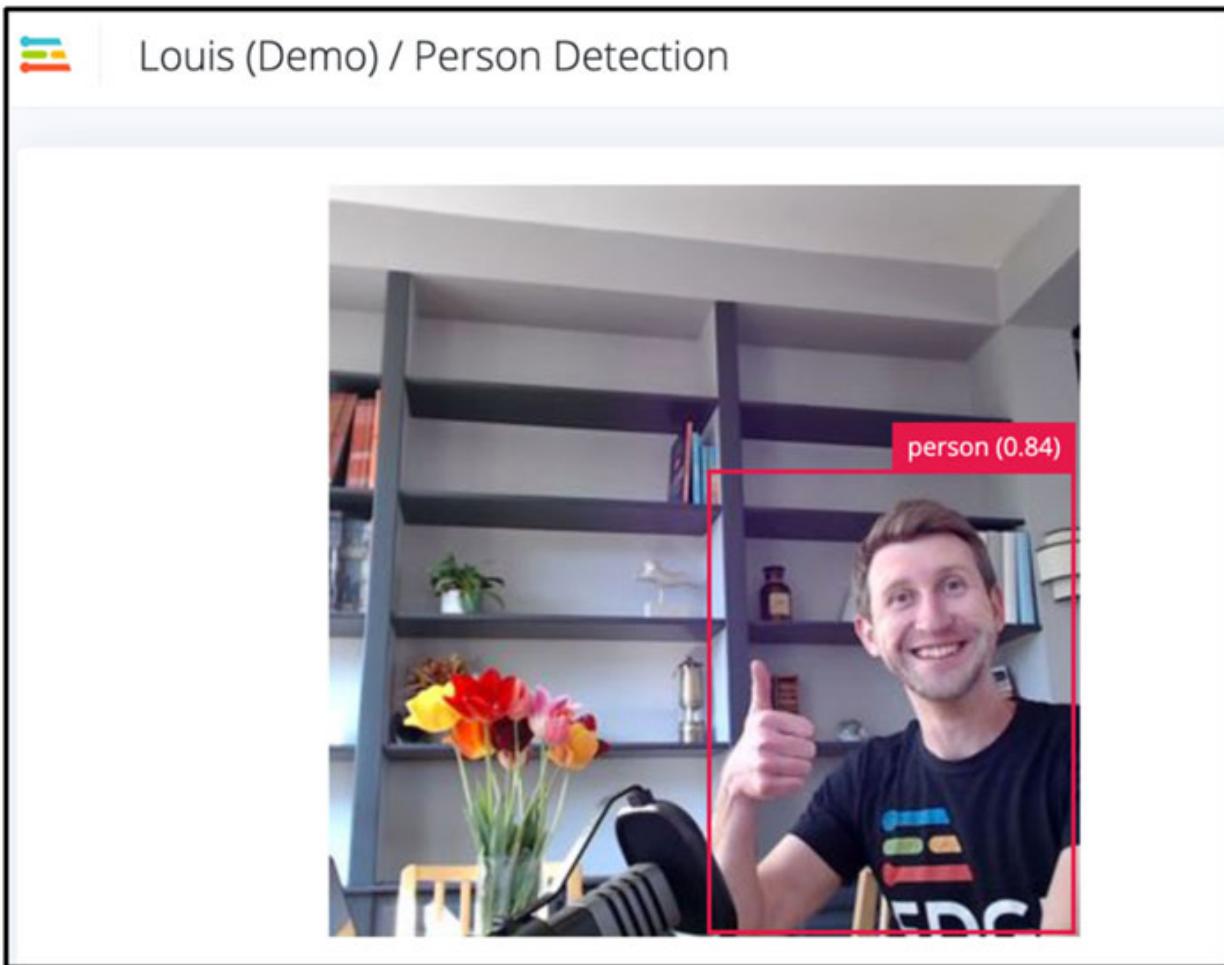


Figure 5.21: Person detection using visual AI can be useful in applications, for example, verifying who is at the door. **Source:** www.EdgeImpulse.com

Glassbreak detection

AI is being used for events when someone breaks in the home. Window glasses are the weakest point of the house and that is where intruders attack. Glass break produces very distinct sound which can be analyzed and it can be detected if someone has broken the windows.

Figure 5.22 shows a picture from teardown analysis of Ring's glassbreak detector.

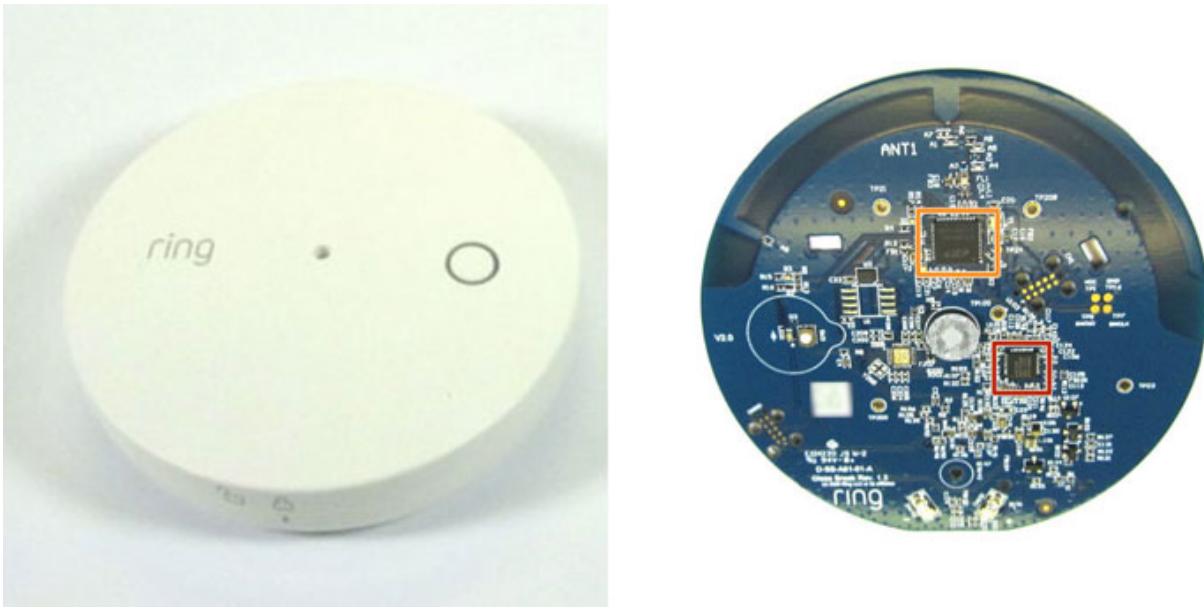


Figure 5.22: Teardown analysis of Ring's glassbreak detector shows Syntiant's AI chip used in the device. Source: <https://www.ifixit.com/Teardown/Ring+Glass+Break+Sensor+Teardown/149082>

Smart baby monitoring

Baby monitors have been available for some time but now they are getting smarter. The monitor can check certain positions which are not safe. So now, the mother does not have to continuously stare at the baby monitor. Smart AI built in the monitor will trigger an alarm. There is a company which claims to detect when the infants get hungry but before they start crying. As per their study, it is too late to feed infants when they are crying.

Figure 5.23 shows how vision AI can analyze subtle movements, for example, smacking lips, fist near mouth and so on, imply that the baby is hungry:

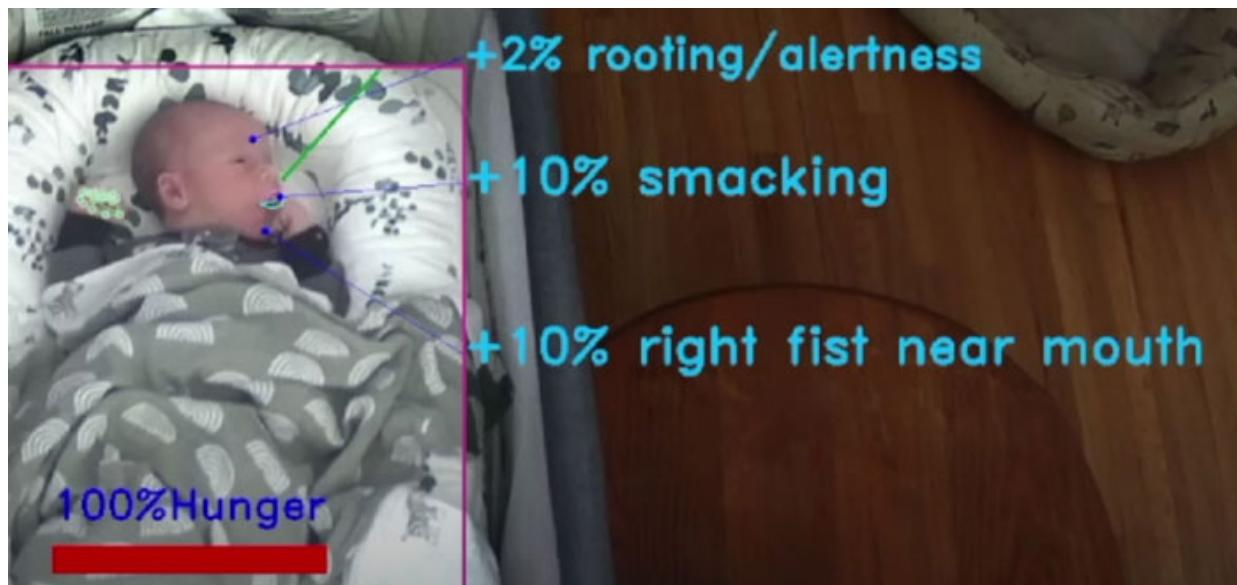


Figure 5.23: The vision AI is detecting cues that the baby is hungry before he cries. Source: <https://youtu.be/Lda1Sq8HRY4>.

Voice recognition for home automation

Josh.ai has perfected voice recognition through the home. With distributed microphones and speakers through the home, the users can interact with the house as if they are interactive with household help. Whether you want to turn on lights or drop-down window shades, all you need to do is to ask Josh for help. It can play music for you or adjust the thermostat for you.

Figure 5.24 is showing how curtains can be controlled using voice commands.



Figure 5.24: Automatic curtain opening and closing using voice commands. Source Josh.ai.

Smart industry

Industry can make use of Artificial Intelligence in a similar manner as smart homes and smart cities. Different modalities, for example, vision, audio, and sensors could be used with artificial intelligence.

Railway track defect detection

Historically, railway tracks have been inspected manually which is a very labor intensive and time-consuming process. Due to human fatigue, the quality can be compromised. AI is put to work by **Ignitarium Technology Solutions** and can check the defect at real time speed.

Figure 5.25 shows how railway tracks can be monitored in real time:

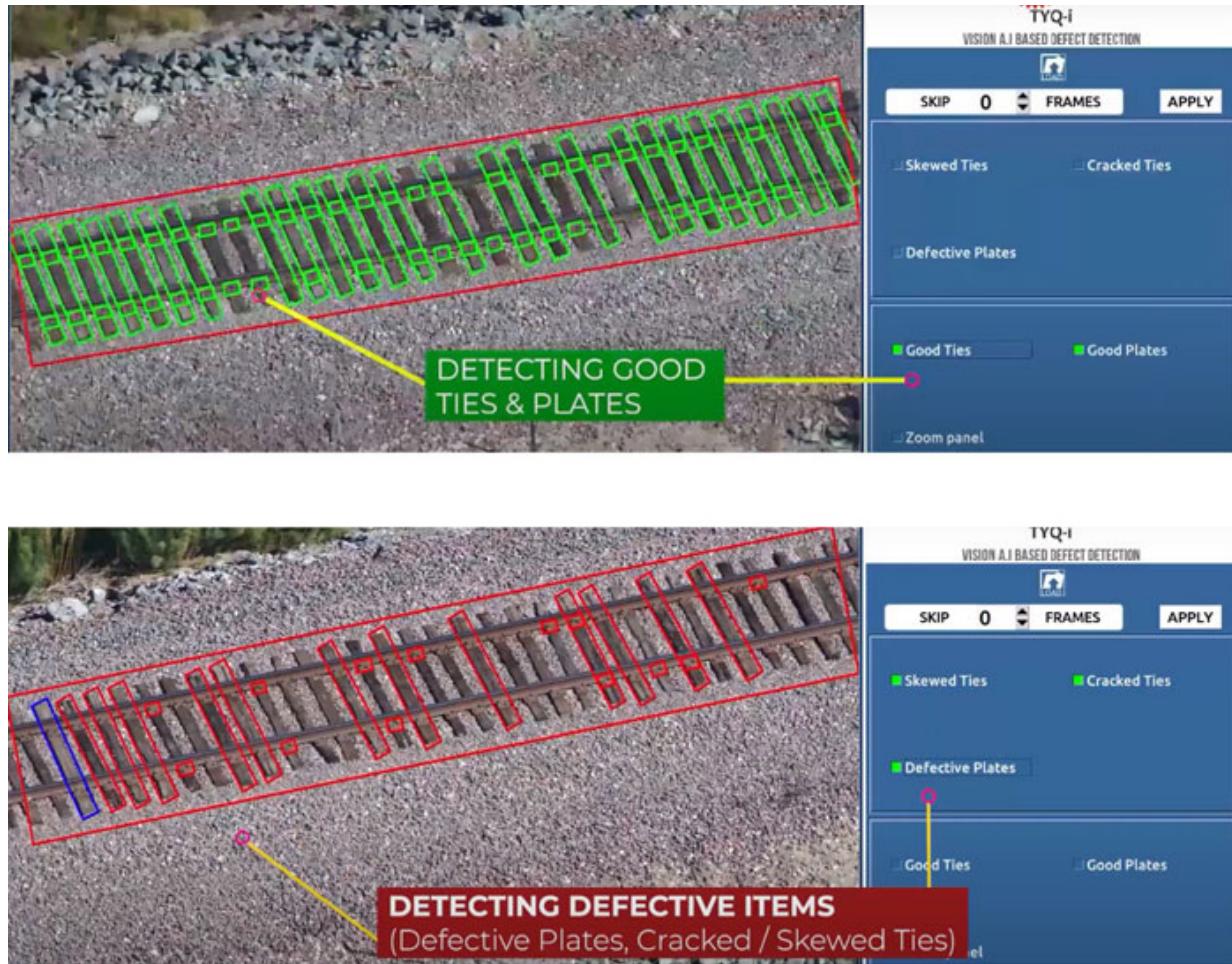


Figure 5.25: Railway track faults detection using edge AI. Source: www.ignitarium.com

Telecom towers defect detection

The defect detection technology can be extended to find faults in the building structures. In this project, the telecom towers are inspected using drones and high-resolution cameras. The low-resolution cameras can be used while the drone can scan the entire structure slowly. *Figure 5.26* shows how the towers can be inspected for structural defects:

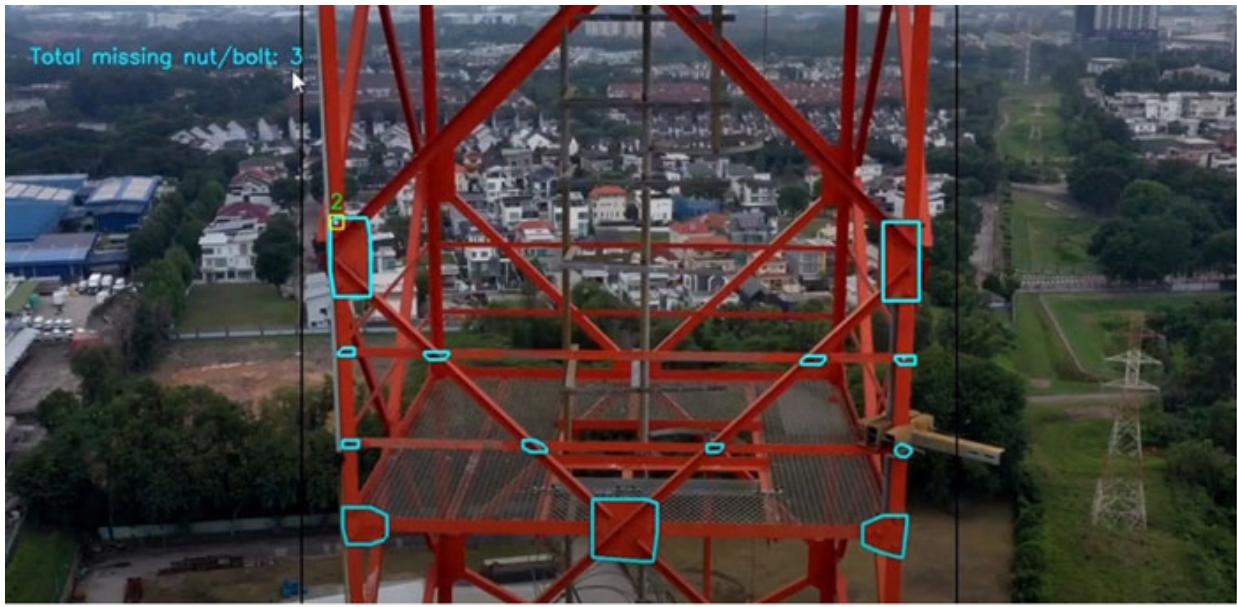


Figure 5.26: Tower inspection using AI. Source: www.ignitarium.com

Defect detection in components

The defect detection can be used for the production line. Here in *Figure 5.27*, we can see a bad washer is inspected and failed:

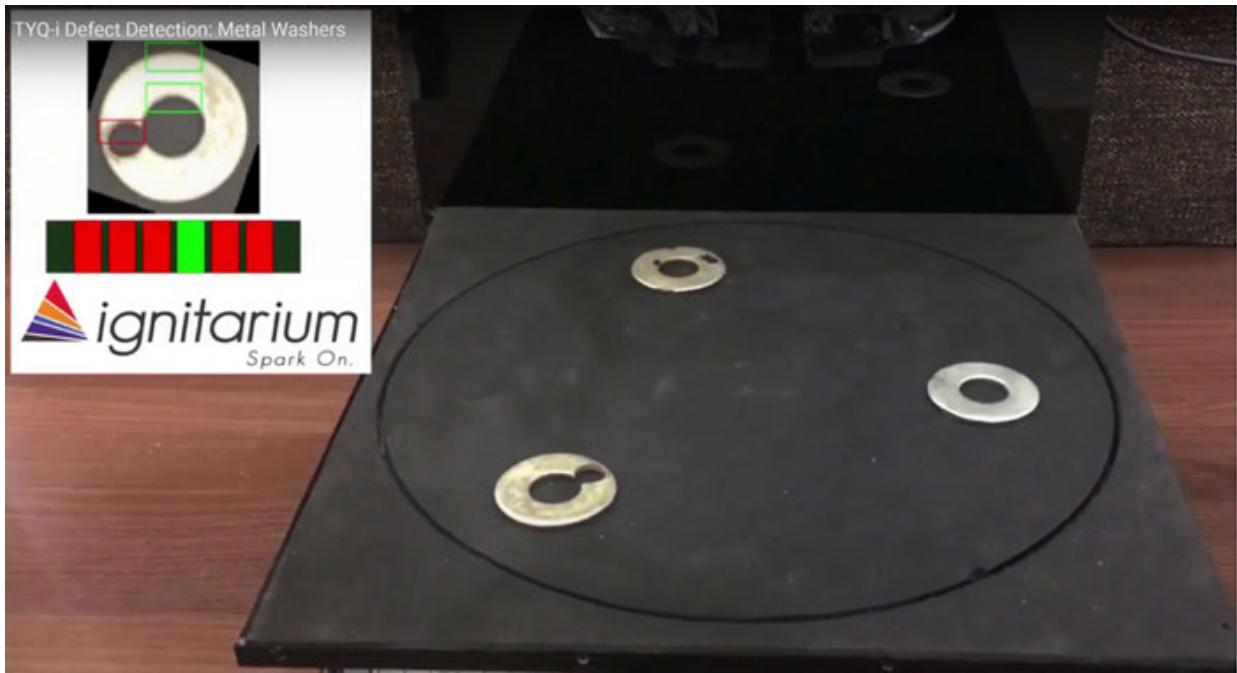


Figure 5.27: Faults in the components can be detected in the production line using vision based AI. Source: www.ignitarium.com

Smart automotive

Automotive industry is also benefiting from the advancements in the AI systems. It can be used to enhance safety and security of vehicles.

Drowsy driver alert

Often, tired drivers become drowsy without realizing. Smart detectors looking at the driver can detect it and alert the driver. Sometimes, fleet managers are also alerted, so that they can provide better training to drivers for their own safety and others.

Figure 5.28 is showing how a camera in the cabin can keep an eye on the driver:

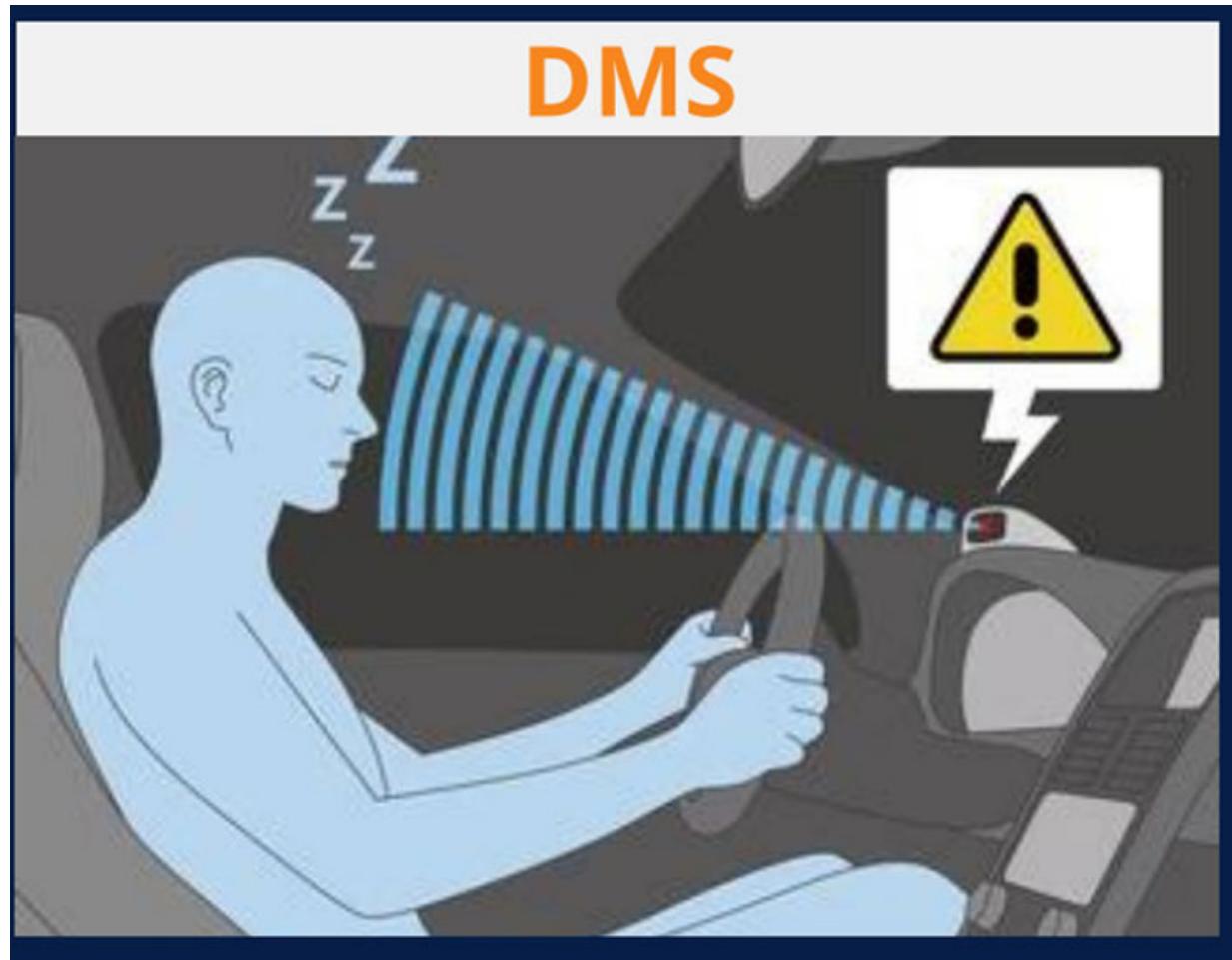


Figure 5.28: An example of a Driver management system (DMS). Source: www.ignitarium.com

Advance collision detection

By analyzing camera pictures, RADAR and/or LIDAR warnings can be generated ahead of time. Similarly, blind checks can be analyzed and the driver can be alerted when he turns on the turning signal.

Figure 5.29 depicts that RADAR and Driver management systems can warn the driver of possible accidents.



Figure 5.29: Combination of RADAR and DMS can avoid potential accidents. Source: www.ignitarium.com.

Conclusion

In this chapter, we looked at different applications of AI which have gained popularity in the last few years. The categorization over different market segments is somewhat arbitrary. The same kind of technology can be used at two different places. For example, predictive maintenance can be applied in the industry as well as household appliances. The predictive maintenance can make use of all three modalities, for example visual, audio or sensor based.

As mentioned earlier, these are just a few examples. Numerous applications are in development, making it a true nebulous market. These examples are talked about here, to spark creativity of the reader.

Key facts

- Agriculture is sometimes neglected as an industry. AI can make a big difference in improving yield and profits while reducing weedicides and pesticides.
- Household appliances have human-machine interfaces which can be improved using AI, through voice commands and gestures. The appliances will become smarter, thus providing us more time with our families.
- Smart cities started relying on artificial intelligence for maintenance, law enforcement and public safety and security.
- Smart homes can be more safe and secure with artificial intelligence-based image and voice recognition.
- Smart industries are making use of all three modalities of artificial intelligence, namely vision, audio, and sensor.
- The automotive industry started relying on artificial intelligence for safe and secure drives.

Questions

1. Estimate how many man hours can be saved if a robot can pick fruits in a 100-acre farm.
2. How are weedicides used today? How can AI reduce or eliminate the use of weedicides?
3. Brainstorm 3 new ways on how AI can be applied in agriculture, which are not mentioned in this chapter.
4. It is estimated that over 30% households have pets in the USA. Brainstorm about 3 different AI applications which will make the lives of pets more comfortable.
5. Brainstorm 3 problems of cities and a solution based on AI, which can solve the problems.
6. Think about a way which can save energy usage in a common house using AI.
7. If you were to add an AI based improvement in a car, what would that be?

References

1. Nilg: Crop monitoring: Crop monitoring & AI: The future of agriculture - NILG.AI
2. Forbes: <https://www.forbes.com/sites/louiscolumbus/2021/02/17/10-ways-ai-has-the-potential-to-improve-agriculture-in-2021/?sh=7f1f0e4f7f3b>
3. VineScout: <http://vinescout.eu/web/>
4. Herbicide:
<https://ehjournal.biomedcentral.com/articles/10.1186/s12940-016-0117-0>
5. Person Detection: Person Detection - Dashboard - Edge Impulse
6. Glassbreak
<https://www.ifixit.com/TearDown/Ring+Glass+Break+Sensor+Tear-down/149082>
7. Mosquito 1: <https://arxiv.org/abs/1705.05180>
8. Mosquito 2:
<https://sites.google.com/g.harvard.edu/tinyml/finalprojects>
9. Mosquito 3: <https://youtu.be/wtCFntSjXzs>.
10. Smart cities: 10 ways AI can be used in Smart Cities | AI Magazine
11. RoadBotics: RoadBotics - Make Data-Driven Decisions
12. Railway track 1: Ignitarium and AVerMedia Launch AI-driven Airborne Railway Track Inspection Solution | ignitarium.com
13. Railway track 2: <https://youtu.be/mpWR40CIFWM>
14. Snoring <https://youtu.be/aQpIooBEGsA>
15. Drowsy driver Alert: https://ignitarium.com/wp-content/uploads/2022/08/Infographics-800-%C3%97-3000-px.pdf?utm_source=Popup&utm_medium=Website&utm_campaign=4+WAYS+AI+IS+TRANSFORMING+DRIVER+SAFETY+AND+COMFORT
16. Defect detection Metal and Washers:
<https://youtu.be/7QP44EuegX8>
17. Hungry baby detection: <https://youtu.be/Lda1Sq8HRY4>

18. JoshAI: Josh.ai | Voice Control - Artificial Intelligence - Smart Home Automation

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 6

Practical Experiments with TinyML

Introduction

In this chapter, we will perform practical experiments with the TinyML hardware board. In general, to perform any practical TinyML experiment, we need items such as a microcontroller board, and popular boards such as Arduino Nano 33 BLE, Raspberry Pi Pico, and Nano RP2040 can be used. Once the board is available, we will require a dataset that corresponds to the sort of model we wish to train. For instance, if we wish to create a model for image classification, we will need a dataset of images with matching labels. After collecting the required dataset, a machine-learning architecture is required. TinyML frameworks such as TensorFlow Lite for Microcontrollers, Edge Impulse, and Arduino Machine Learning can be used for this purpose.

After selecting the proper framework, we need a development environment. We must configure a development environment on our computer to write and test microcontroller code. Typically, this involves installing the necessary software, drivers, and libraries for the selected board. After we have these items, we can run a TinyML experiment by following these steps such as collecting and preprocessing our data by cleaning, normalizing, and separating it into training and testing sets. With the training data and the selected framework, we will train our model. This may involve choosing an acceptable model architecture, configuring hyperparameters, and optimizing the model for optimal performance. Depending on our framework, we may need to convert the learned model to a format compatible with the microcontroller, such as a C array. We will use our development environment (for example, Arduino IDE) to write code for loading the model onto the microcontroller and processing fresh data. Depending on our application, we may need to incorporate additional sensors or hardware.

In the following sections, we will be performing air gesture digit recognition using an Arduino Nano RP2040 board, which will involve using a sensor, such as accelerometer, to detect the movement of a Nano RP2040 board in the air with the help of hand movements. This movement would then be interpreted by the Arduino Nano RP2040 microcontroller, which would then send a signal to a connected device, such as a computer or an LED, to perform a specific action.

This experimental result can be used for various applications such as controlling media player, presentation slide, home automation, and more. This experiment would require basic programming knowledge in C or Python, and experience with both the Arduino platform and sensor interfacing.

Structure

In this chapter, the following topics will be covered:

- Introduction to Nano RP2040 board
- Data collection for the air gesture digit recognition
- Model training in Edge Impulse platform
- Model testing with the collected dataset
- Model deployment in Nano RP2040 board
- Inferencing/Prediction of results with Nano RP2040 board

Objectives

In this chapter, we will focus on practical experiments with TinyML hardware board with the use of Arduino IDE. We will collect sensor data from the TinyML board, clean the data for the specified task (Air Gesture Digit Recognition), upload the data in the Edge Impulse platform, and then train and test the model with the collected data from sensors in the Nano RP2040 board. Finally, we will download the inference model from the Edge Impulse and test it on the RP2040 with the help of Arduino IDE and check the performance results.

Introduction to Nano RP2040 TinyML board

To carry out our experiment, the development board known as an Arduino Nano RP 2040 connect was utilized. The Raspberry Pi Foundation designed the Nano RP2040 development board, which is compact and inexpensive. It is based on the Raspberry Pi RP2040 microcontroller and is intended for use in a variety of applications requiring high performance and low power consumption. The RP2040 is a dual-core Arm Cortex-M0+ microcontroller with 2MB of onboard flash memory and 264KB of RAM. The Nano RP2040 board contains a USB connector for programming and power, a microSD card slot for additional storage, and support for a variety of sensors and peripherals, making it appropriate for a wide range of applications. It also has several input and output pins, such as 3.3V power pins, I2C, SPI, UART, and PWM pins, which can be used to connect a

variety of sensors and other devices. This board represents the most recent release from Arduino. Hence, this board is made up of several different peripherals, including an inbuilt microphone, accelerometer, and gyroscope.

In comparison to previous RP2040 boards, the Arduino Nano RP2040 Connect features an integrated 6-axis IMU, Bluetooth with BLE, temperature sensor and Wi-Fi in the NINA W102 u-Blox module, as well as a microphone and MicroPython support. The board consists of an on-chip LED, a power LED, and a tricolor RGB LEDS. This board has one reset button, which can be used to both reset the board as well as enter the boot loader mode. This button is provided for our convenience. The board is powered by using the USB. It has sensors for both acceleration and rotation on it. In addition to that, the chip has a temperature sensor and a microphone on it, so that it can be used in studies involving audio. The 6-axis IMU and NINA W102 module distinguish the Arduino Nano RP2040 Connect from other development boards. It is essentially the offspring of the Arduino Nano BLE Sense and Raspberry Pi Pico W, making it the perfect board for TinyML development.

The board is designed to be user-friendly and supports a range of programming languages, including C/C++, Python, and other popular languages. It can be programmed with Arduino IDE, the CircuitPython development environment, and the MicroPython programming language, among others. This makes it a flexible and robust platform for a wide range of tasks, including robotics and automation, IoT applications, and more.

Setting up Arduino IDE and testing the Nano RP2040 Board

To carry out the experiment, we need to make the necessary configurations in the Arduino IDE for the board, which is a Nano RP2040. To begin with, we will need to navigate to the **Tools** menu in the IDE and then select the **Board Management** option. Search for Nano RP2040 so that it will come up in the results. Be sure that we are installing the appropriate packages for this specific board, and not the ones for any other boards. To determine whether the board is operational, we may select some sample programs that are contained within the IDE and uploading them to the board by executing examples like blink.

Figure 6.1 features the Nano RP2040 board connected with a computer:

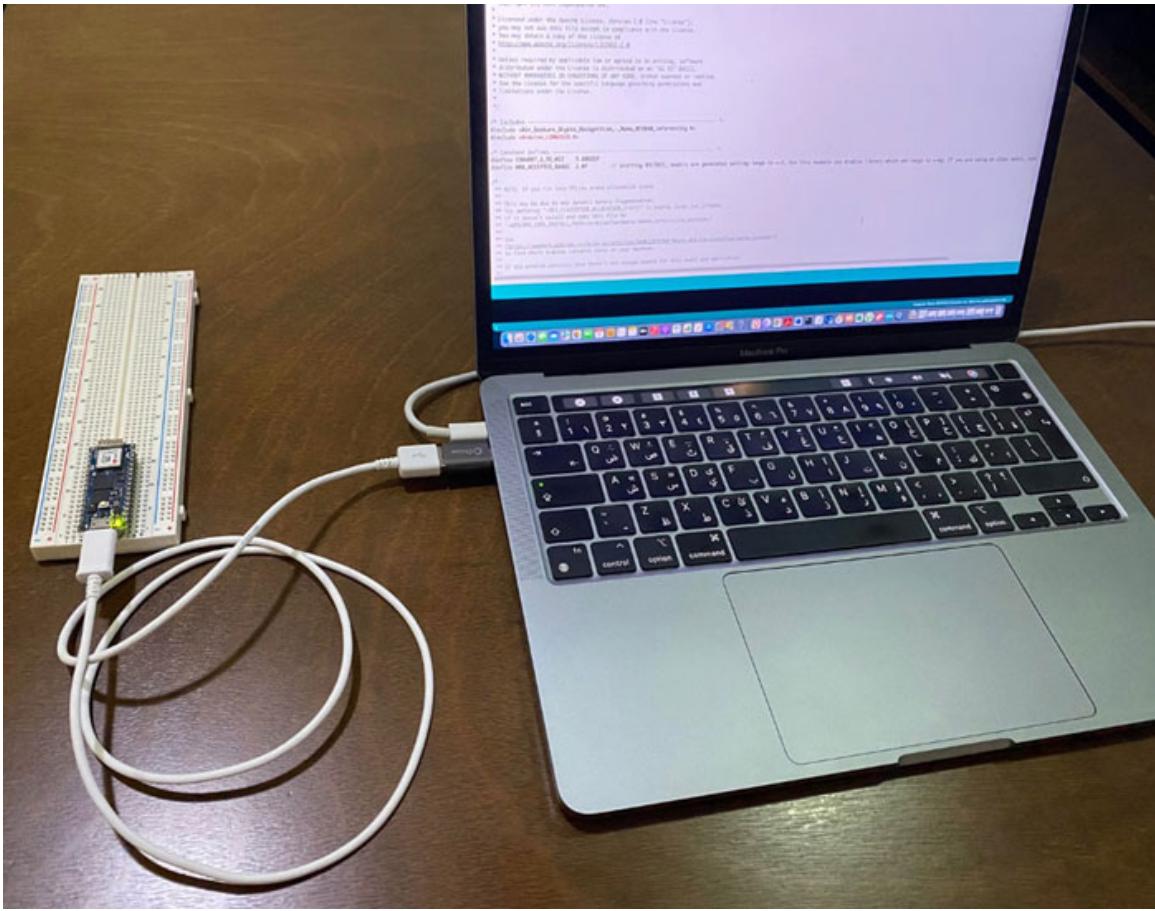


Figure 6.1: RP Nano 2040 board connected with computer

Testing on-board accelerometer and gyroscope

After successfully configuring our board, we must now determine whether all the peripherals included on the board are operational. For example, we may try an example or simply test the accelerometer. We can access examples through the IDE tool, and this example will appear after the required library has been installed. To successfully install the library, we must first navigate to the IDE's **tool** menu and then select the option to manage the library. After that, we will do a search through the library manager. We can select for LSM6DSOX-related modules that need to be installed. There are multiple versions of the modules available and we must choose the correct one.

After installing the required library, all that remains is for us to take the example available under LSM6DSOX module, open the accelerometer, and build it to see if the accelerometer is working properly. When the compilation is complete, we can see that the USB disk opens and closes, and the application begins to operate. We can see the data displayed by opening the serial monitor on our PC. If we are

having trouble reading the LSM6DSOX IMU values, make sure you have the most recent NINA Firmware installed.

[Figure 6.2](#) features the accelerometer data collection and display using the serial monitor:

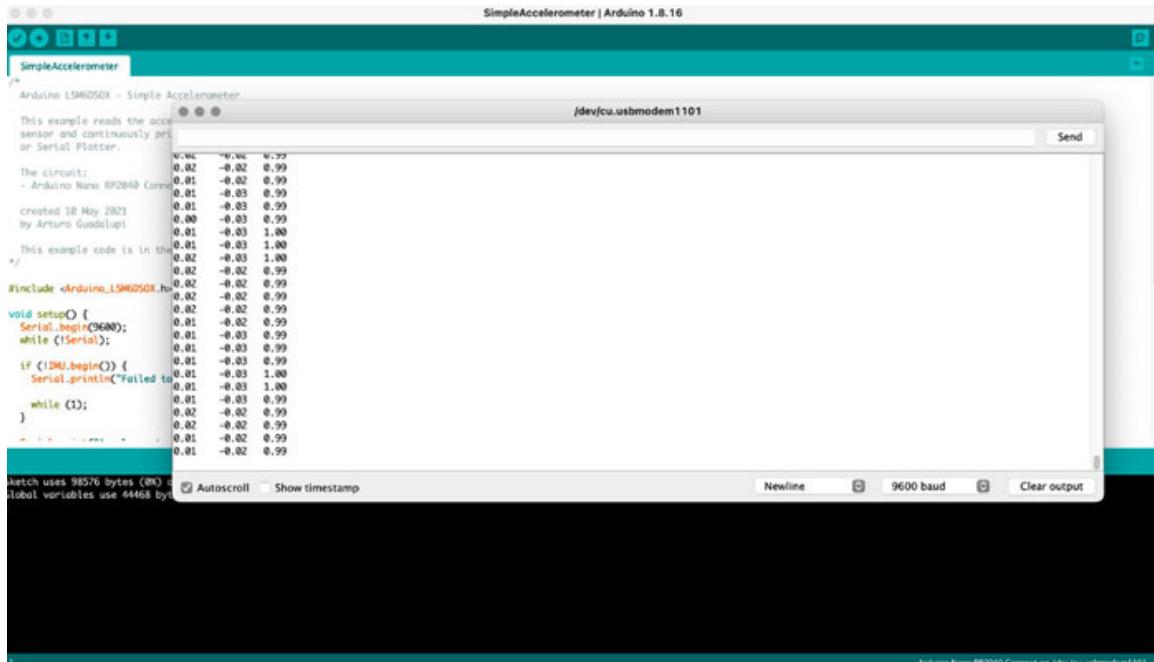


Figure 6.2: Accelerometer data collection and display using serial monitor with Nano RP2040 board

High level steps involved in the air gesture digit recognition in Edge Impulse platform

To deploy our ML model using the Edge Impulse platform, we must follow important steps such as data acquisition, impulse design, EON tuner and deployment.

Here is the step-by-step procedure for developing a model for air gesture digit detection from zero to nine, on the Edge Impulse platform utilizing an RP2040 board.

1. Initially, we will perform hardware setup, which will involve connecting our RP2040 board to our computer through a USB cable. Make sure our board is turned on and ready to go.
2. The next step is to log into Edge Impulse and create a new project. Give it a suitable name and select "**Air Gesture Digits Recognition - Nano RP2040**" as the project name. Users can select their own name.

3. Edge Impulse supports several data import methods, such as uploading a CSV file, connecting to a data collection device, and importing from an existing data source.
4. After importing the data, each data point must be labeled with the proper label for each gesture. In our use case, the labels range from "zero" to "nine".
5. The next stage is pre-processing; depending on the data quality, some pre-processing may be required. For example, we may need to add filters, remove noise, or normalize the data provided.
6. Based on the data we have, Edge Impulse provides several feature engineering possibilities. We can select from several attributes, including MFCC and spectrogram.
7. The next phase is model training, which comes after feature engineering. Edge Impulse has several model training methodologies available, including Deep Learning models.
8. We can select the algorithm to be used for training, as well as the parameter and complete the training and testing. Following that, we will choose the deployment using the target board and generate the code to be tested on the final board, the Nano RP2040.

Data collection for the air gesture digit recognition

The air gesture digit recognition experiment recognizes and translates airborne hand movements into numerical digits. For this, an Arduino Nano RP2040 microcontroller board is used. To build a successful air gesture digit recognition model, high-quality data must be collected. The steps involved in acquiring data for this experiment are as follows:

1. **Define the gestures:** The first step in the air gesture digit recognition experiment is to specify which gestures will be used. For our experiment, we will use ten digits, "ZERO" to "NINE", for recognition.
2. **Create the data collection environment:** The setting in which the data is collected, might have an impact on its quality. It is vital to ensure that the background noise is kept to a minimum, and the area is spacious enough for the data collection user to move around.
3. **Data collection:** We will utilize the RP2040 board accelerometer sensor to record the gestures. Before collecting the data, we must first determine whether the sensor is operational.

4. **Label the data:** After recording the data, it must be labeled. This requires identifying gestures in the data and providing a numerical value to each gesture.
5. **Clean the data:** Data cleaning is required to remove any errors or inconsistencies. This could include deleting outliers, flattening the data, or correcting data points that were erroneously classified.
6. **Split the data:** After cleaning the data, it must be divided into training, validation, and testing sets. The training set will be used to train the model, the validation set will be used to fine-tune the model's hyperparameters, and the testing set will be used to evaluate the model's performance. In our experiment using Edge Impulse, we will be splitting the data into training and test sets only.

So, let us start our air gesture digit recognition data gathering. To accomplish this, the first step is to acquire some data from the accelerometer. First, include the header with the proper LSM module. Then, configure the serial monitor. Finally, check to see if the accelerometer sensor is present. We are going to store the data in a CSV file so that the time and acceleration parameters can be appropriately recognized by the Edge impulse. These parameters will be kept in the three columns and will be labeled Accx, Accy, and Accz respectively.

If we open the serial monitor, we will be able to see the timestamp as well as the data from the accelerometer that has been printed. The millisecond timestamp is displayed in the first column, followed by the Accx value in the second column, the Accy value in the third column, and the Accz value in the fourth column. To record the data, we are going to record it for a specific digit gesture. To do so, we are going to rotate our accelerometer for at least two to three minutes. For the experiment, we gathered data for a total of four minutes during the training phase and one minute during the testing phase. We followed 80:20 rule for the dataset collection, which is 80% training data and 20% testing data. The more data we collect, the better the machine learning model can be trained, and the better the results will be.

Before we can begin collecting the data, we need to make sure that the reset button on our development board, that is, Nano RP2040, is pressed. After that, to capture the data, we will need to move the board into the gesture orientation. After compiling the data in notepad or any other text editor, we will save it as a .csv file. This step will be repeated for each individual digit action. The data set is already in the format of commas separated by spaces, and the first line is a timestamp. The acceleration parameters are listed in the following three columns.

Figure 6.3 features the air gestures digit data collection with the RP2040 board:

```

Collecting_dataset_Gestures
// CODE TO COLLECT DATASET FOR DIFFERENT AIR DIGIT RECOGNITION
#include <Arduino_LSM6DSOX.h>
#include <Adafruit_L3GD20.h>
unsigned int i=0;
void setup() {
  Serial.begin(9600);
  while (!Serial);
}
void loop() {
  if (L3D.begin()) {
    Serial.println(" Board ID");
    L3D.print("Board ID");
    while(i<3);
    // Print the dataset header
    L3D.print("Timestamp ");
    L3D.print("Ax ");
    L3D.print("Ay ");
    L3D.print("Az ");
    L3D.print("Gx ");
    L3D.print("Gy ");
    L3D.print("Gz ");
    L3D.print("AccelX ");
    L3D.print("AccelY ");
    L3D.print("AccelZ ");
    L3D.print("Temp ");
    L3D.println("°C");
    Serial.println();
    Serial.println("Data");
    L3D.println("Timestamp Ax Ay Az Gx Gy Gz AccelX AccelY AccelZ Temp °C");
    L3D.println("16680,0,017,-0,027,0,094");
    L3D.println("16520,0,016,-0,023,0,093");
    L3D.println("16160,0,015,-0,027,0,095");
    L3D.println("16200,0,007,-0,026,0,093");
    L3D.println("16240,0,010,-0,031,0,094");
    L3D.println("16280,0,013,-0,018,0,092");
    L3D.println("16320,0,007,-0,035,0,093");
    L3D.println("16360,0,015,-0,023,0,094");
    L3D.println("16440,0,010,-0,022,0,093");
    L3D.println("16480,0,013,-0,026,0,093");
    L3D.println("16520,0,006,-0,028,0,094");
    L3D.println("16560,0,016,-0,023,0,094");
    L3D.println("16600,0,005,-0,030,0,094");
    L3D.println("16640,0,014,-0,024,0,093");
    L3D.println("16680,0,004,-0,031,0,093");
    L3D.println("16720,0,012,-0,022,0,093");
    L3D.println("16760,0,013,-0,029,0,093");
    // READ THE DATA FROM A:
    L3D.println("16800,0,008,-0,024,0,093");
    L3D.println("16840,0,013,-0,026,0,094");
    L3D.readAcceleration(&Ax,&Ay,&Az);
    L3D.readGyro(&Gx,&Gy,&Gz);
    L3D.readAccel(&AccelX,&AccelY,&AccelZ);
    L3D.readTemperature(&Temp);
    L3D.println();
    L3D.println("17080,0,010,-0,026,0,093");
    L3D.println("17040,0,010,-0,025,0,093");
    L3D.println("17080,0,009,-0,028,0,093");
    L3D.println("17120,0,011,-0,024,0,093");
    L3D.println("17160,0,013,-0,029,0,094");
    L3D.println("17200,0,014,-0,025,0,093");
    L3D.println("17240,0,016,-0,027,0,094");
    L3D.println("17280,0,005,-0,027,0,093");
    L3D.println("17320,0,016,-0,026,0,094");
  }
  Sketch uses 58534 bytes (0%) of 644472 bytes
  Global variables use 44472 bytes (0%) of 644472 bytes
}

```

Figure 6.3: Air gestures digit data collection with RP 2040 board

Finally, data collection is an important stage in developing an air gesture digit recognition model using the Nano RP2040 board. It is critical to properly describe the gestures, create an appropriate setting for data collecting, accurately record the data, label, and clean the data, and divide it into training, and testing sets. We can build a robust and accurate machine learning model for air gesture digit identification using high-quality data.

Figure 6.4 features Project creation in the Edge Impulse Platform:

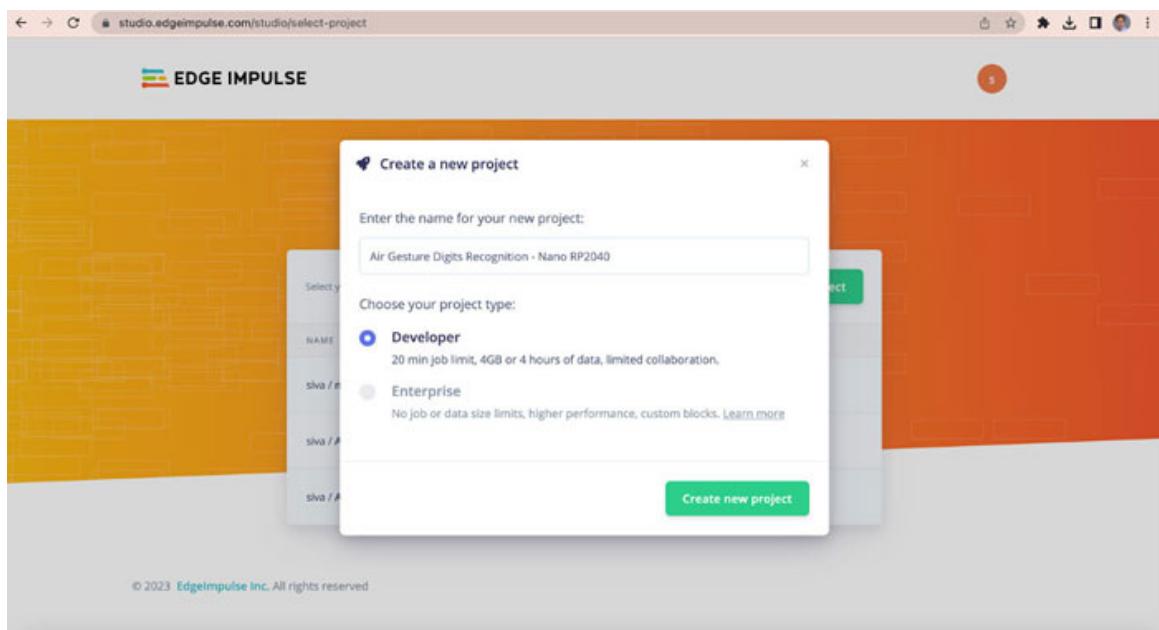


Figure 6.4: Project creation in the Edge Impulse Platform

Figure 6.5 features the available data collection options with different sensors in the Edge Impulse Platform:

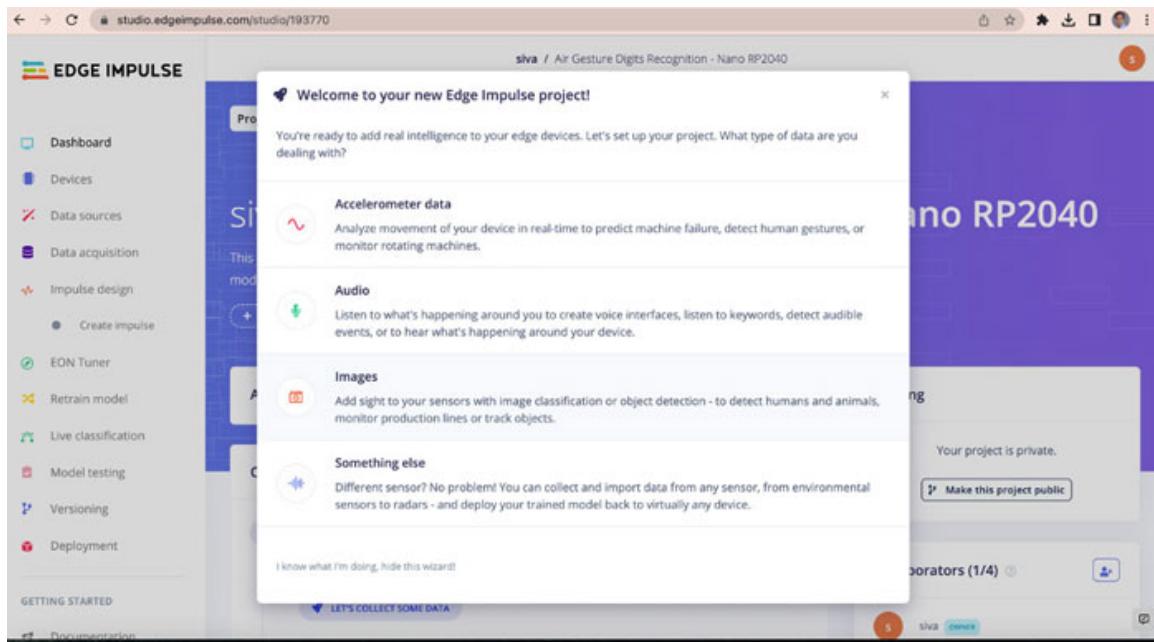
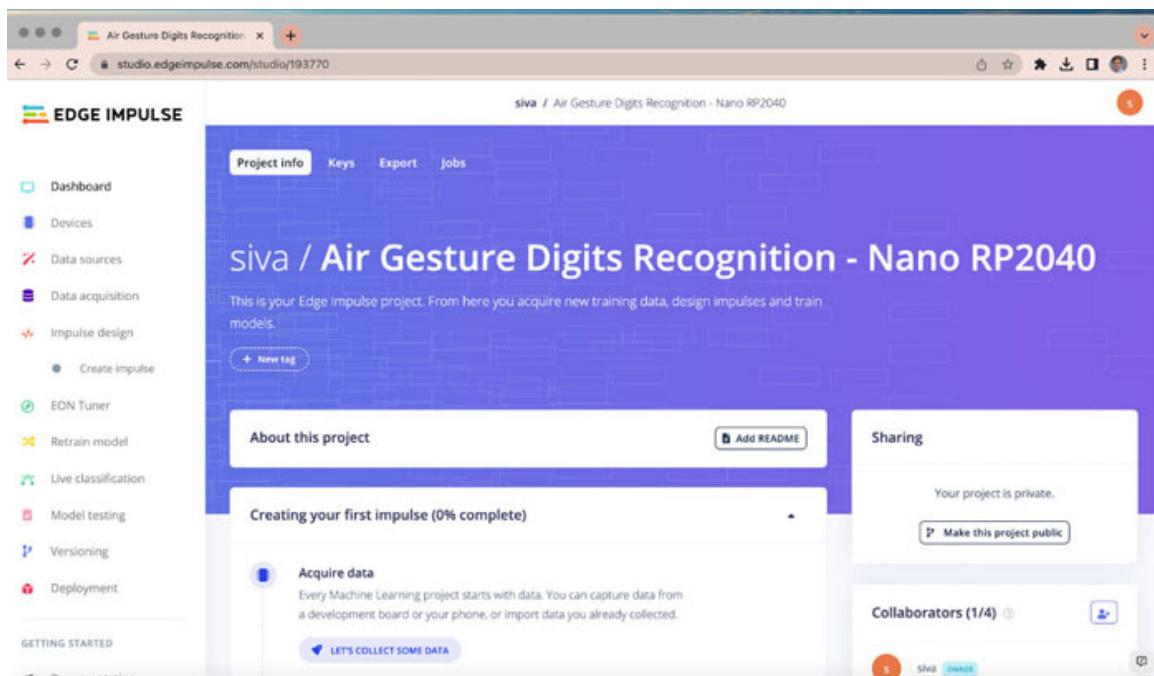


Figure 6.5: Available data collection options with different sensors in the Edge Impulse Platform

Figure 6.6 (a) and **(b)**, features data acquisition and building dataset in the Edge Impulse Platform:



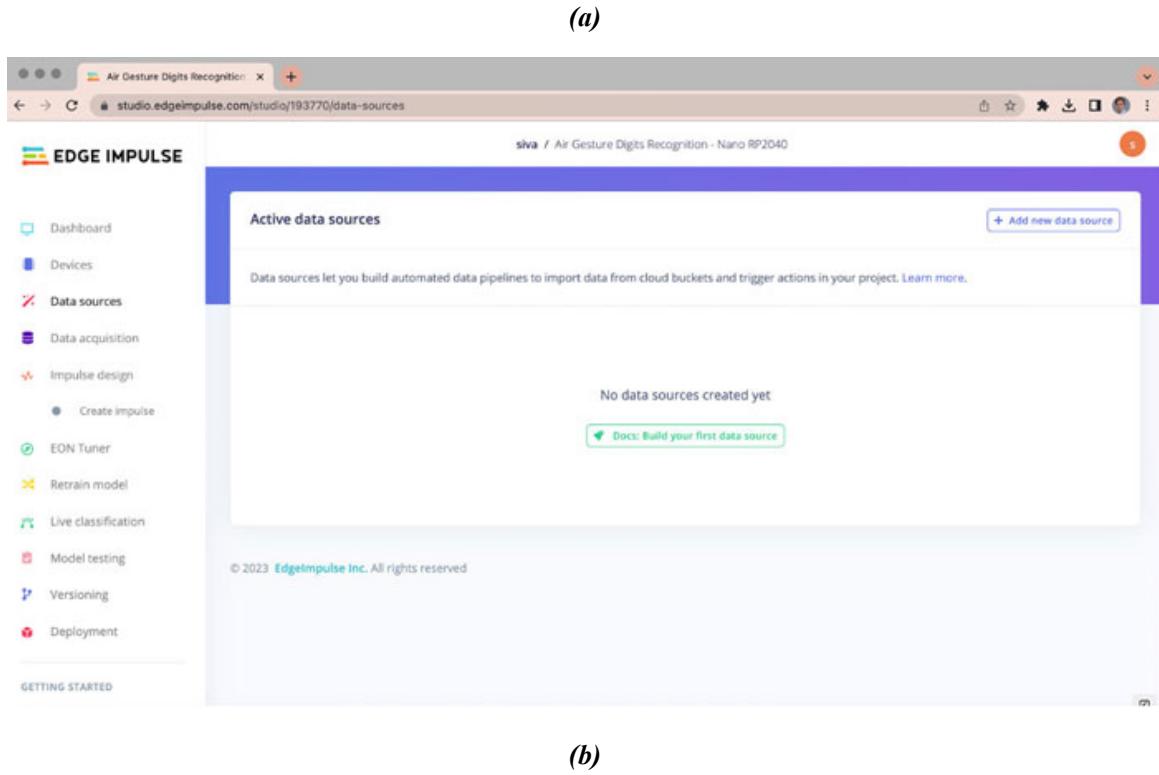


Figure 6.6 (a) and (b): Data acquisition and building dataset in the Edge Impulse Platform

Loading the dataset in Edge Impulse Platform

The first step after attaching the RP Nano 2040 board to the Arduino IDE is data acquisition. We must upload the data. Therefore, under the Data Acquisition tab, we just go to upload data, and here we can upload data in any of the accepted file formats, such as json, csv, jpeg and so on.

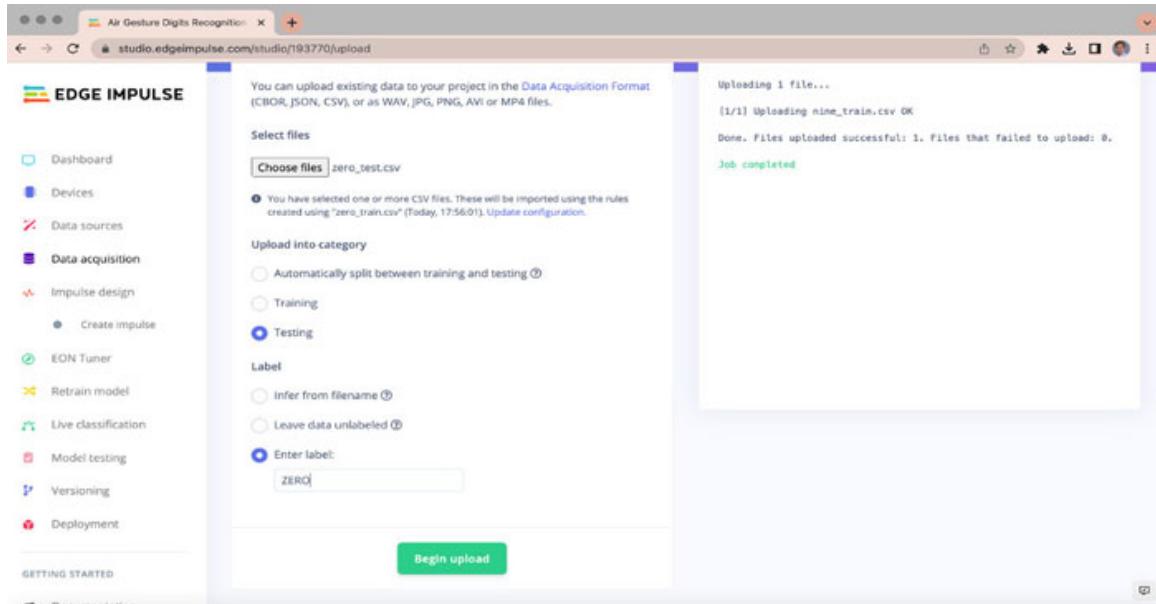
We must understand that when we upload time series data, such as accelerometer data with three axes x, y, and z, we must prepare it to ensure that there are no missing values and that it is properly aligned. Hence, while creating the data, whether from a cell phone or our development board, we must collect the data in the right format.

When we choose the file, for example, the data gesture of zero, we can choose whether the data is for training or testing. In general, for every machine learning application, training must come first, followed by testing. As a result, the data in the training data set will not be used in testing. The final testing will be done on an entirely new data set.

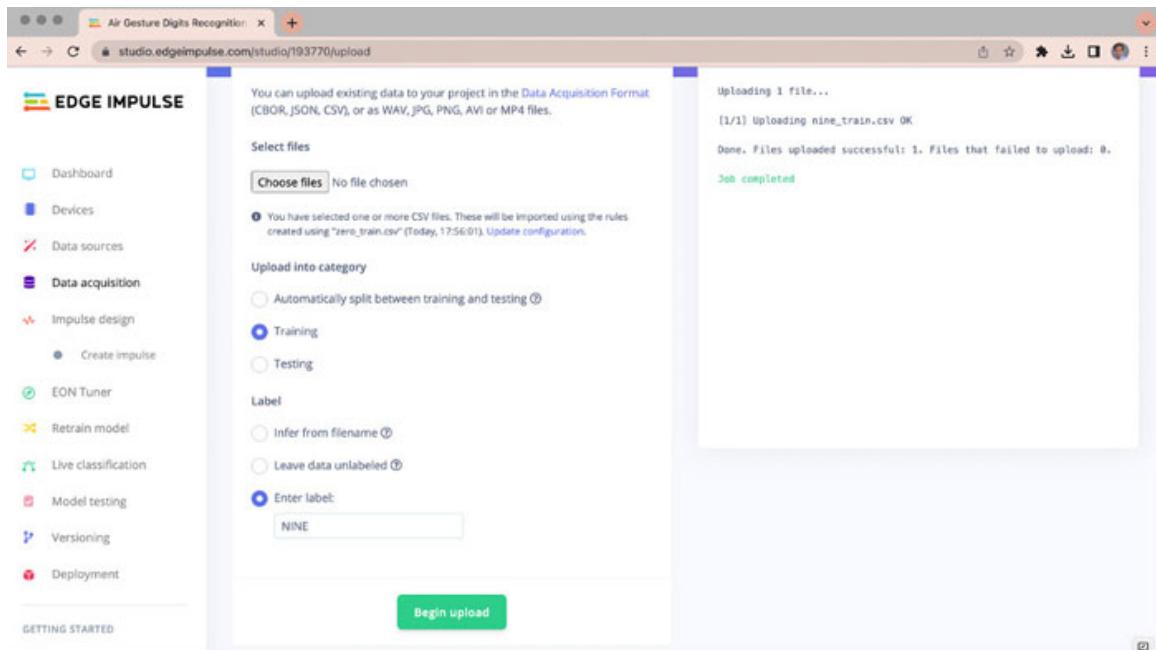
After uploading the dataset to the Edge Impulse Platform, the tool will use it as a training data set by default, as we are undertaking a classification problem and we have ten varieties of gestures starting from "ZERO" to "NINE". If there are any

formatting errors in the data file, the tool will display an error stating the file could not be uploaded. We can manually select the individual data columns and inform the tool about the labels we have in the dataset .csv file. We can upload all the digits gesture data files. After uploading, we can also visualize the training and testing data. When we submit the data, we must specify the label.

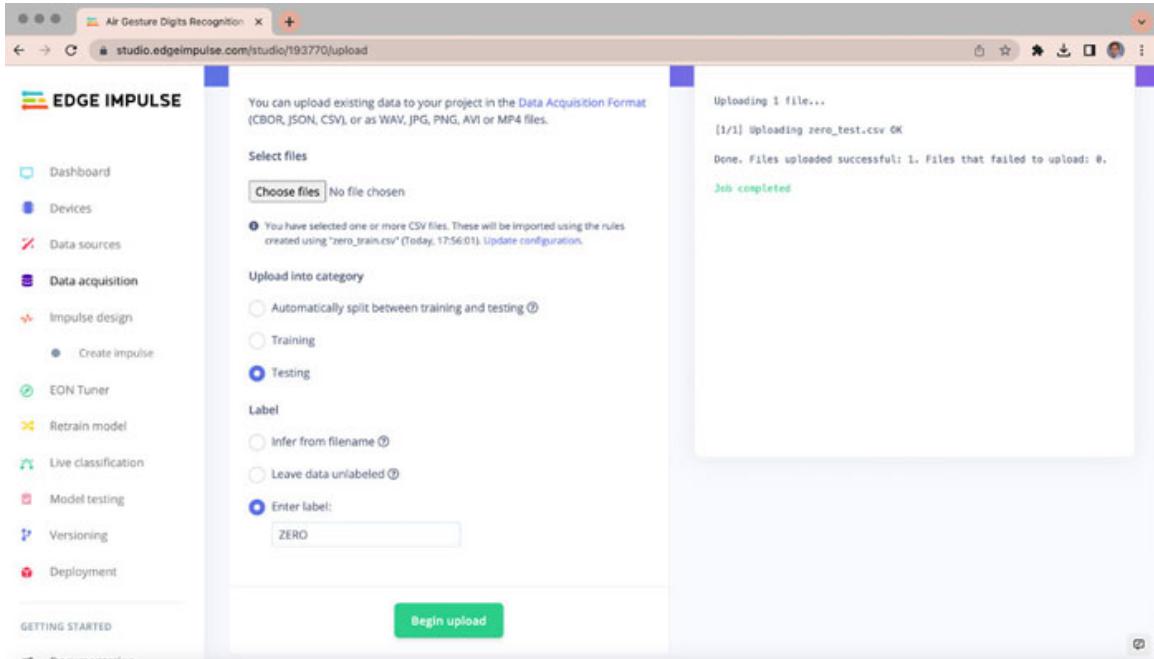
Figure 6.7 (a) to (d), features the entire process of training and testing dataset uploading with label names in the Edge Impulse platform:



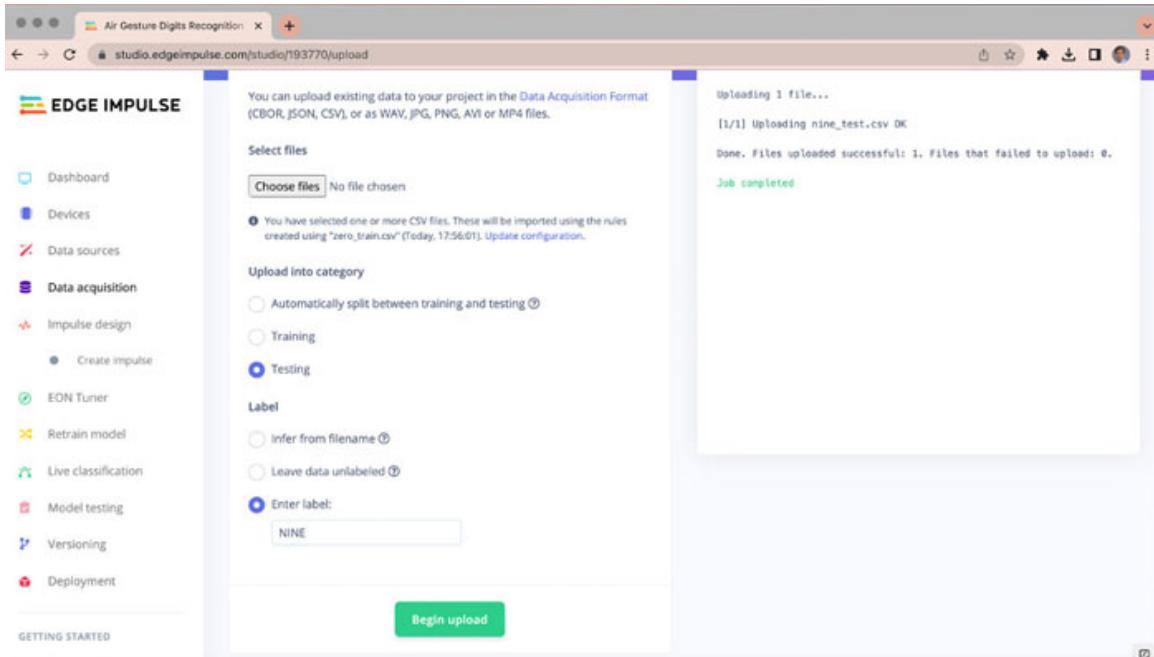
(a)



(b)



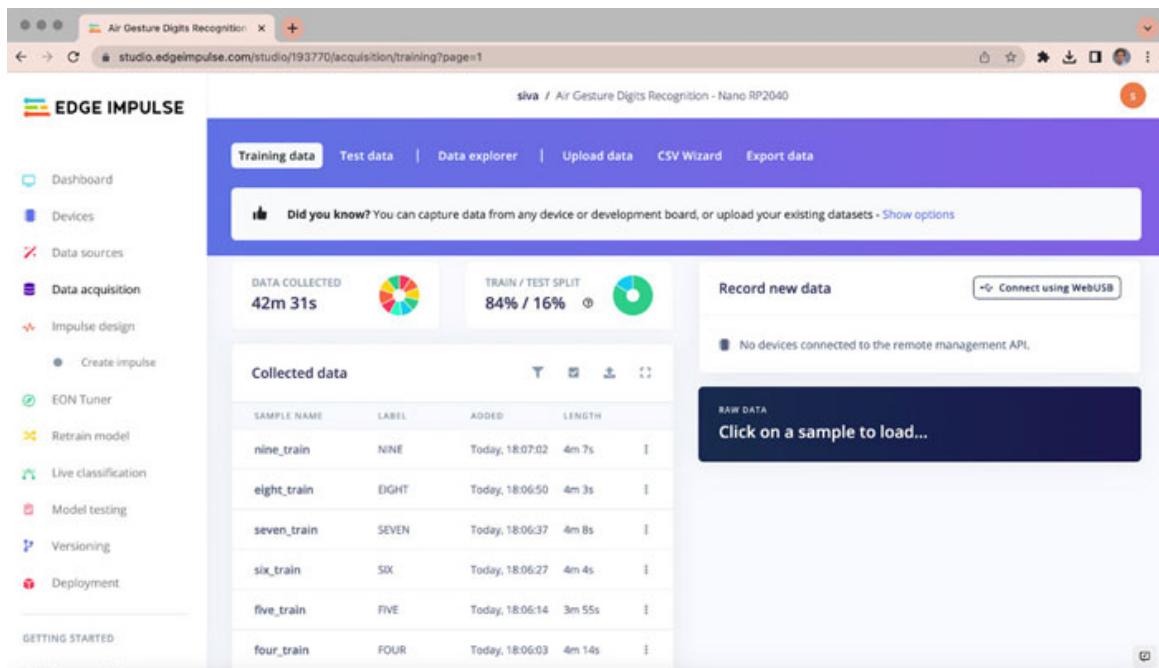
(c)



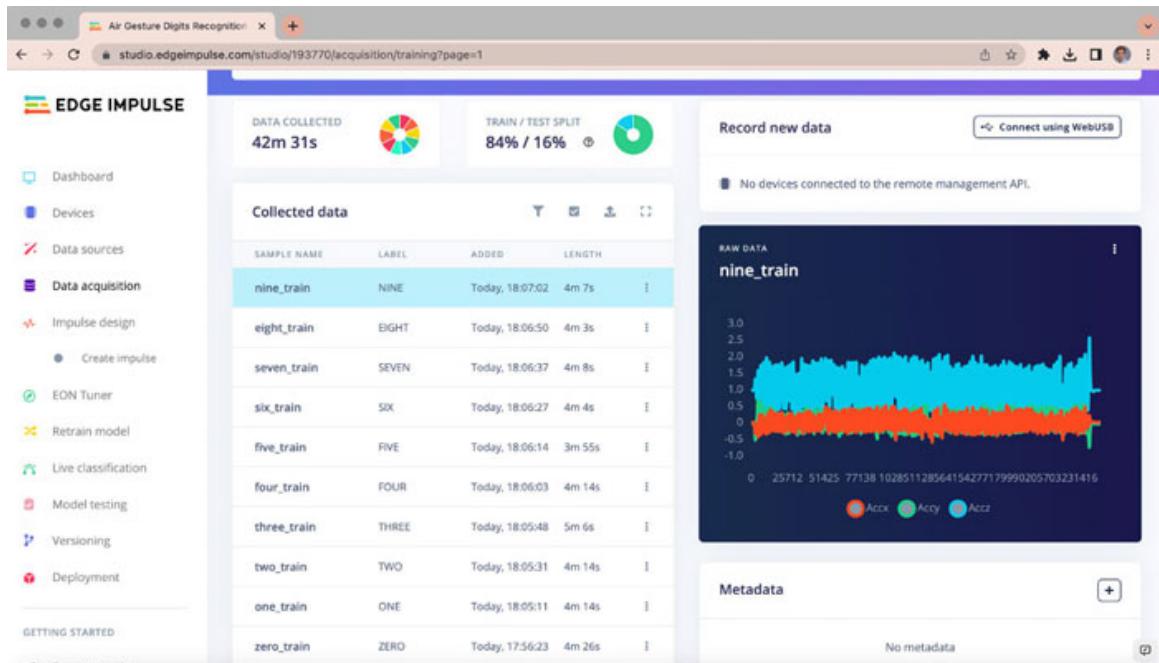
(d)

Figure 6.7 (a) to (d): Training and testing dataset uploading with label names in the Edge Impulse platform

Figure 6.8 (a) and (b), features the final uploaded datasets with their visualizations in the Edge Impulse platform:



(a)



(b)

Figure 6.8 (a) and (b): Final uploaded datasets with their visualizations in the Edge Impulse platform

Setting up the development framework and design of neural network classifier

A machine learning framework is required to begin our work on any TinyML development-based project. Therefore, the framework basically accomplishes all the necessary tasks, starting from data collection, data processing, model training, model creation and eventually deployment.

In general, we have a 20-minute job limit after signing up for the Edge Impulse platform as a developer. We may require an enterprise license and a total data storage capacity of four **gigabytes (GB)** or four hours of data is available.

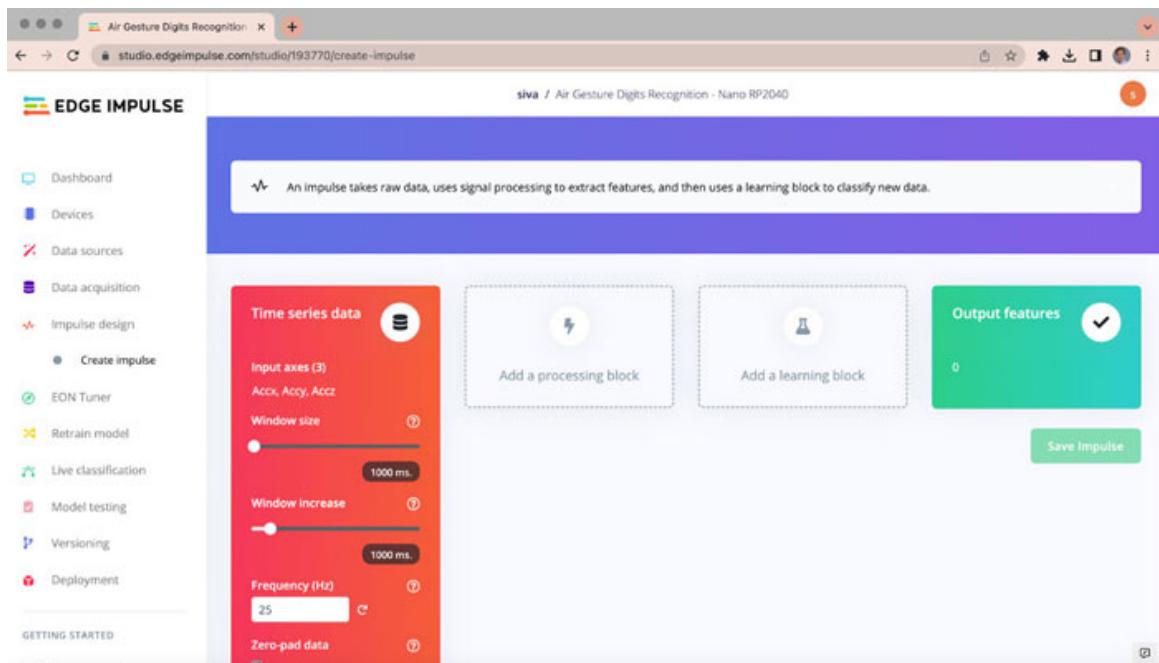
To begin with, create a new project, and then we will see one dialog box that will show us how to collect data. In general, we acquired the accelerometer data from the microcontroller, which is found on development boards. If we are working on speech recognition, image classification, sentiment prediction or keyword spotting in projects, the data may be in the form of audio, text, or images. As a result, we can acquire data from additional sensors, such as ultrasonic sensors or any other form of sensor. In this use case, we will be collecting accelerometer data.

For gesture recognition, the accelerometer data needs to be acquired first, for a specific gesture. There are now various options for loading data into Edge Impulse Studio. If our development board supports it, we can load the data straight from it, or we can upload it via the Edge Impulse CLI, which is essentially a command line interface. Alternatively, we can upload the data with offline.

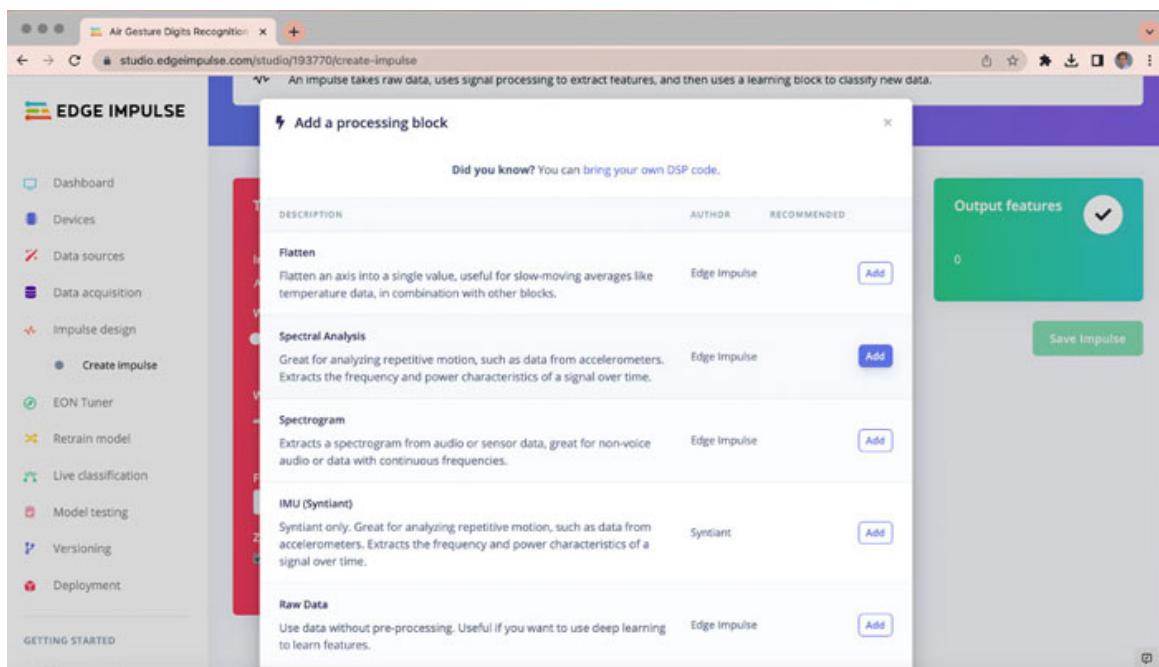
We can save the data in a csv file format, and then we can upload the csv file, and the same is true for any other data, such as images or audio. First, basic processing is performed, followed by the Neural network block. Because this is a time series data, we only need to define window size information.

Spectral analysis is important after adding a processing block for this time series data. We can include a processing block for spectral analysis. Spectral analysis converts the input time series data into the frequency domain. The neural network block is essentially the learning block. We intend to classify everything into ten levels. It is a KERAS classification. After selecting all parameters, we can save the **impulse**. The features will then be generated under the spectral feature. After we have chosen all the parameters and neural layers, we can begin training and then test it.

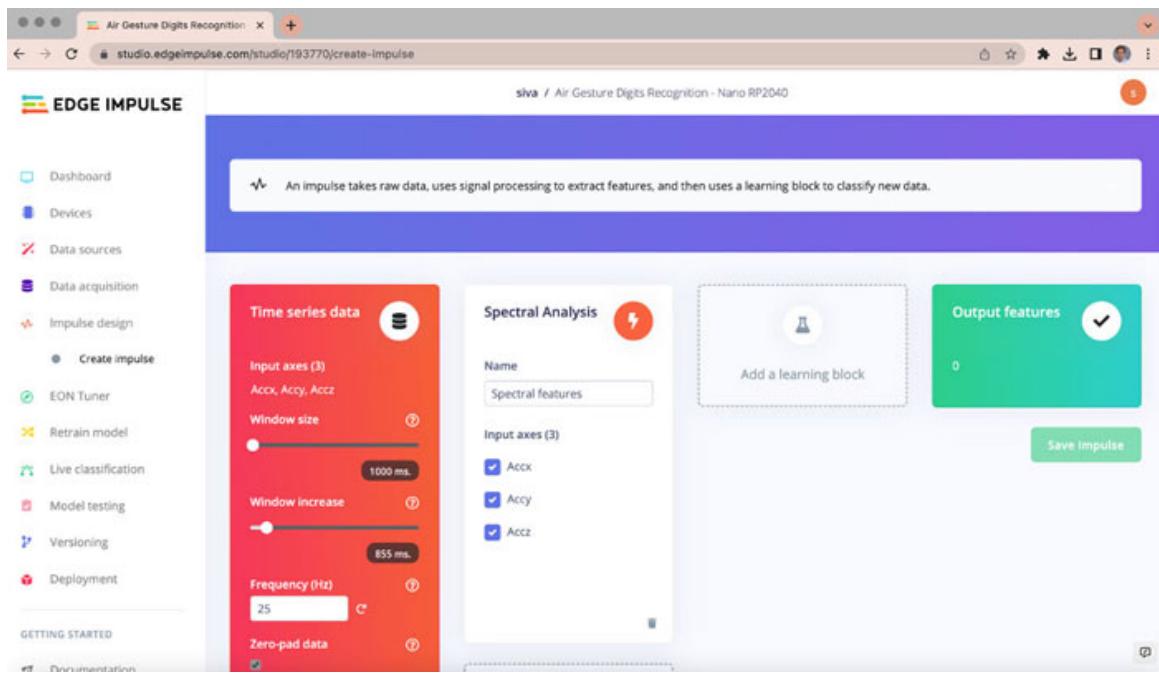
Figure 6.9 (a) to (d) features the addition of different processing blocks for the model training in the Edge Impulse platform:



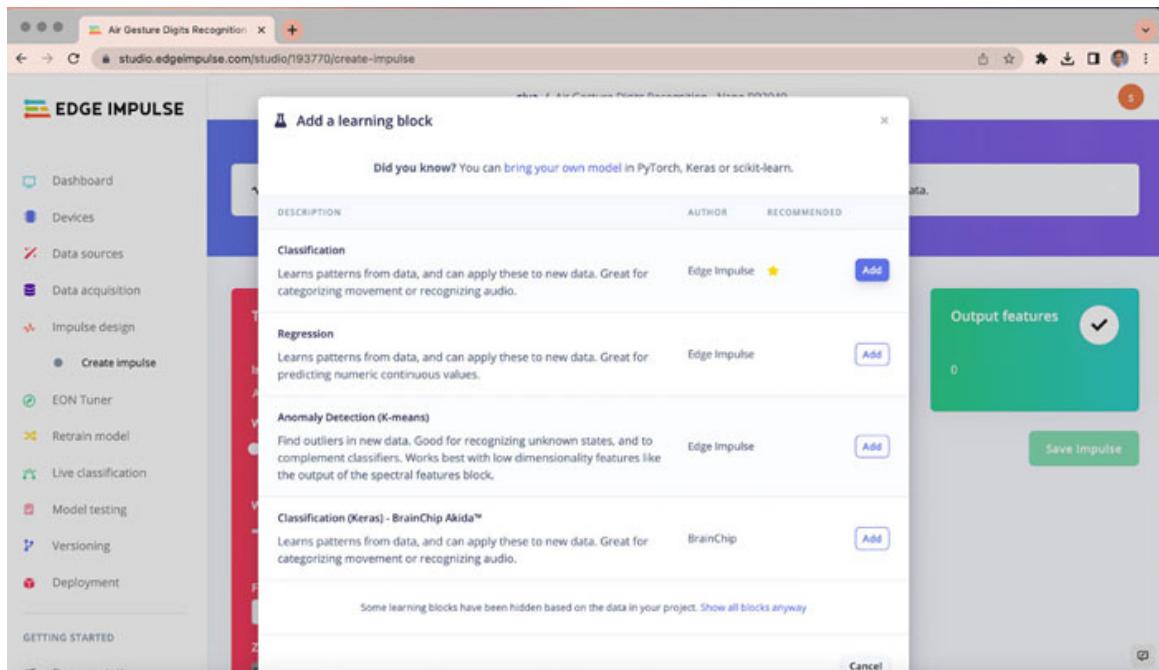
(a)



(b)



(c)



(d)

Figure 6.9 (a) to (d): Addition of different processing blocks for the model training in the Edge Impulse platform

Model training in Edge Impulse platform

Because the gathered data (Accx, Accy, Accz) is a time series data, in the edge impulse platform, we must define window size information, then add the classification block, followed by the processing block, which will be the spectral analysis. We must save our design when we have added these blocks.

After performing the previous steps, it is now time to extract the spectral feature from the gathered data. As the data is raw data sample, we may visualize the raw data sample and it may contain some high frequency ripple. Hence, before applying it to the neural network block, it is preferable to apply some sort of filtering to reduce the background noise in the data. We can use filters like the low pass filter to do this. The cutoff frequency is set at 3 Hz by default. For the filter, selection of order is critical, and thus we must apply the order in even numbers. If we increase the order, the response will be faster, but it will require more hardware and the execution speed will be slower due to the increased memory consumption. We chose sixth order for our design. After we applied filtering, the data waveform became considerably smoother than in earlier renderings. The number of training windows has been established, and it is now time to generate the feature.

This feature generation is very significant because from this feature generation, we can guess performance of our machine learning model. If our features are already separated, we can expect a high level of precision. If our features are not perfectly segregated, the ML model will struggle to predict the outcome.

We can select many inner layers in the Neural Network block. The model will consume more memory as the number of neurons and layers increases, and the model may not fit on our board.

An Impulse is the training pipeline for our machine learning model, and it is composed of the following three blocks:

- **Input Block:** This specifies the kind of data we are using to train our mode. Therefore, pay close attention to it. It might be a time series, or it could be images.
- **Processing Block:** The function of this block is to extract features.
- **Learning Block:** It is a neural network that has been trained to learn based on the data we provide.

When we are training your model, we have access to a variety of processing blocks that we can employ. Edge Impulse makes our life easier by providing us with a concise summary of the capabilities possessed by each processing block. For the edge impulse platform, everything is done automatically in the background, and the impulse will do everything for us. The floating-point

representation approach requires more memory compared to the fixed point. It essentially reduces a 32-bit floating-point model to an eight-bit quantized model. Because the microcontroller has limited hardware, we can only use less memory. Change the model's quantization from 32-bit floating point to an 8-bit quantized version. We can now see that the model's accuracy is 89%. However, in real circumstances, we must collect a greater number of training samples, and only then, the accuracy will be increased.

Overfitting can occur when a model detects all these factors flawlessly with 100% accuracy during training, but does not predict accurately for a new dataset.

[Figure 6.10](#) shows creating impulse with time series data and neural network selection:

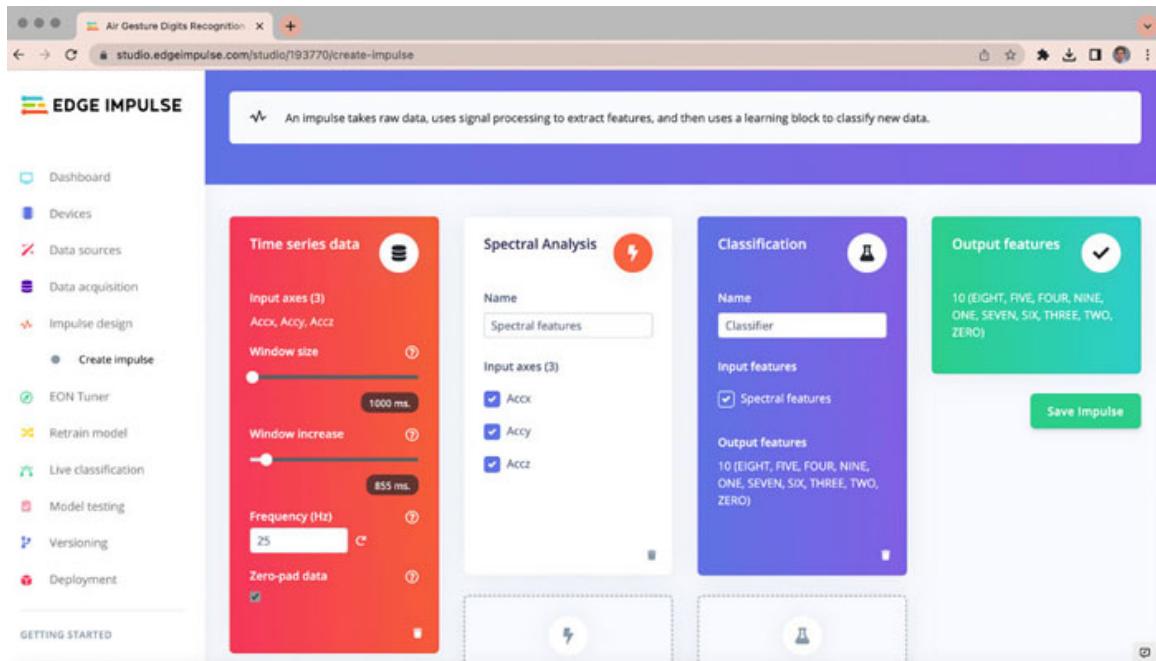


Figure 6.10: Creating impulse with time series data and neural network selection

[Figure 6.11](#) features the filter parameter selection for noise removal in the dataset:

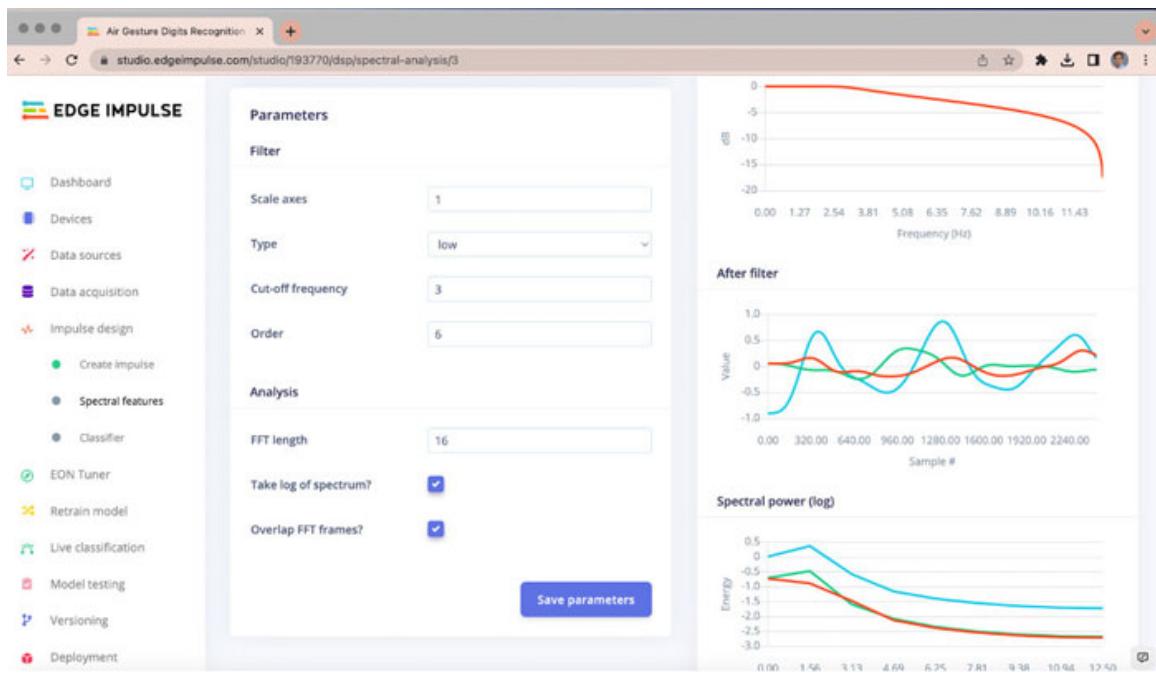
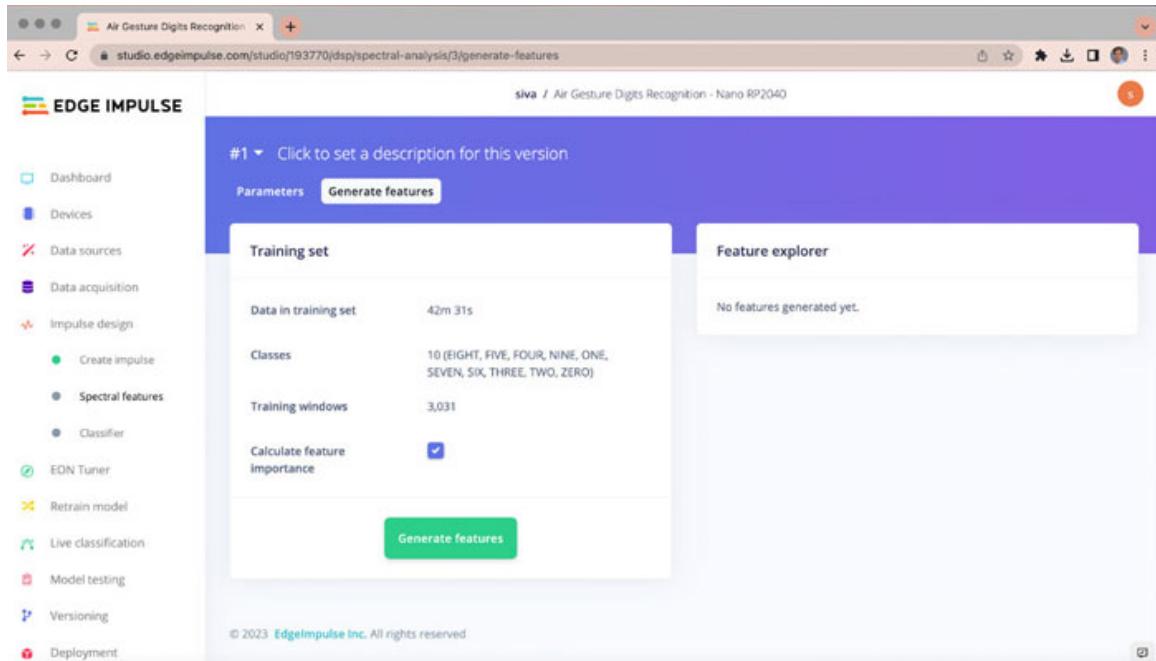
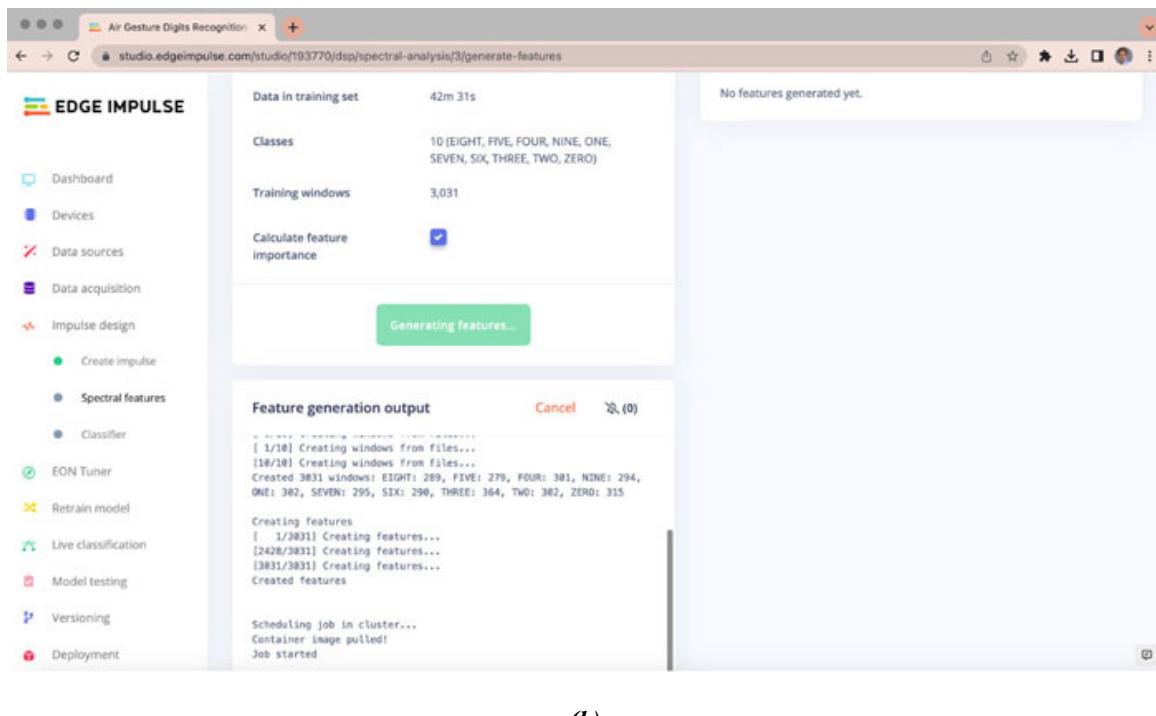


Figure 6.11: Filter parameter selection for noise removal in the dataset

Figure 6.12 (a) and **(b)** illustrates generating spectral features after creating the impulse design:



(a)



(b)

Figure 6.12 (a) and (b): Generating spectral features after creating the impulse design

Figure 6.13 features the generated spectral features with dataset information in impulse design:

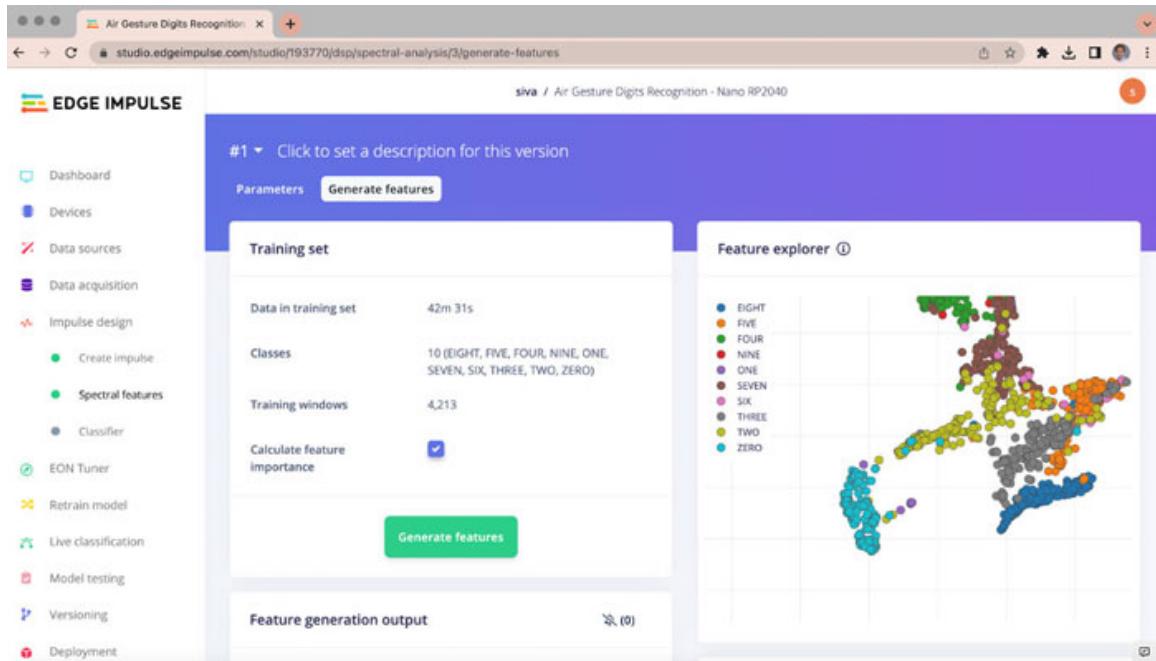


Figure 6.13: Generated spectral features with dataset information in impulse design

Figure 6.14 features the training parameters selection in Edge Impulse:

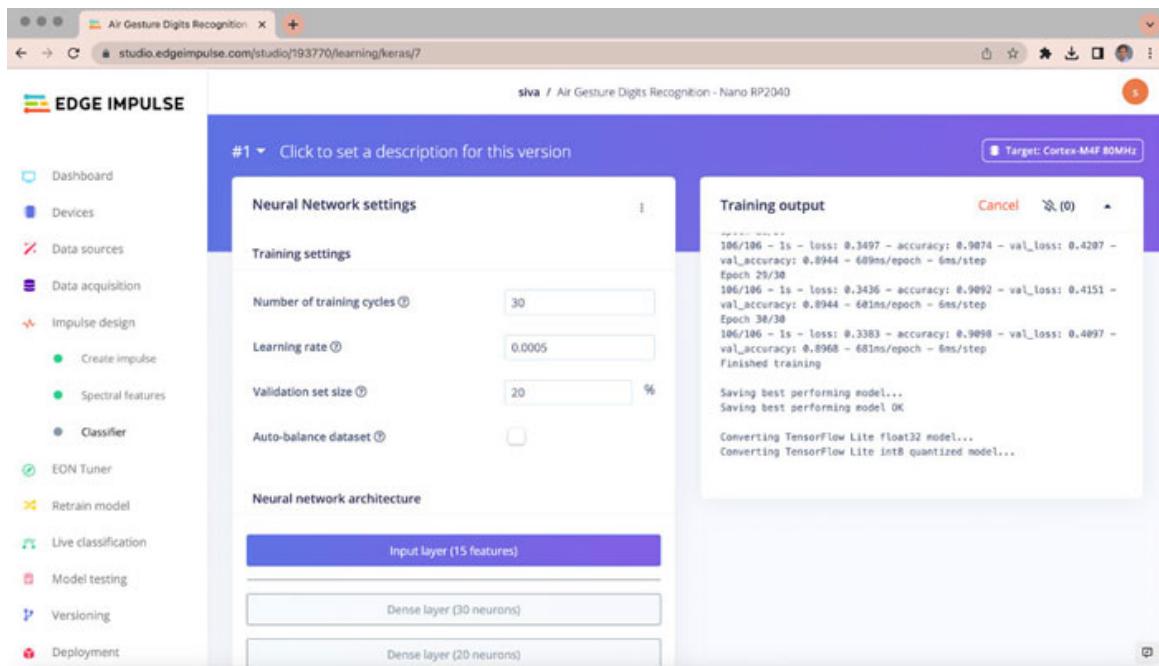
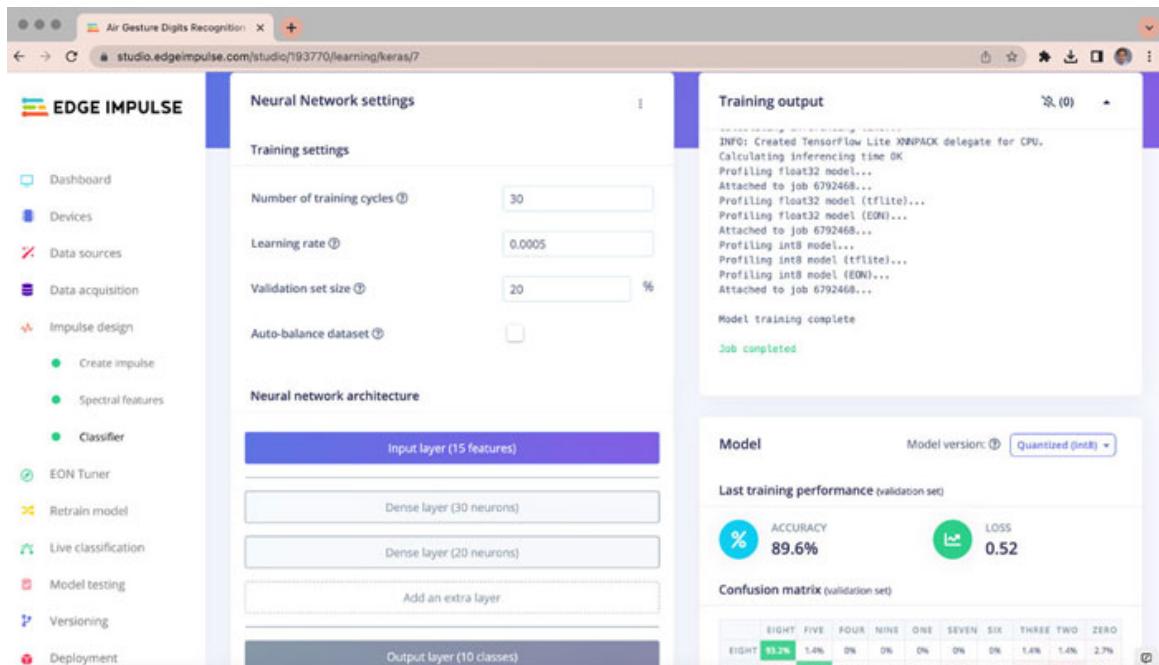
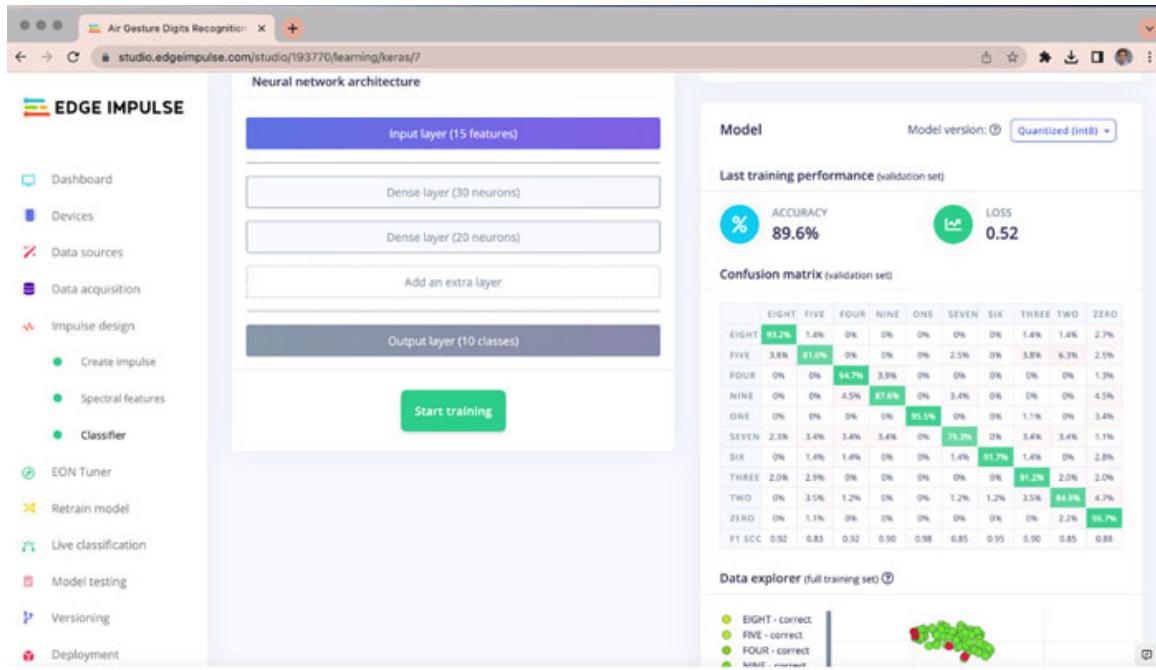


Figure 6.14: Training parameters selection in Edge Impulse

Figure 6.15 (a) to (b) features the training accuracy and confusion matrix for the model training in Edge Impulse:



(a)



(b)

Figure 6.15 (a) and (b): Training accuracy and confusion matrix for the model training in Edge Impulse

Figure 6.16 illustrates the visualizing of the complete trained dataset in Edge Impulse:

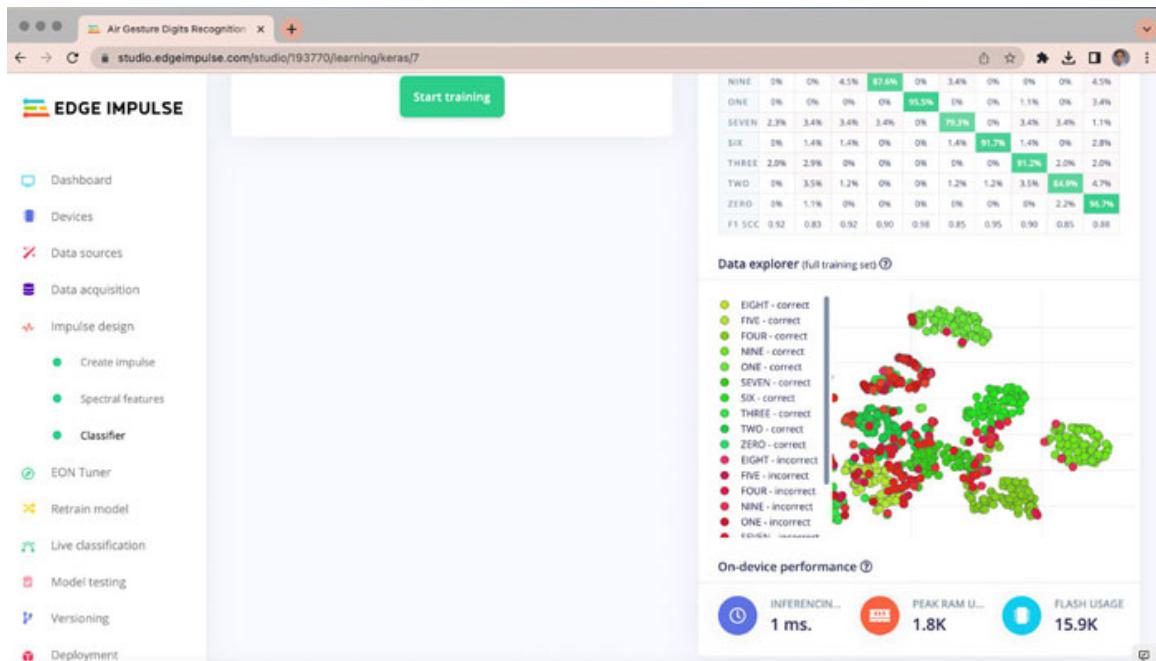


Figure 6.16: Visualizing complete trained dataset in Edge Impulse

Model testing with the collected data

After the model is trained with the training dataset, we can test the model with the test dataset shown in [Figure 6.17](#). The overall testing accuracy observed is 70% for the collected test data.

The screenshot shows the Edge Impulse studio interface. On the left is a sidebar with various options like Dashboard, Devices, Data sources, Data acquisition, Impulse design, Create impulse, Spectral features, Classifier, EON Tuner, Retrain model, Live classification, Model testing, Versioning, and Deployment. The main area has two tabs: 'Test data' and 'Model testing output'. The 'Test data' tab shows a table with rows for nine_test, eight_test, seven_test, six_test, five_test, four_test, and three_test. Each row includes columns for SAMPLE_NAM..., EXPECTED_OUT..., LEN..., ACCURACY, and RESULT. The 'Model testing output' tab shows a progress bar at 0% (0/0) and a status message: 'This lists all test data. You can manage this data through Data acquisition.'

Figure 6.17: Test dataset in Edge Impulse

[Figure 6.18](#) features the model testing output in Edge Impulse:

This screenshot is similar to Figure 6.17, showing the Edge Impulse studio interface. The 'Model testing output' tab is active, displaying a progress bar at 0% (0/0) and a status message: 'Scheduling job in cluster... Job started Reducing dimensions for visualizations... (IMAP(verbosetrue)) Construct Fuzzy simplicial set Sun Mar 5 15:01:01 2023 Finding Nearest Neighbors Sun Mar 5 15:01:03 2023 Finished Nearest Neighbor Search'. The rest of the interface is identical to Figure 6.17, including the sidebar and the 'Test data' table.

Figure 6.18: Model testing output in Edge Impulse

[Figure 6.19](#) features the model testing accuracy in Edge Impulse:

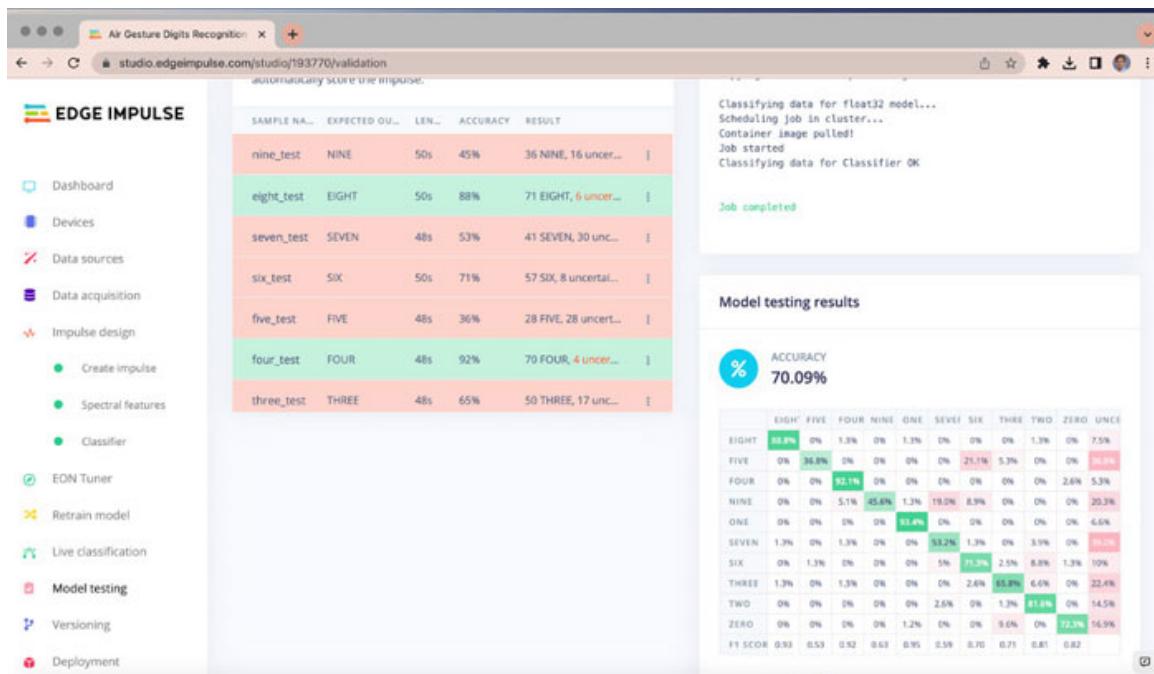


Figure 6.19: Model testing accuracy in Edge Impulse

Model deployment in Nano RP2040 board

There are various alternatives for the Nano RP2040 board. Therefore, we have distinct possibilities for each development board. We can also directly download the built-in firmware for all these supported boards. At present, the Nano RP2040 board is not currently supported by the Edge Impulse. To accomplish deployment on the board, we must first obtain the Arduino library. From the **Digital Signal Processing (DSP)** algorithm to the machine learning model, the EON Tuner optimizes the whole process. The EON Tuner runs multiple model configurations at the same time, depending on the target device and how it will be used. The EON compiler is a fantastic tool for further optimizing the model and increasing its correctness, and it will also use less memory. That means RAM requirements will be reduced because preserving memory is critical when working with a microcontroller because we are operating in a very limited hardware, and memory constraint environment. Now we use the integer-8 model rather than the float-32 model.

Now that we have seen all the RAM parameters for the model, it is time to create. Thus, it will generate the Arduino header file as well as all other available libraries, which we can save as zip file that can be saved on our laptop or computer.

Edge Impulse presently supports the following types of deployment options:

- Deploy as a library that can be customized.
- Deploy as a pre-built firmware, for development boards that are fully supported.
- Run on our phone or PC immediately.
- For Linux targets, use Edge Impulse for Linux.
- Make our own deployment block.

[Figure 6.20](#) features the model deployment with different hardware options in Edge impulse:

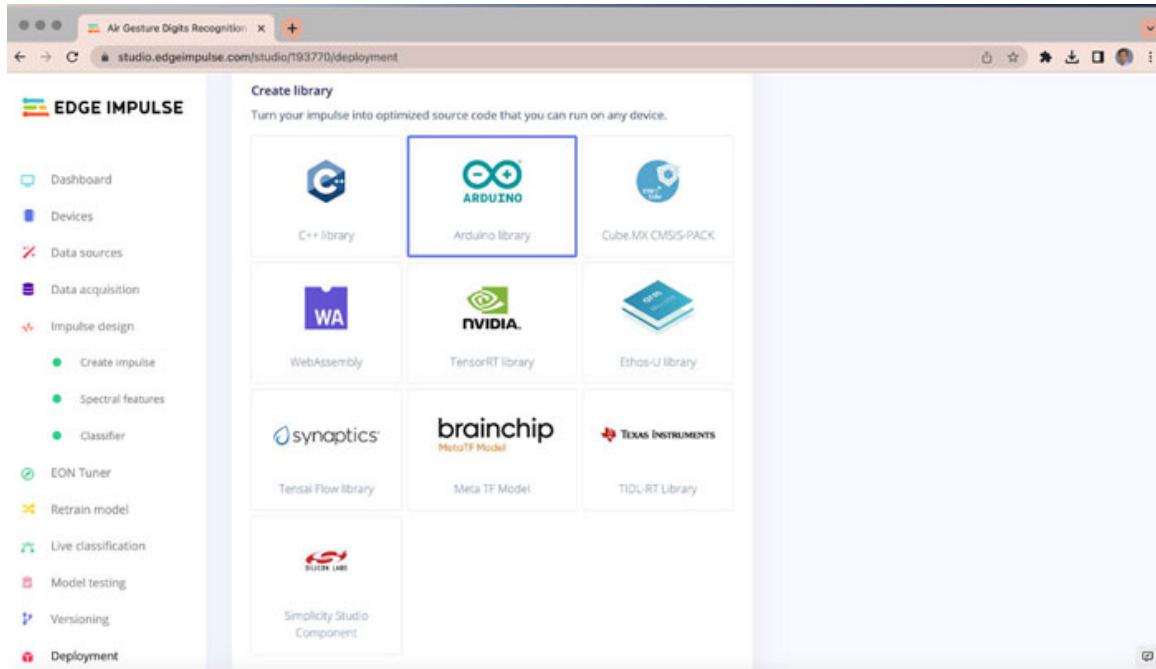


Figure 6.20: Model deployment with different hardware options in Edge Impulse

[Figure 6.21](#) features model deployment EON compiler build options in Edge Impulse:

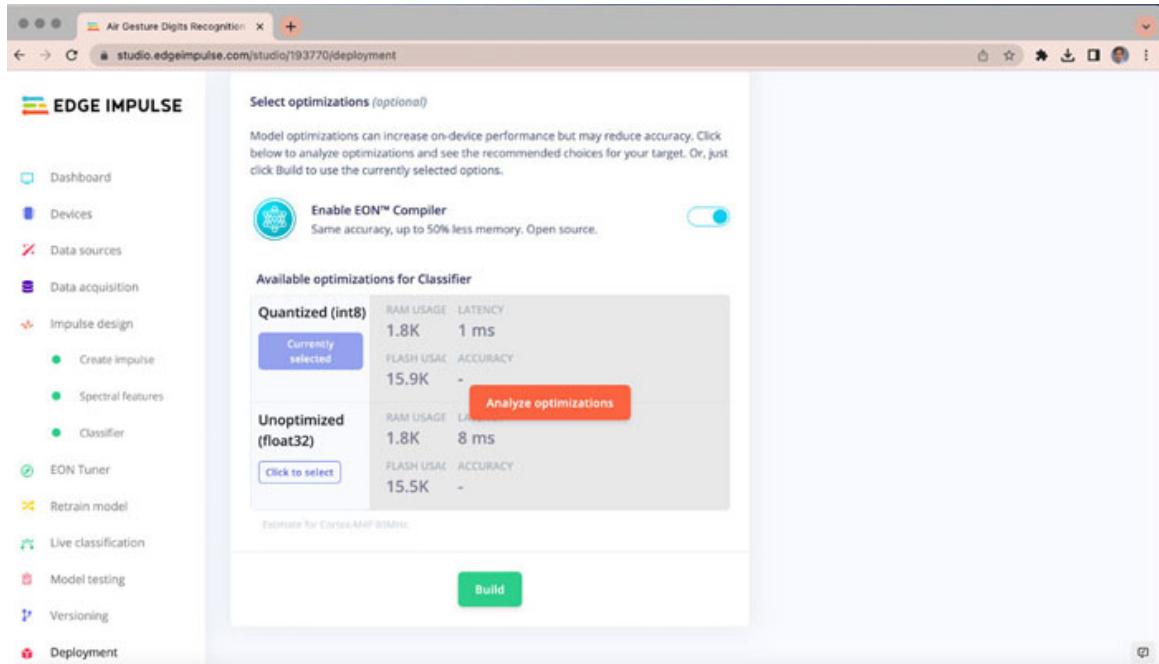
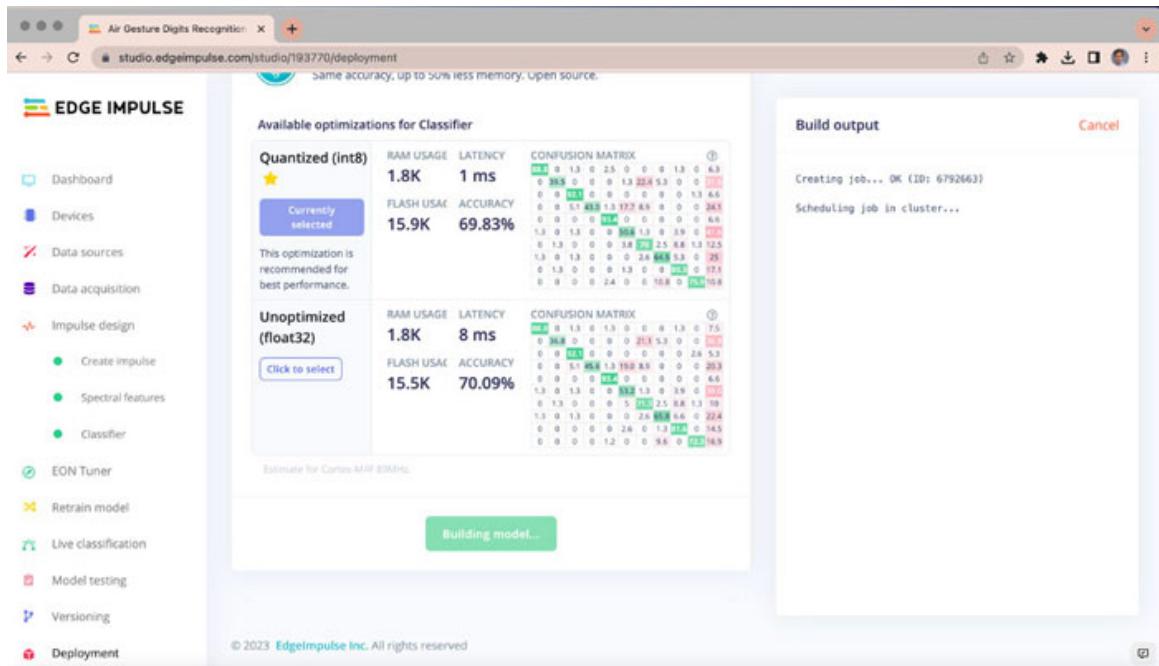
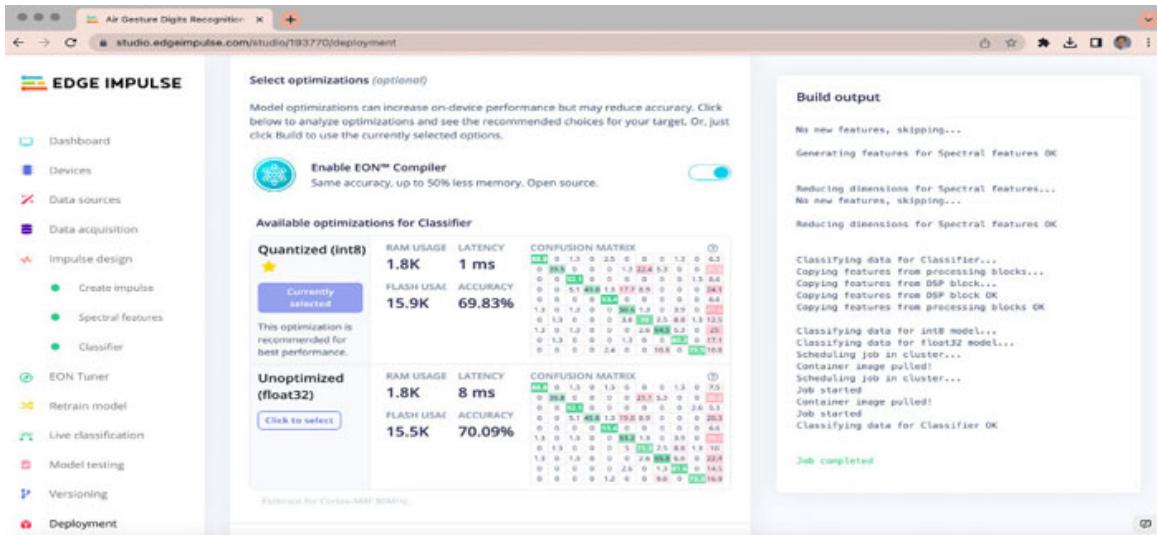


Figure 6.21: Model deployment EON compiler build options in Edge Impulse

Figure 6.22 (a) and **(b)** feature model deployment EON compiler build output in Edge Impulse:



(a)



(b)

Figure 6.22: Model deployment EON compiler build output in Edge Impulse

Figure 6.23 features the final model download in zip Arduino library in Edge Impulse:

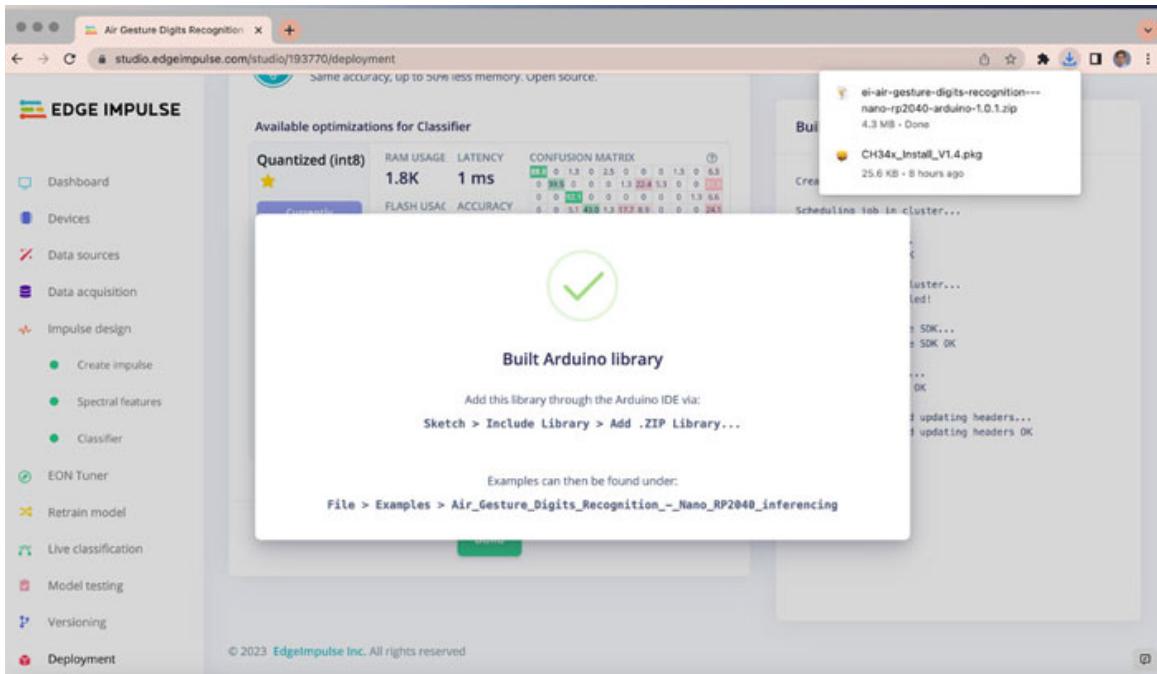


Figure 6.23: Final model download in zip Arduino library in Edge Impulse

Inferencing/Prediction of results with RP2040

For the inferencing/prediction of results with RP2040, follow the given steps:

1. First, we will see how to incorporate the library, which is generated from Impulse Studio. We need click on **Sketch** in IDE, then click on **Include Library**. Finally, click on **Add Library**. From there, we just add the library which is downloaded from Edge Impulse.
2. Now, simply select the library which is downloaded from the impulse and open it. In the file example section, we can see our library is present. Under the project name **Air_Gesture_Digits_Recognition_Nano_RP2040_inferencing**, our library will be present, as shown in the [Figure 6.24](#):

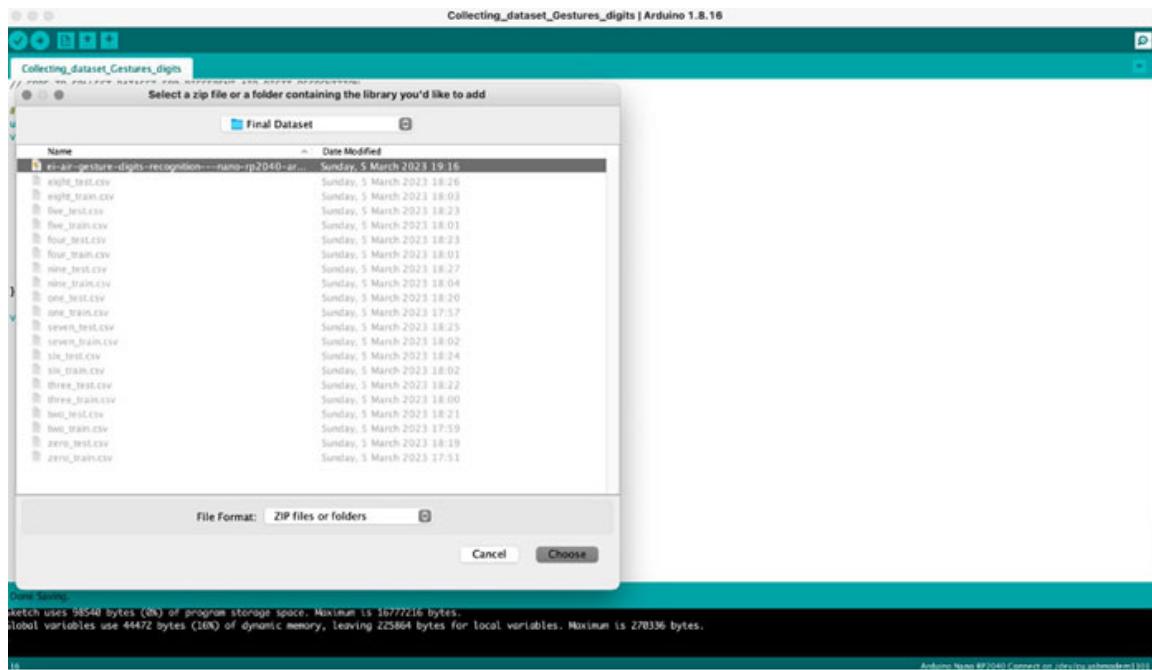


Figure 6.24: Arduino library download in Arduino IDE

3. Open a sample template. Since we are working with the accelerometer, simply click on that Nano RP2040 board. We need to change the header file to LSM6DSOX. Check on the serial baud rate. We also need to comment some of the blocks not used. After checking all parameter, we can do the upload action which may take a long time.
4. Now let us start the classification using the board. Make sure the orientation of board during training and testing must be same. We can move our board in different gesture positions and results will be displayed, based on the gesture in the serial port window.

[Figure 6.25](#) features Nano BLE Sense selection from Nano RP2040 inferencing in Arduino IDE:

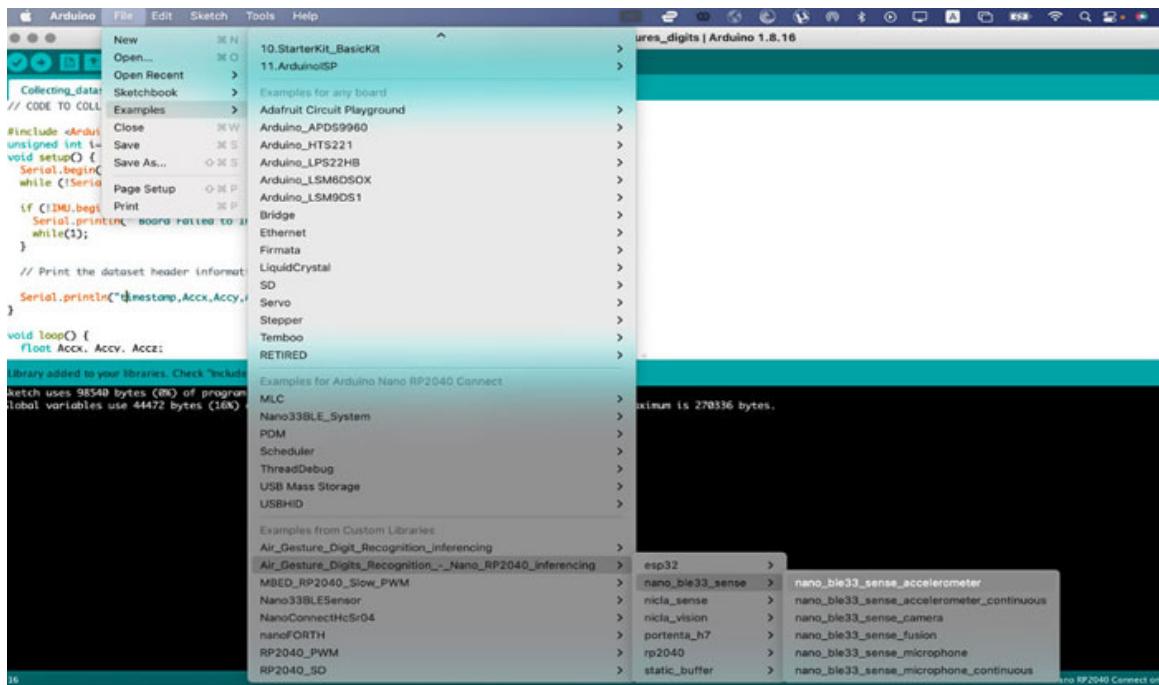


Figure 6.25: Nano BLE Sense selection from Nano RP2040 inferencing in Arduino IDE

Figure 6.26 features compiling sketch file from Nano RP2040 in Arduino IDE:

```
nano_ble33_sense_accelerometer | Arduino 1.8.16

/*
 * limitations under the License.
 *
 */

/* Includes */
#include <Air_Gesture_Digits_Recognition_-_Nano_RP2040_inferencing.h>
#include <Arduino_LSM6DSOX.h>

/* Constant defines */
#define CONVERT_G_TO_MS2 9.80665F
#define MAX_ACCEPTED_RANGE 2.0F      // starting 03/2022, models are generated setting range to +2, but this example use Arduino library which set range to +4g. If you are using an older model, ignore.

/* NOTE: If you run into TFLite arena allocation issue,
** This may be due to memory fragmentation.
** Try defining "-D_GLIBC_MALLOC_STATIC" in boards.local.txt (create
** if it doesn't exist) and copy this file to
** '$ARDUINO_CORE_INSTALL_PATH/arduino/hardware/embedded_core/<core_version>/'.
**
** See
** https://support.arduino.cc/hc/en-us/articles/360012070000-Where-are-the-installed-cores-located-
** to find where Arduino installs cores on your machine.
**
** If the problem persists then there's not enough memory for this model and application.
*/

/* Private variables */
static bool debug_nn = false; // Set this to true to see e.g. features generated from the raw signal.

Done compiling.

sketch uses 145795 bytes (8%) of program storage space. Maximum is 16777216 bytes.
Global variables use 47140 bytes (17%) of dynamic memory, leaving 223156 bytes for local variables. Maximum is 270336 bytes.

```

Figure 6.26: Compiling sketch file from Nano RP2040 in Arduino IDE

Figure 6.27 features inferencing results for Nano RP2040 in Arduino IDE:

```
name_ble33_sense_accelerometer
**
** See
** (https://support.arduino.cc)
** to find where Arduino IDE
** is located.
** If the problem persists:
Edge Impulse Inferencing Demo
DNU initialized

/* Private variables -----
static bool debug_nn = false;
Sampling...
*/
// Brief Arduino setup
void setup()
{
    // put your setup code here
    Serial.begin(9600);
    // comment out the below
    while (!Serial);
    Serial.println("Edge Impulse
    if (!DNU.begin()) {
        elprintf("Failed to
    } else {
        elprintf("DNU initia
    }

    // EET CLASSIFIER RAM SAM
}

Sketch uses 145795 bytes (88%)
Global variables use 47140 byt

Autoscroll Show timestamp Newline 9600 baud Clear output
```

Figure 6.27: Inferencing results for Nano RP2040 in Arduino IDE

[Figure 6.28](#) features inferencing results for air gesture digit “ONE” in Nano RP2040 in Arduino IDE:

nano_ble33_sense_accelerometer | Arduino 1.8.16

```
** See
** GitHub://support_arduino
** to find where Arduino libraries
** If the problem persists, file a bug report at
** https://github.com/arduino/Arduino

/* Private variables -----
static bool debug_nn = false;

// Brief Arduino setup
void setup()
{
    // put your setup code here, to run once:
    Serial.begin(9600);
    // comment out the below
    while (!Serial);
    Serial.println("Edge Impulse classifier ready");

    if (!IMU.begin()) {
        Serial.println("Failed to initialize IMU");
    } else {
        Serial.println("IMU initialized");
    }

    // If CFFT CLASSIFIER RAW SAMPLES are selected, uncomment the following line
    // CFFTClassifierRawSamples();
}

Sketch uses 145795 bytes (8%) of free memory.
Global variables use 47548 bytes (1%) of dynamic memory.
Starting inferencing in 2 seconds...
Sampling...
Predictions (DSP: 70 ms., Classification: 1 ms., Anomaly: 0 ms.):
  EIGHT: 0.00000
  FIVE: 0.00000
  FOUR: 0.00000
  NINE: 0.00000
  ONE: 0.00000
  SEVEN: 0.00000
  SIX: 0.00000
  THREE: 0.00000
  TWO: 0.00000
  ZERO: 0.00000

Starting inferencing in 2 seconds...
Sampling...
Predictions (DSP: 69 ms., Classification: 1 ms., Anomaly: 0 ms.):
  EIGHT: 0.00000
  FIVE: 0.00000
  FOUR: 0.00000
  NINE: 0.00000
  ONE: 0.00000
  SEVEN: 0.00000
  SIX: 0.00000
  THREE: 0.00000
  TWO: 0.00000
  ZERO: 0.00000

Starting inferencing in 2 seconds...
Sampling...
Predictions (DSP: 69 ms., Classification: 1 ms., Anomaly: 0 ms.):
  EIGHT: 0.00000
  FIVE: 0.00000
  FOUR: 0.00000
  NINE: 0.00000
  ONE: 0.00000
  SEVEN: 0.00000
  SIX: 0.00000
  THREE: 0.00000
  TWO: 0.00000
  ZERO: 0.00000
```

Figure 6.28: Inferencing results for air gesture digit “ONE” in Nano RP2040 in Arduino IDE

[Figure 6.29](#) features inferencing results for air gesture digit “FOUR” in Nano RP2040 in Arduino IDE:

The screenshot shows the Arduino IDE interface with the sketch `nano_ble33_sense_accelerometer`. The serial monitor window displays the following output:

```

nano_ble33_sense_accelerometer | Arduino 1.8.16
/dev/cu.usbmodem1101
Send

Predictions (OSP: 69 ms., Classification: 1 ms., Anomaly: 0 ms.):
EIGHT: 0.00000
FIVE: 0.03906
FOUR: 0.32812
NINE: 0.44141
ONE: 0.02734
SEVEN: 0.00391
SIX: 0.11719
THREE: 0.00391
TWO: 0.00000
ZERO: 0.03906

Starting inferencing in 2 seconds...
Sampling...
Predictions (OSP: 70 ms., Classification: 1 ms., Anomaly: 0 ms.):
EIGHT: 0.00000
FIVE: 0.00000
FOUR: 0.59609
NINE: 0.00000
ONE: 0.00000
SEVEN: 0.00391
SIX: 0.00000
THREE: 0.00000
TWO: 0.00000
ZERO: 0.00000

Starting inferencing in 2 seconds...
Sampling...

```

Sketch uses 145795 bytes (8%) Global variables use 47140 bytes

Autoscroll Show timestamp Newline 9600 baud Clear output

Figure 6.29: Inferencing results for air gesture digit “FOUR” in Nano RP2040 in Arduino IDE

Figure 6.30 features inferencing results for air gesture digit “THREE” in Nano RP2040 in Arduino IDE:

The screenshot shows the Arduino IDE interface with the sketch `nano_ble33_sense_accelerometer`. The serial monitor window displays the following output:

```

nano_ble33_sense_accelerometer | Arduino 1.8.16
/dev/cu.usbmodem1101
Send

Sampling...
Predictions (OSP: 70 ms., Classification: 1 ms., Anomaly: 0 ms.):
EIGHT: 0.01172
FIVE: 0.17188
FOUR: 0.00000
NINE: 0.00000
ONE: 0.00000
SEVEN: 0.00391
SIX: 0.00391
THREE: 0.76562
TWO: 0.04688
ZERO: 0.00000

Starting inferencing in 2 seconds...
Sampling...
Predictions (OSP: 69 ms., Classification: 1 ms., Anomaly: 0 ms.):
EIGHT: 0.00000
FIVE: 0.00000
FOUR: 0.00000
NINE: 0.00000
ONE: 0.00000
SEVEN: 0.00391
SIX: 0.00000
THREE: 0.92578
TWO: 0.02734
ZERO: 0.00000

Starting inferencing in 2 seconds...
Sampling...

```

Sketch uses 145795 bytes (8%) Global variables use 47140 bytes

Autoscroll Show timestamp Newline 9600 baud Clear output

Figure 6.30: Inferencing results for air gesture digit “THREE” in Nano RP2040 in Arduino IDE

Figure 6.31 features inferencing results for air gesture digit “TWO” in Nano RP2040 in Arduino IDE:

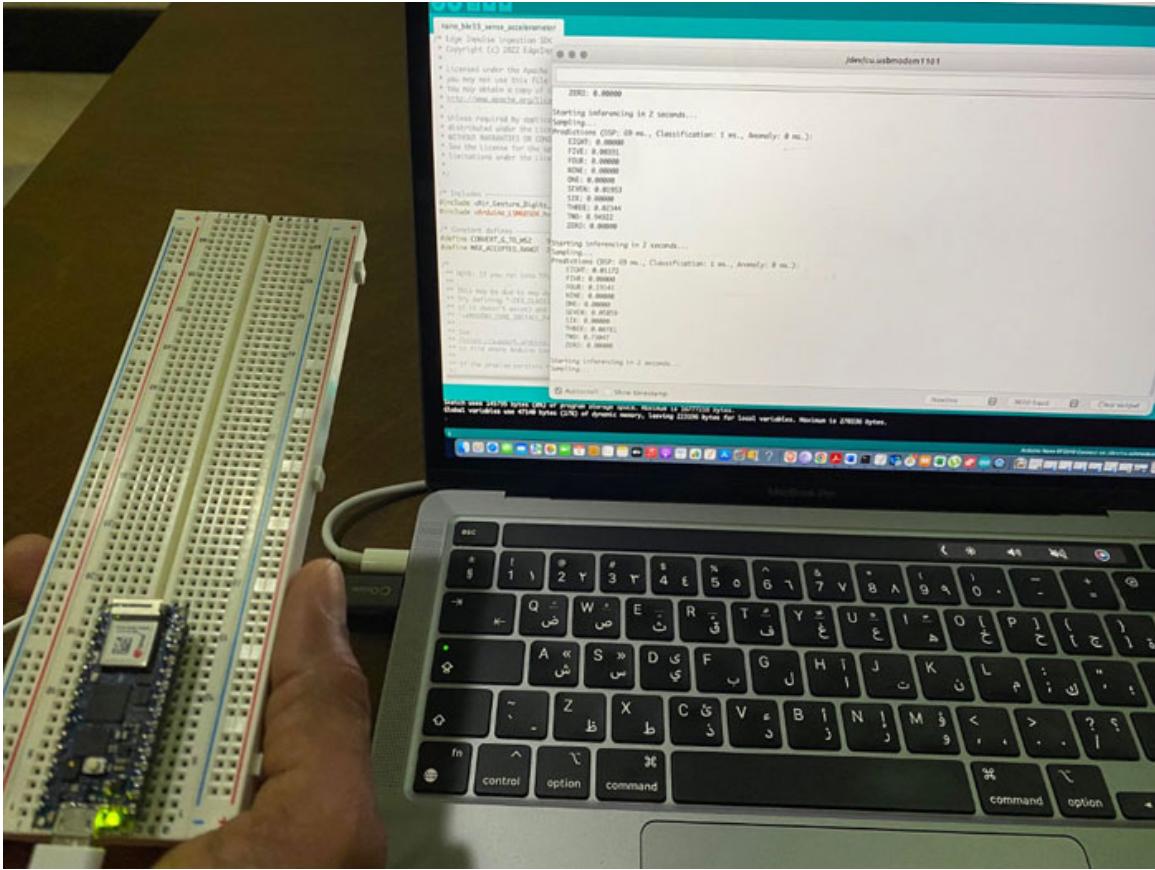


Figure 6.31: Inferencing results for air gesture digit “TWO” in Nano RP2040 in Arduino IDE

Conclusion

In this chapter, we learned how to implement air gesture digit recognition using an Arduino Nano RP2040 board. The Nano RP2040 TinyML board is discussed in greater detail after a quick introduction. We started from scratch when implementing the use case, and focused primarily on the steps that were involved, such as collecting data using the Nano RP2040 board and the Arduino IDE for a variety of gestures (zero to nine), cleaning the dataset, creating a project in the Edge Impulse platform, and uploading the datasets for use in training and testing. After uploading the datasets, we proceeded to select the necessary architecture for the classification (which included establishing the development framework), and then we carried out the training activity using those datasets. We constructed the final model which is installed in the Nano RP2040 board in preparation for the final inference. It is necessary to make the necessary adjustments to the code that has been built, before it can be deployed on the Nano RP2040 board. The final inference results are observed in real time for different air gestures digits for the deployed code on the board.

Key facts

- Air gesture digit recognition is performed using the Arduino Nano RP2040 board and Edge impulse platform, and it involves many steps such as collecting and processing data, training and deploying a machine learning model using edge impulse platform, and the inferencing is performed using Arduino IDE.
- Based on the specifications of our target device, the EON Tuner will simultaneously run several different model configurations.
- Edge impulse supports multiple types of deployment options (customizable library, pre-built firmware, run directly on our phone/system, Linux targets and custom deployment blocks).
- Due to the lack of edge impulse's direct support on the Nano RP2040 board, we implemented it as a customizable Arduino library.

Questions

1. What are the main steps involved in deploying ML model on any target platform?
2. What is the benefit of Impulse?
3. List the advantages of Nano RP2040 board compared with previous versions.
4. What is overfitting and what are the problems associated with overfitting?
5. EON Tuner performs end-to-end optimizations: True or False?
6. Can we deploy the models directly on our mobile device? If yes, give an example.

References

1. <https://www.edgeimpulse.com/>
2. <https://store.arduino.cc/products/arduino-nano-rp2040-connect>
3. <https://docs.arduino.cc/tutorials/nano-33-ble-sense/gesture-sensor>
4. Sharma, J.K., Gupta, R., and Pathak, V.K., 2015, December. Numeral gesture recognition using leap motion sensor. In 2015 International Conference on Computational Intelligence and Communication Networks (CICN) (pp. 411-414). IEEE.

5. Wang, Z., Song, X., Fan, J., Chen, F., Zhou, N., Guo, Y. and Chen, D., 2021, May. WiDG: An Air Hand Gesture Recognition System Based on CSI and Deep Learning. In 2021 33rd Chinese Control and Decision Conference (CCDC) (pp. 1243-1248). IEEE.
6. <https://www.udemy.com/course/tinyml-with-arduino-nano-rp2040-connect/>
7. <https://www.udemy.com/course/making-gesture-controlled-projects-in-electronics/>
8. <https://levelup.gitconnected.com/creating-a-machine-learning-workflow-for-arduino-nano-rp2040-connect-with-edge-impulse-284792021e79>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 7

Advance Implementation with TinyML Board

Introduction

We have seen how code is written for off-the-shelf microcontrollers in the previous chapter. Though writing a code gives the user a lot of control, the development is bogged down due to the speed of code writing. One of the best promises of machine learning is that the code writing can be replaced with a data driven model.

We have also seen that no matter how much efficiently a code is written, **Complex Instruction Set Computer (CISC)** or **Reduced Instruction Set Computer (RISC)** style processors are quite inefficient. In this chapter, we will see how to leverage **Application Specific Integrated Circuits (ASICs)** which perform neural computation at the wire speed. Since the neural network is implemented as a hardware, there is really no need to write the code. Rather, there are registers which can configure neural network computation flow. We will consider two such ASICs: NDP101 and NDP120. We will refer to these ASICs as **Neural Decision Processors (NDP)**.

Structure

In this chapter, the following topics will be covered:

- NDP101 Architecture
- NDP120 Architecture
- Practical implementation and deployment
 - Creating project
 - Uploading data
 - Impulse Design

- Epochs setting
- Learning rate setting
- Validation data set setting
- Auto balance setting
- Data augmentation
- Neural network architecture
- Neural network training
- Model testing
- Deployment

Objectives

In this chapter, we will go deeper and make a complete project from scratch and test it on real hardware. The project will be phrase detection for the phrases “Marvin go” and “Marvin stop”. We are using “Marvin” as the wakeup word detection, just as “Alexa” or “Siri” are used. We chose “Marvin” as this was a name available in the `speech_commands_v0.02` (see reference) that is available publicly. “go” and “stop” keywords were also taken from the previously mentioned speech commands dataset. A complete project is made within Edge Impulse studio (see reference) and deployed on the TinyML board (see reference).

NDP101 Architecture

NDP101 is based on Syntiant’s core 1 technology. As shown in [*Figure 7.1*](#), a fixed neural network architecture is used in this chip, which is more than sufficient for many applications. It is a 4-layer neural network. Input layer is configurable to as many as 2048 inputs, although typically, 1600 inputs are used. The input layer produces 256 outputs, which becomes input to the next layer. Before the input is presented to the next layer, a ReLu transformation is applied. The two hidden layers take 256 inputs and produce 256 outputs. The output layer takes 256 inputs and produces 64 classifiers. The total number of parameters comes out to be about 500k. This architecture has been proven to handle keyword detection as well as motion-based gestures.

[*Figure 7.1*](#) features the four-layer neural network used in NDP101:

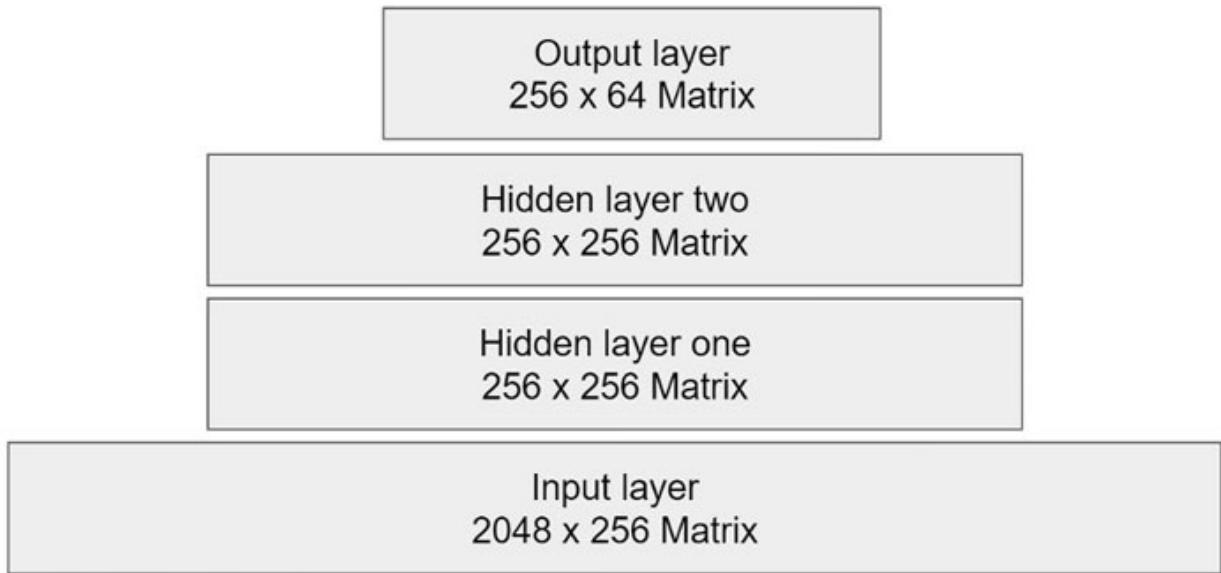


Figure 7.1: Fixed four-layer neural network used in NDP101

The NDP101 chip also integrates many functions, which makes it convenient to use, as shown in [Figure 7.2](#):

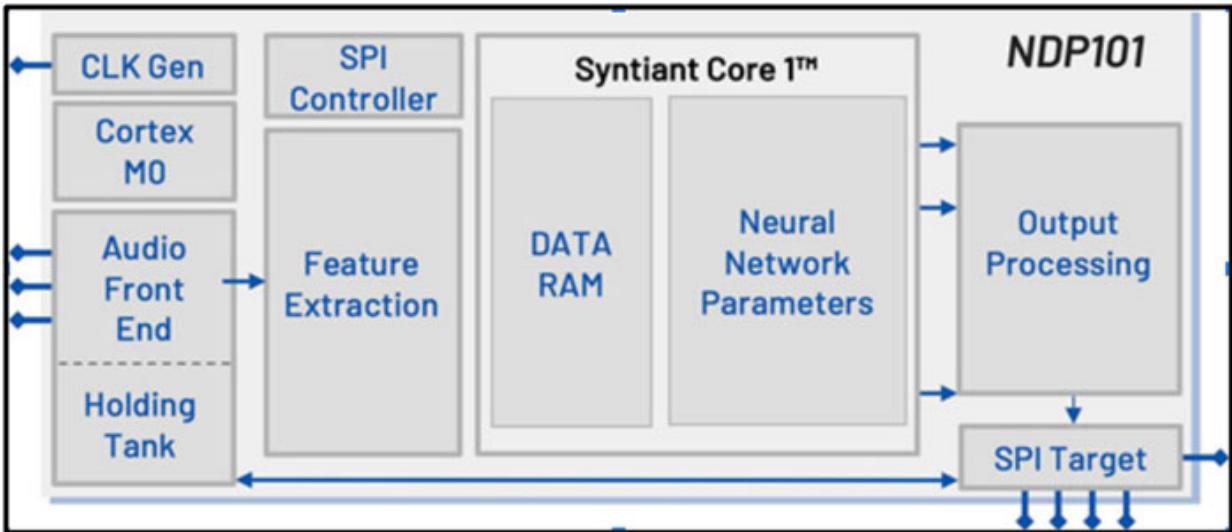


Figure 7.2: Block diagram of NDP101

As we can see in the preceding figure, the NDP101 chip incorporates ARM's M0 Cortex MCU. The MCU provides a user interface and does some post processing as well.

The SPI Target block is used for interfacing to the host. The neural decision processor is configured by SPI interface, where the host processor will be the controller and the neural decision processor will be the target device.

Audio Front End is a dedicated block for the microphone interface, since audio is one of the most significant use cases. A digital microphone can interface directly with the neural decision processor. The microphone interface is referred to as **Pulse Density Modulation (PDM)** interface, where sigma delta **Analog to Digital Converter (ADC)**'s outputs are sent directly to the neural decision processor. The digital PDM stream is decimated to produce 16kHz signal with 16 bits of resolution.

Feature extraction block converts the time domain signal into a frequency domain signal. Though it is possible to feed the 16kHz decimated data directly to the neural network, it is quite inefficient from a computation point of view. Traditionally, **Mel** frequency conversion is used. The audio signal is passed through a high pass signal, thus mimicking human hearing. Then using FFT transformation, the time domain signal is converted into frequency domain signal. Only the magnitude part is retained. Then another non-linear log transformation is applied.

A window of 512 points is taken from the incoming continuous stream of audio data, before converting it to frequency domain. With a 16kHz sampling rate, it represents 32ms of audio data. To avoid any artifacts of the windowing, proprietary windowing techniques (for example, Hanning/Hamming window) are used. To avoid loss of information, the windows are overlapping by 8ms or 128 samples. Thus, the delay between the two windows will be $32 - 8 = 24$ ms or 384 samples. For typical size of input tensor of 40 windows, this will cover $32 + 39 \times 24 = 968$ ms. [Figure 7.3](#) shows how the incoming audio stream is converted to the frequency domain spectrogram:

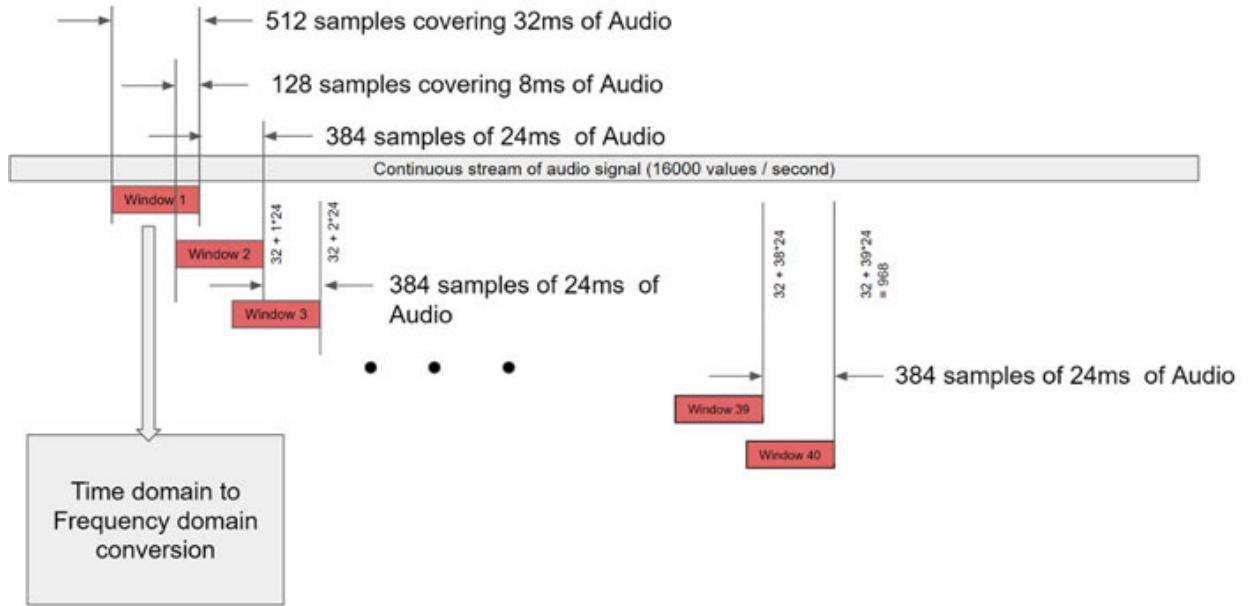


Figure 7.3: Time domain signal conversion to frequency domain

From 512 samples in each frequency domain conversion, 256 magnitude values are generated with increasing frequency. To reduce the input complexity while mimicking human hearing, 40 bins are created. Thus, 256 values are mapped into 40 values. The distribution is not uniform. Lower frequency bins have fewer magnitude values lumped, while higher frequency bins have more frequency values.

A spectrogram is generated, which is an arrangement in two dimensions. X-axis represents the time when the frequency domain conversion was done. At each point, 40 frequency bins are plotted vertically on the Y axis. This makes a 2-dimensional image. The values are color coded to show intensity. We will discuss more about Spectrogram in the visualization section.

Input Holding tank: There is a provision of holding 2-3 seconds worth of audio data in a circular buffer, so that we can capture the data when a match is inferred. This can help in double checking the data, debugging the system, as well as data capture for future training.

NDP120 Architecture

NDP120 is based on Syntiant's Core 2 design. It is mostly like NDP101, except that it has almost 2x parameters and flexibility in the neural network design. It also has a dedicated **Digital Signal Processing (DSP)** block which can give flexibility in generating features.

Practical implementation and deployment

With hardware implementation of feature extraction and neural network, we need to only configure the chip as compared to writing code. A **Graphical User Interface (GUI)** based platform offered by Edge Impulse is a very convenient option. Edge Impulse is a 3rd party platform which has both free and paid options to use their platform. The configuration firmware is automated, based on the GUI system design and machine learning flow. This makes it a zero-code AI deployment system.

Creating a project

Once the user makes an account with Edge Impulse, a new project can be created by clicking on the user symbol on the top right corner. Click on `+ create new project`, and that will open another panel where the user can enter the name of the project. *Figure 7.4* shows the panel to create a new project:

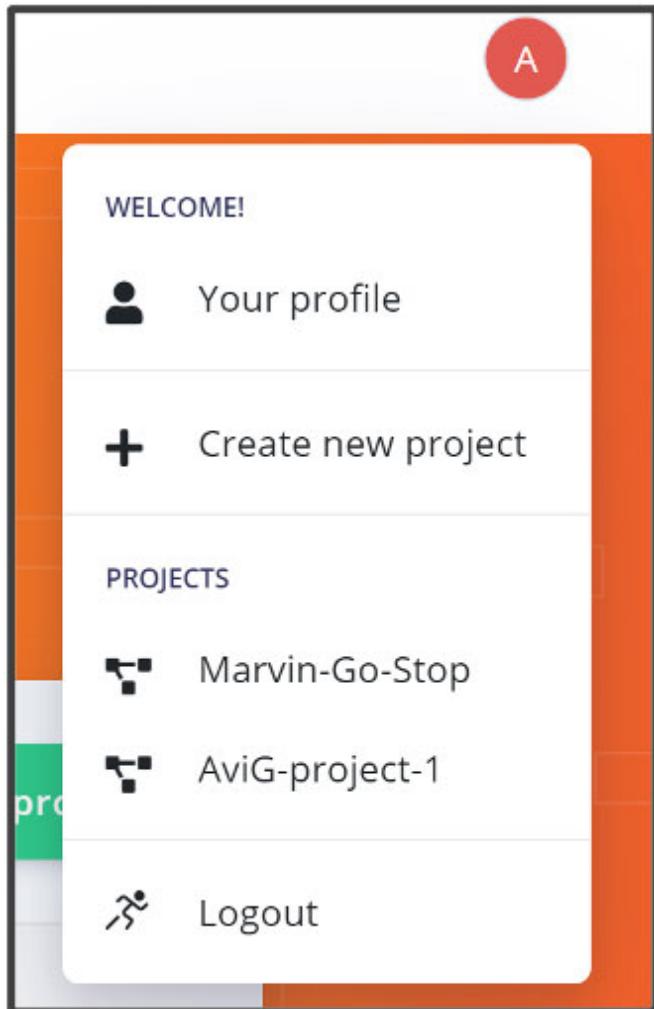


Figure 7.4: Panel to create a new project

A new panel will open when a new project is created, as shown in [Figure 7.5](#). By default, the **Developer** option is selected, which is a free of cost option, although with some limitations. The **Enterprise** option is the paid subscription option. Choose the first option for now and then click on the **Create new project**. Refer to [Figure 7.5](#):

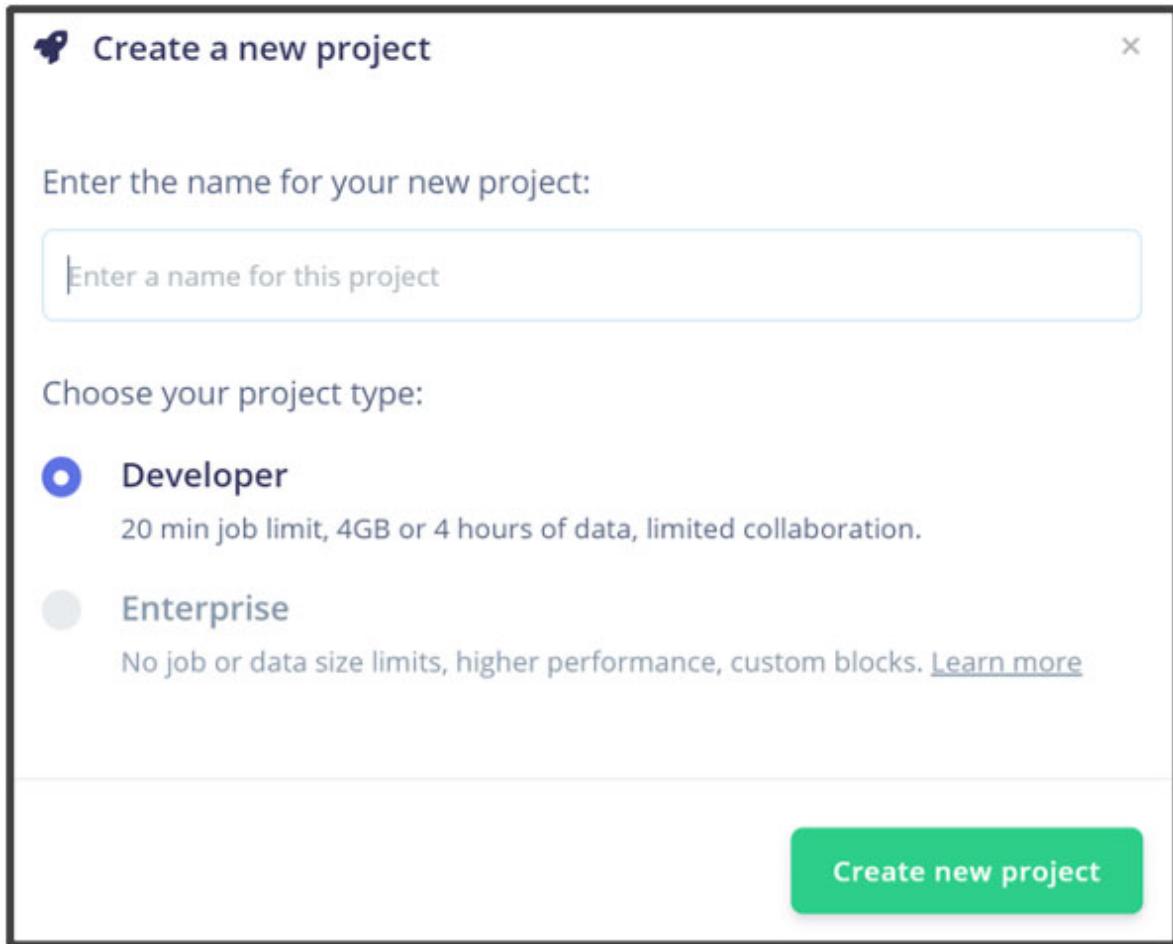


Figure 7.5: Panel to create a new project

A wizard panel shown in [*Figure 7.5*](#), will open with the title `Welcome to your new Edge Impulse project!` In this chapter, we will use an audio project. Choose the `Audio` option, as shown in [*Figure 7.6*](#):

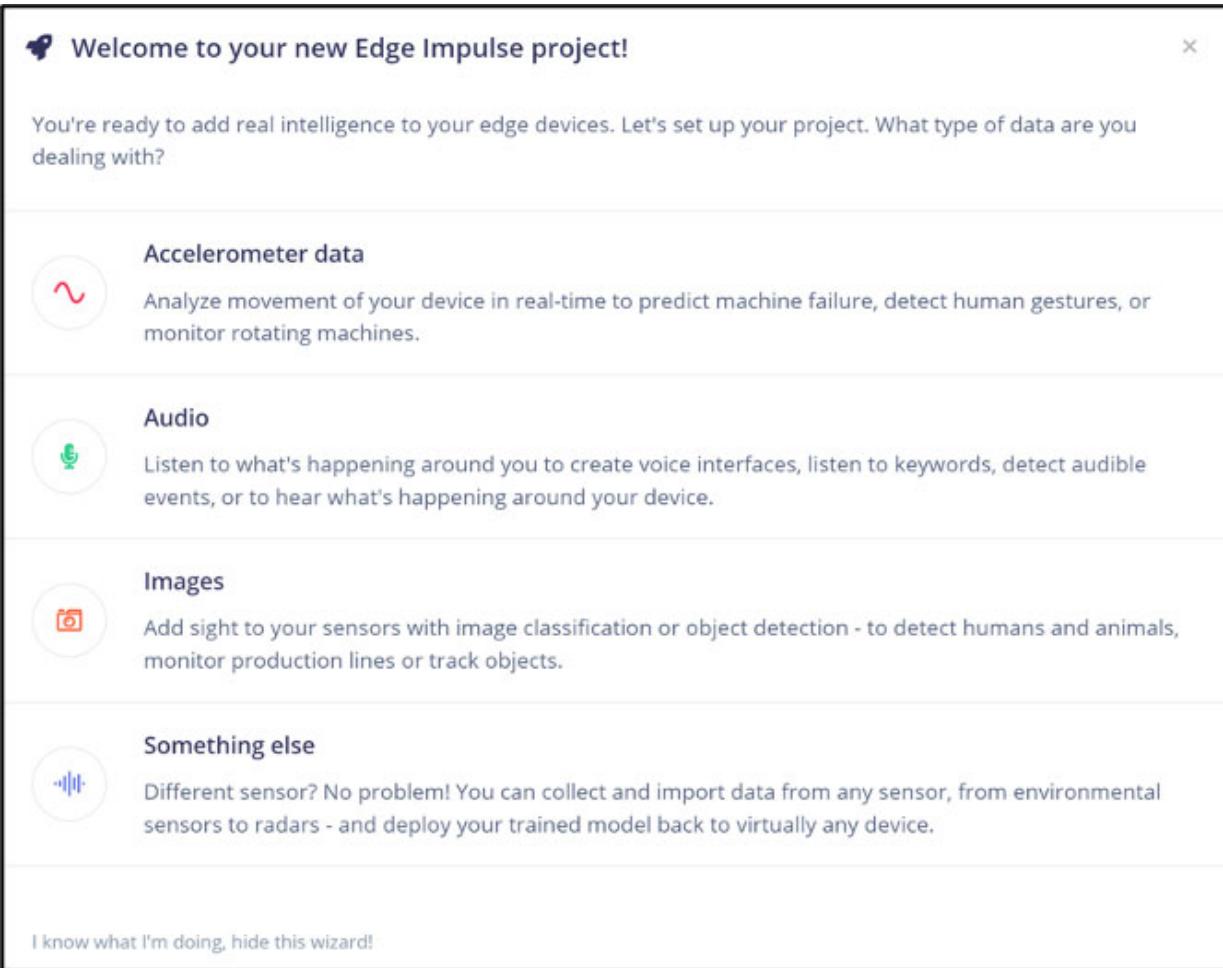


Figure 7.6: Welcome panel after creating a new project

Uploading Data

Once you choose the Audio option, another panel will open. It is recommended to use already captured data that is sanitized. Therefore, choose **Go to the uploader** option. You will see a panel as shown in [Figure 7.7](#).

It is expected that the data for training and tests are already organized in your personal computer. Browse to the training directory where sanitized data of the keyword is kept, and then select all the files. Then choose the **Training** option. Choose the option **Enter label** and type the name of the keyword. Click on **Begin upload** to upload the data into the studio:

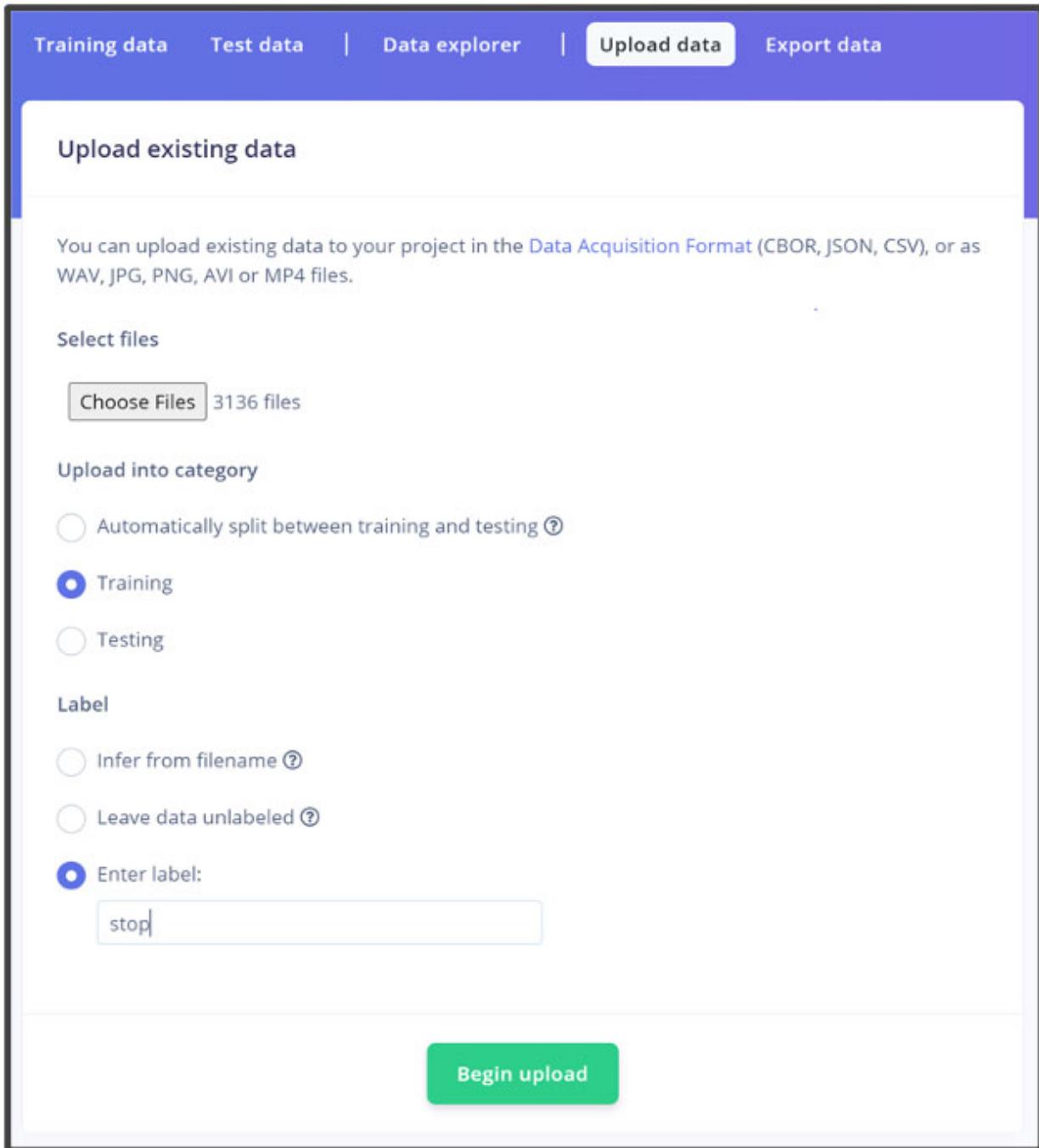


Figure 7.7: Uploading data into Edge Impulse studio

Another panel, `Upload output` shown in [Figure 7.8](#), comes up on the top right corner, which shows the activities and error messages, if any. It takes time to upload. Wait until all the files are uploaded. However, there may be instances when some of the files are not uploaded.

Continue the process to upload all the training data. The open set should be labelled such that it is listed as the last classifier in the alphanumeric order. In our example, we labelled it as `z_openset`, so that it is listed as the last classifier. Then choose the `test data` option to upload test data.

Refer to [Figure 7.8](#):

Upload output

```
[3120/3136] Uploading fa52ddf6_nohash_3.wav OK
[3121/3136] Uploading fa57ab3b_nohash_3.wav OK
[3122/3136] Uploading fa57ab3b_nohash_1.wav OK
[3123/3136] Uploading fa57ab3b_nohash_4.wav OK
[3124/3136] Uploading fa44fcf5_nohash_1.wav OK
[3125/3136] Uploading fb7eb481_nohash_3.wav OK
[3126/3136] Uploading fa446c16_nohash_1.wav OK
[3127/3136] Uploading fb7eb481_nohash_0.wav OK
[3128/3136] Uploading facd97c0_nohash_0.wav OK
[3129/3136] Uploading fa446c16_nohash_2.wav OK
[3130/3136] Uploading fa7895de_nohash_0.wav OK
[3131/3136] Uploading fb7eb481_nohash_1.wav OK
[3132/3136] Uploading fad7a69a_nohash_0.wav OK
[3133/3136] Uploading fb7eb481_nohash_2.wav OK
[3134/3136] Uploading fa446c16_nohash_3.wav OK
[3135/3136] Uploading fa7895de_nohash_1.wav OK
[3136/3136] Uploading facd97c0_nohash_1.wav OK
```

Done. Files uploaded successful: 3115. Files that failed to upload: 21.

Job completed

Figure 7.8: Message panel to show the “Upload output”

Once all the data is uploaded, it can be reviewed by clicking on the **Training Data** button on the top left corner, as shown in [*Figure 7.9*](#). Then click on the filter icon in the menu labelled as **Collected data**, also shown

in [Figure 7.9](#). The pie chart on the top left shows the training data combination. Ideally, all the classifiers should be of the same size. However here, the data for “Marvin” is significantly less than the data for “Go”, “Stop” and `z_openset` data. Another pie chart on the right side, shows the split between training and test. For production models, the training set could be large (in millions), while test sets could be in thousands. So, the pie chart may look the same or could be even more skewed. The filter option shows the actual number of training sets per classifiers.

More data can be uploaded by capturing more data, to balance all the classifiers. We will proceed here despite this not being perfectly balanced data. Refer to the following figure:

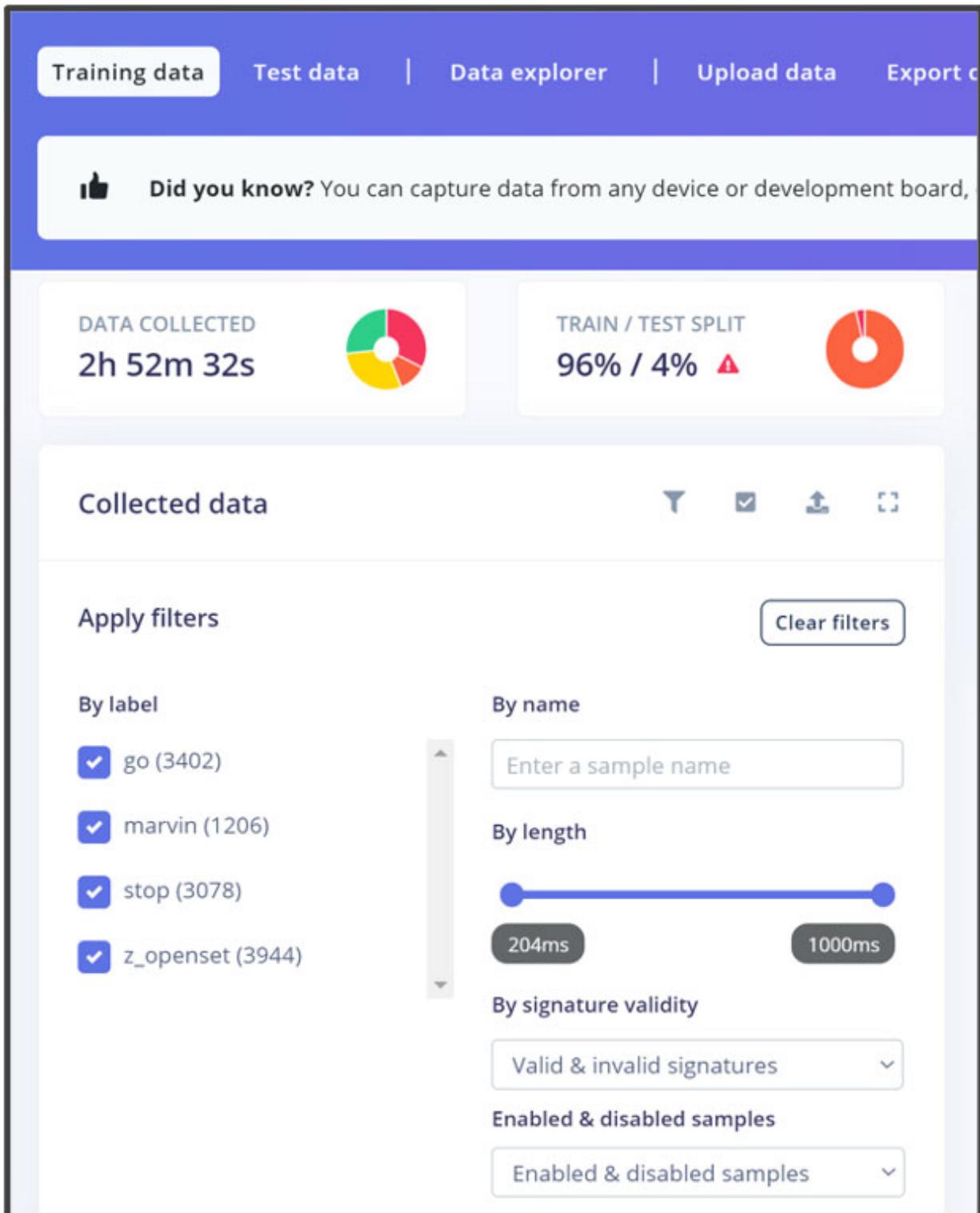


Figure 7.9: Exploring training data

Impulse Design

Neural network design and training steps can be found under the **Impulse Design** menu. Depending on your browser size, you may see the main menu or you may have to invoke it. Your browser may or may not show the left menu. If it does not, then you can invoke it by clicking on the three lines on the very top left corner. Refer to [Figure 7.10](#):

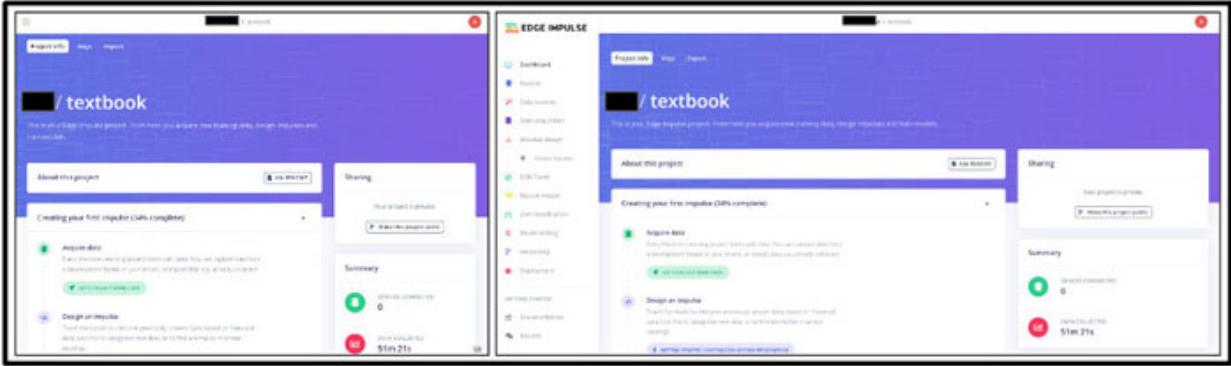


Figure 7.10: Invoking main menu which may not be shown in certain browsers. Screen shot of the display with less resolution where left menu does not show up (left). Screen shot where left menu shows up (right).

You may see **Create Impulse** under the **Impulse Design** menu. Click on the **Create Impulse** Menu. It will open a new panel, as shown in [Figure 7.11](#):

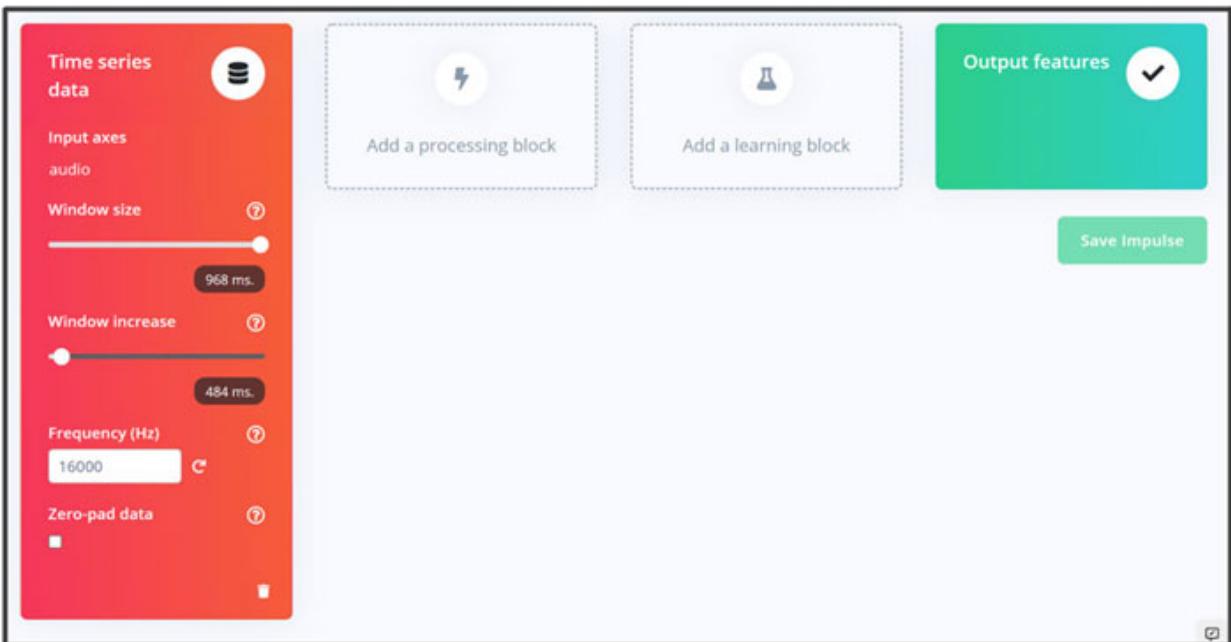


Figure: 7.11: Create impulse menu in Edge Impulse studio

In the left column labelled as **Time series data**, change the **window size** to 968ms. It may be easier to type the value in the box, as compared to changing it on the bar. **Window increase** can be set to 484ms, which is half of 968ms. If a training wav file is longer than 968ms, this setting will create more training sets from the one training wav file. More sets will be created by cropping the data of 968ms while shifting the cropping window by 484ms. The choice of shifting the audio data by 484ms is somewhat arbitrary. If a smaller delay is chosen, then more training sets will be generated.

All these training sets will be labelled the same classifier under which the file is saved. If the size of the file is for a duration slightly longer than the tensor size but smaller than **window size** and **window increase**, that is, 1452ms (968ms + 484ms), then only one training tensor will be generated. If the size of the file is longer than 1452ms, then two or more tensors will be generated. In certain cases, this may be ok; for example, if the classifier is representing a continuous event such as the whistling sound of wind, grinding sound of a failing ball bearing and so on, it may not have a specific starting or end point. In such a case, longer wav files could be used and these parameters will make multiple tensors for training.

Remember that this waveform should not create tensors with absence of the classifier, which is to be detected. However, for keywords, with finite utterance time and specific start and end time, generating multiple tensors from a single file may not be the right approach. Let us take an example of the keyword, *stop*, which is uttered once in a 3 second window, as shown in [Figure 7.12](#). Then this choice will create a few tensors with only silence, which will be labeled as *stop*. This is essentially going to cause *data poisoning*.

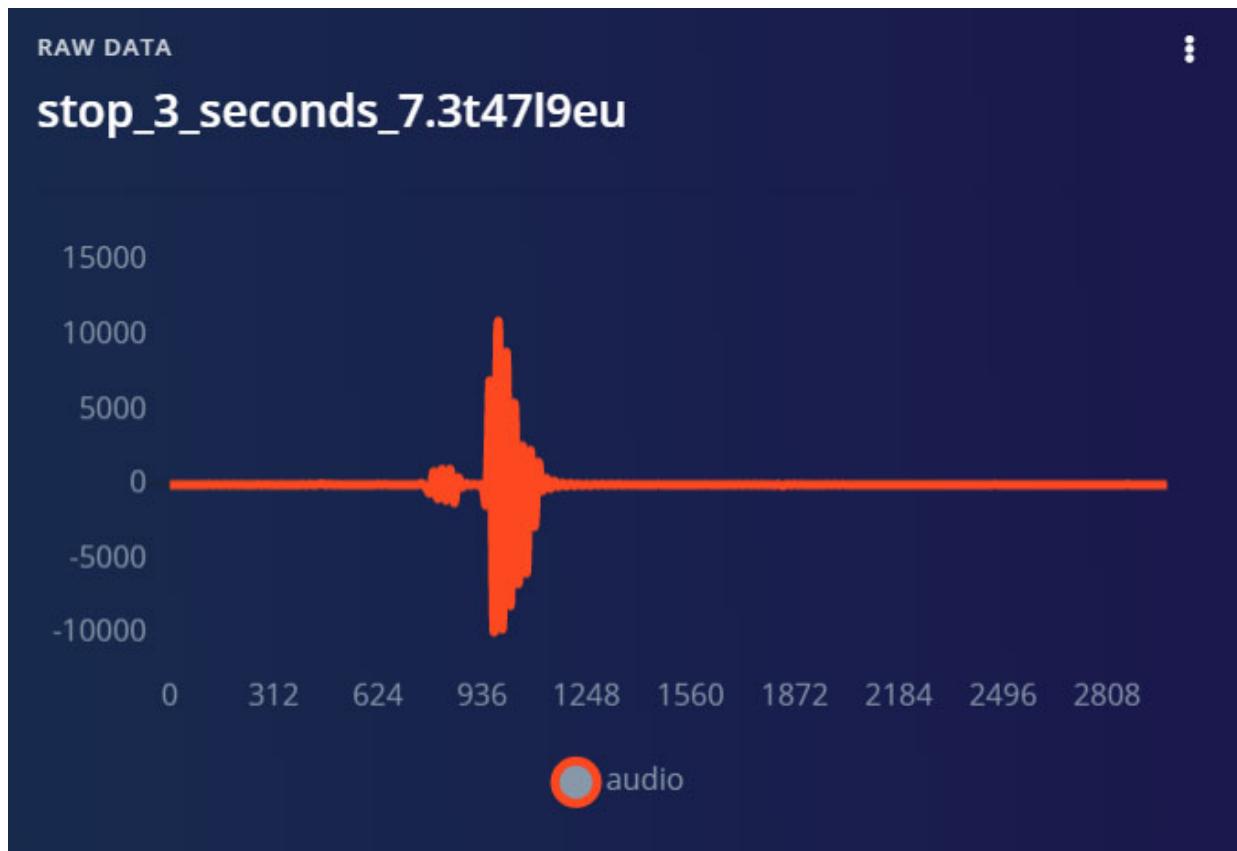


Figure 7.12: One “stop” utterance in 3 second window.

Most of the data in the public Kaggle speech command dataset is about 1000ms long, while the tensor size is only 968ms. This choice will only generate one tensor per sample. This will discard the last 32ms of data. This is still not optimal as we have seen the data is generally present during the end part of the 1s window, with a portion already truncated. This will further truncate it by 32ms.

It is not recommended to check off `zero-pad data` option, in the panel shown in [Figure 7.11](#). This applies to all the wav files shorter than 968ms. The wav files will be discarded when this check box is not selected, thus reducing the training set. If the check box is selected, then the remaining difference in the duration will be padded with 0s. For example, if the wav file is only 800ms long, then the data will be padded with 168ms of zero values at the end. When the model will be deployed, there will always be non-zero noise. This will create deviation between the training set and real use case, which will only make things worse. This should be considered yet another kind of data poisoning.

The next step is to add a processing block. This represents the mel scale conversion. Click on **Add a processing block** and choose **Audio (Syntiant)**, as shown in [Figure 7.13](#). In the NDP101 chip, a dedicated hardware block extracts log Mel-filterbank energy features, from the time domain signal. The conversion is done in a very power efficient manner. The Edge Impulse emulates the same function in software and produces tensors. These tensors are used for the training.

Click on **Add a learning block** to choose the Classification (Keras) block. This selection guides the studio to use classification type of the neural network, where one of the classifiers will be a valid output. A filled panel is shown in [Figure 7.13](#). Click on the **Save Impulse** to save these settings.

Users will notice that the menu has changed in the **Impulse design** sub menu. Users should see two new sub menus, **Syntiant** and **NN Classifier**.

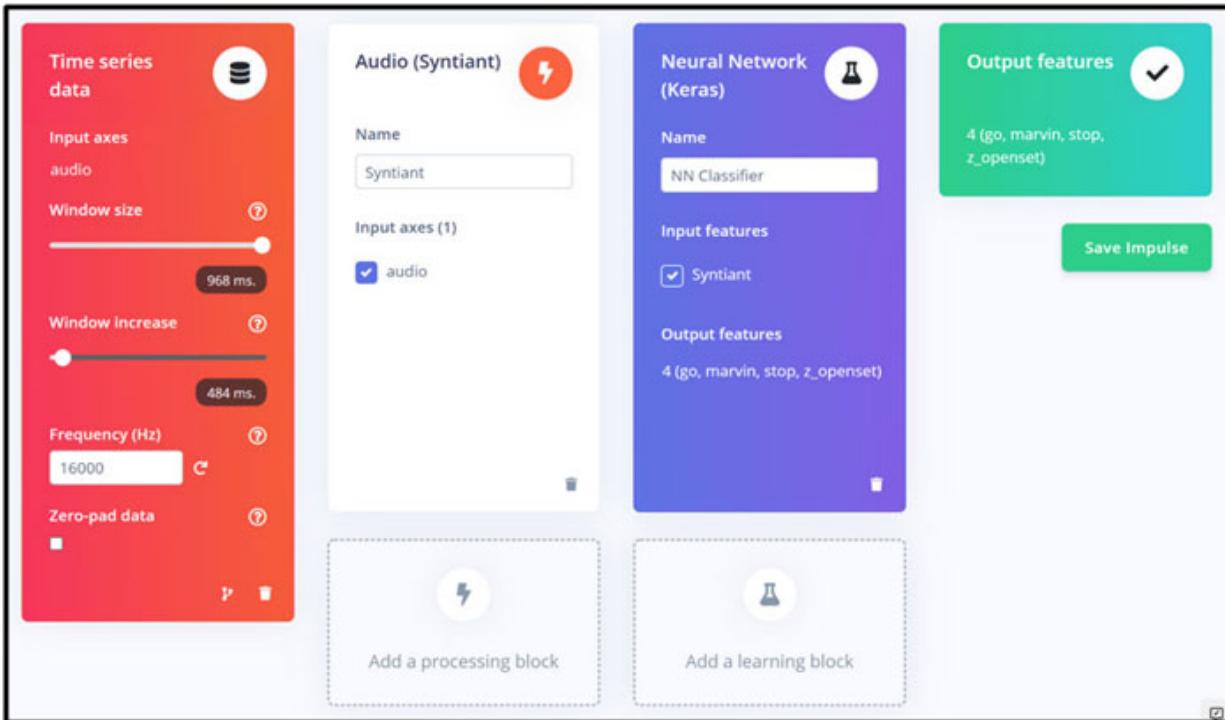


Figure 7.13: Completed Create Impulse panel

Clicking on the **Syntiant** sub menu opens a new panel, as shown in [Figure 7.14](#):

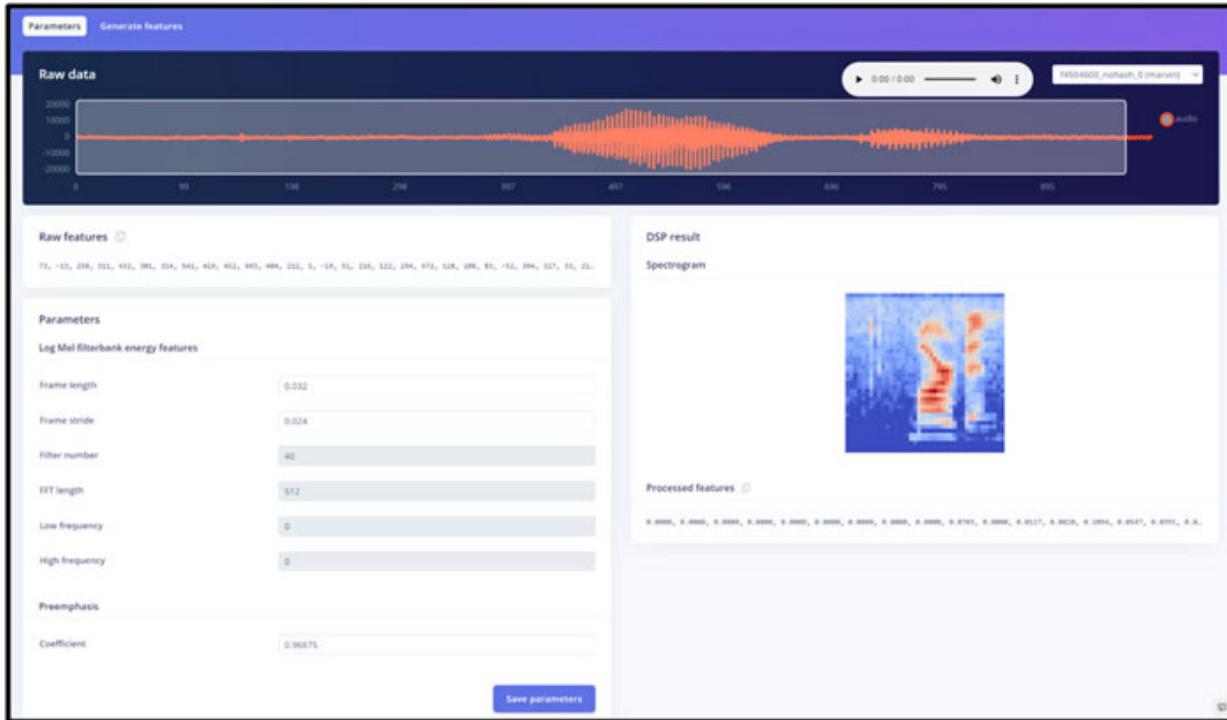


Figure 7.14: Panel opened by clicking on Syntiant sub menu.

This panel shows how the time domain signal is converted to the frequency domain. The parameters are listed in the bottom left corner. These parameters should not be changed as the hardware has fixed values. This gives some idea how the conversion is done. The raw data is plotted on the top side. The data is plotted in time domain; however, the waveform hides some of the details. Users can play the sample by clicking on the play arrow on the very top row.

Different files can be selected to play, using the drop-down menu on the top right corner. On the bottom right side, the user can see the spectrogram. The spectrogram has been described earlier. In essence, a spectrogram is a two-dimensional image, representing a sound file. Once the sound file is converted into an image, it becomes an image recognition problem. It is expected to have similar spectrograms for the same keyword. After staring at these spectrograms, even humans can visually classify the words.

Four spectrograms of the word *Marvin* spoken by different people, are shown in [Figure 7.15](#):

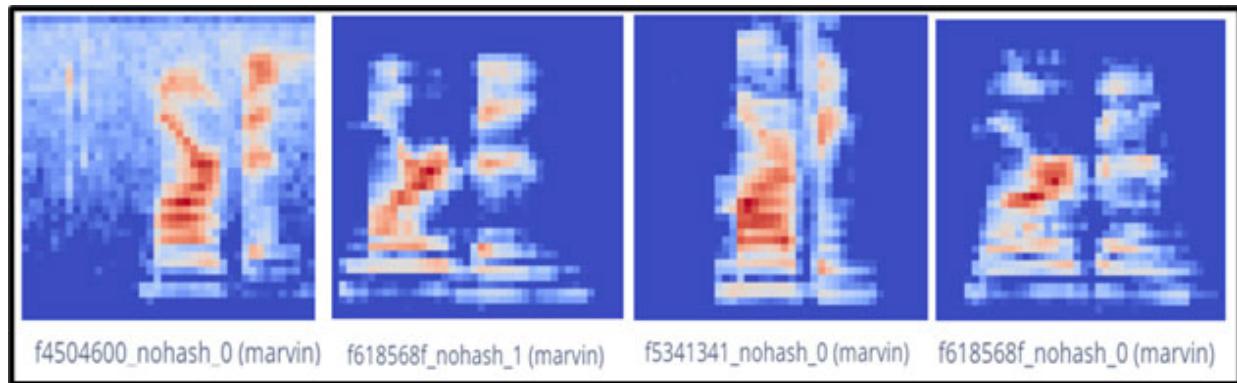


Figure 7.15: Four different spectrograms of word “Marvin” spoken by different people

In the panel shown in [Figure 7.14](#), click on **Save parameters** and then click on **Generate features**. The panel will change to the panel shown in [Figure 7.16](#). On the right side, users will see a two-dimensional plot, which shows how distinct the classifiers are. Each classifier is shown with a different color. If the classifiers are grouped separately, then the classification is a well-defined problem and users can expect very good accuracy. If the classifiers are overlapping with each other, then it could be an ill conditioned problem or difficult problem.

If the users have a choice in selecting the wakeup word, then they should choose the words which are distinct sounding and on this visualization chart they should look different and distinct from each other.

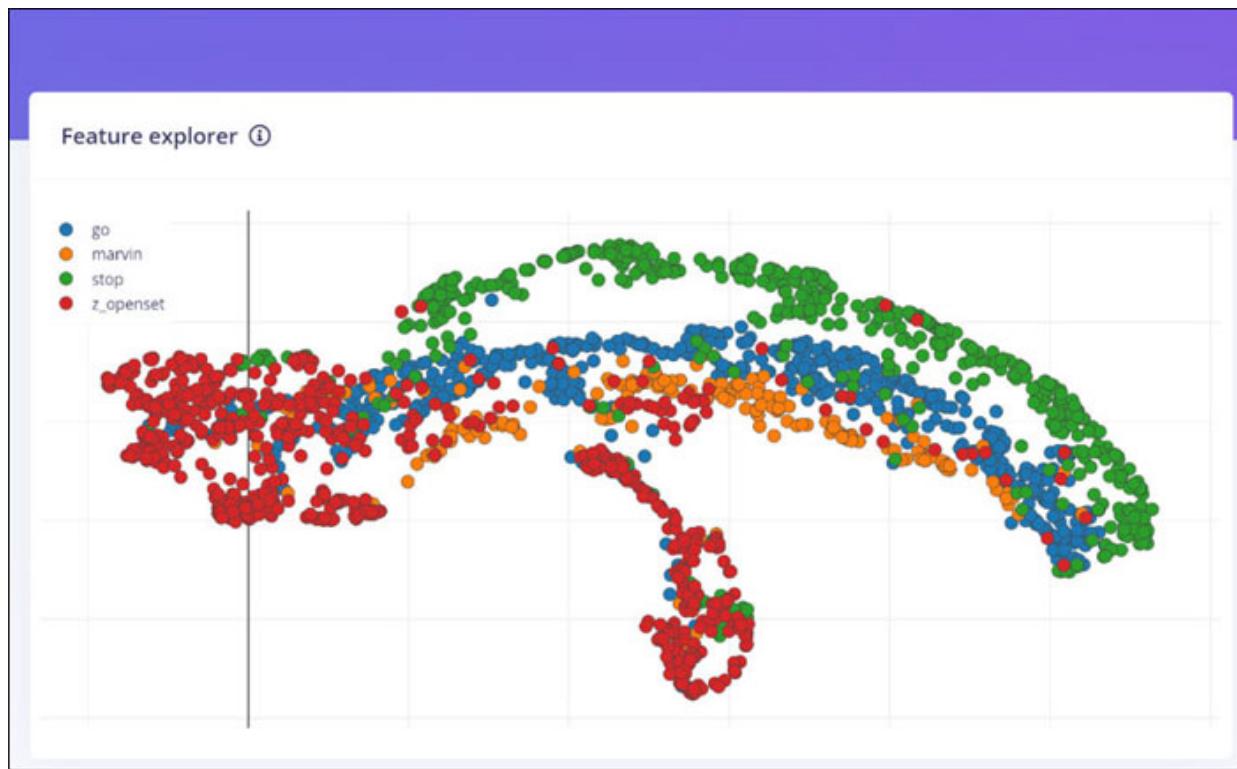


Figure 7.16: Feature explorer panel showing how distinct the keywords are

The next step is training the neural network. Select the **NN Classifier** submenu on the left panel under the **Impulse Design** menu. The panel will change to the one shown in [Figure 7.17](#), showing the definition of the neural network:



Figure 7.17: Definition of the neural network

Epochs Setting

Change the **Number of training cycles** to 100. Number of training cycles are also called epochs in machine learning. By default, it may be set to 30. For production models, the epochs can be set to thousands.

Learning rate setting

Learning rate is a parameter which sets the increments in the variables during the machine learning cycle. If the value is too small, then it will take longer, but convergence will be steady. If the learning rate value is too large, then between the epochs, users will notice the accuracy going up and down and it may diverge. Users can experiment by changing the learning rate and see how the accuracy improves over epochs. The default value is a good starting point.

Validation data set setting

During the machine learning process at each epoch, the model is tested with a data set which is not part of the training set. Given a sufficiently large number of parameters of a neural network, it is possible to overfit and have

100% accuracy for the dataset used for training. This may not necessarily mean that the model is good. Thus, a set of data is randomly set aside which is not part of the training set and is used for testing after every epoch. This set is termed as a validation data set. **validation set size** is specified as a percentage and the default value is 20%. If the validation set is too large, then the training data will be compromised. If the validation set is too small, it may not be representative of the input data. The default 20% is a good starting point. If you remember, we saved some data in the test bucket. That data should not be confused with the validation data set. The test set will be used later for further testing.

Auto balance setting

As discussed earlier, the classifiers should have roughly equal amounts of tensors during the training. 5 to 10% variation is fine. In our case, the data for *Marvin classifier* is exceptionally small. It will have less likelihood of inferring the classifier with less dataset. This selection will copy the data set and balance the dataset. It is not recommended to use this option as this will hide the issue of not having enough variety. Ideally, more data should be captured.

Synthetic data generation can be yet another way to extend the data set.

Data augmentation

Users can enable this option, as shown in [Figure 7.18](#). Start with the default options. Other options could be tried. Ideally, the augmentation should be done in the time domain, while generating real data with different noise backgrounds. The noise background should be chosen such that it is reflecting real life. For example, if the users are expected to be in a traffic noise, then augment the data with traffic noise. The augmentation should be done with varying amounts of noise. We will see the effectiveness of data augmentation in the next chapter. Refer to [Figure 7.18](#):

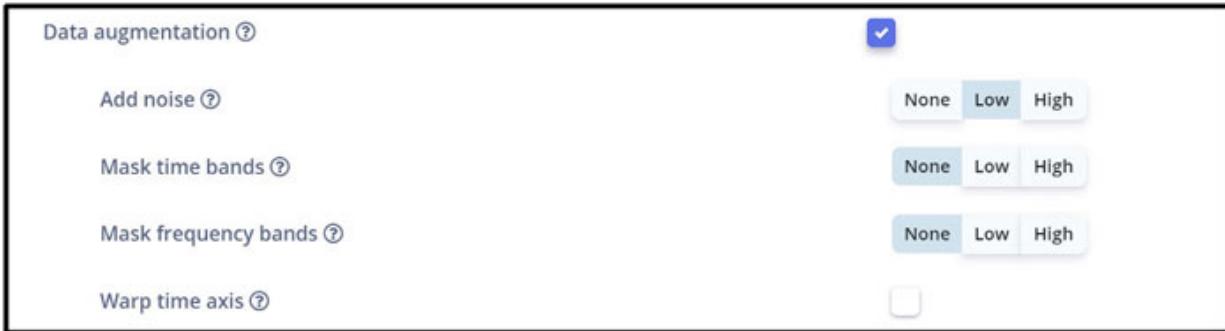


Figure 7.18: Data augmentation selection

For a good model, data augmentation is the second most important thing, while collecting the data is the most important thing. It is the most creative use of the machine learning engineers' talent. Based on the analysis of results, it is expected that the user will find the right methods of augmenting the data. A thoughtless data augmentation will produce a large dataset which will be very compute expensive and still may not produce a good model.

Augmentation methods used in one project may not apply to the other project. This is a field where a lot of tools can be created to guide users to choose the right augmentation methods, while optimizing for compute resources and time.

Neural network architecture

While NDP101 has only one fixed neural network, NDP120 has a lot more flexibility. There are different types of neural network layers, such as, dense neural network, and convolution neural network. There are also multiple choices for the activation layer, such as, ReLu, Tanh and so on. The size of the internal nodes and number of layers can also provide a lot of degree of freedom. With different permutations and combinations, the choices are limitless. However, users will quickly find that most of the neural networks with similar complexity provide very similar results.

From a practical point of view, the author recommends choosing between proven neural networks for a given application type, to save time and money. For a given application along with how data is pre-processed, one type of neural network may be slightly better than the other one.

In most cases, a simple four-layer dense network is good enough and should be the starting point, as a base line. Then, different commonly used neural networks should be chosen, which have a good contrast between number of

parameters, number of operations needed (and thus optimizing power) and number of layers.

Comparing two neural networks with subtle differences may not provide good value for the time and money spent. For the same data, same neural network and same settings, each time that a model is created, the performance will vary due to random starting point.

For a scientific comparison in determining two competing neural networks, users are encouraged to run 30 to 100 runs under the same conditions, and then find the mean and standard deviation of the accuracy. The user should then plot their distribution curves, to see if significant improvement is present or not. The idea is illustrated in [Figure 7.19](#):

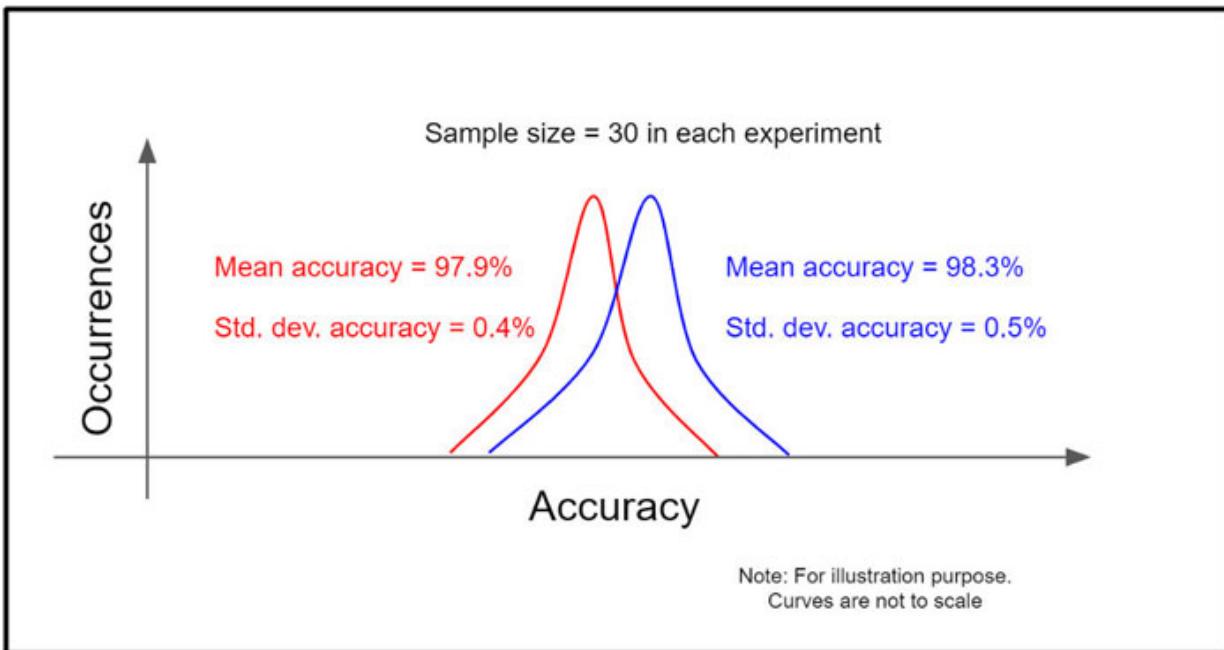


Figure 7.19: Comparing two neural networks while taking into consideration that the starting point of the parameters is a random set of numbers

Iterating multiple cycles for the same conditions may seem like a waste of time, but it is recommended to run a few times in the design cycle of a project. This gives an idea to the users that accuracy is not a fixed value and small improvements should be taken with a grain of salt. It is possible to demonstrate that one neural network is better than the other by some fractional percentage. However, even while using statistical significance methods, it may still be an academic exercise. The validation accuracy reported at the end of the neural network training is for the validation set.

The validation set is randomly picked from the training set. As we know, the training set is only a subset of real use cases, and there is no point in obsessing with the validation accuracy. Since the validation set is chosen randomly, it is expected to be unbiased from the training set, so that it generally has better accuracy than the real world. This assertion is made, realizing that despite the best efforts, the training set does not reflect the real-world use cases.

For a reasonable set of data, choice of neural network and settings, one should expect to see standard deviation of <1% when iterated 30 times under the same conditions. 1% of improvement in the mean should be considered reasonable to switch the architecture.

Neural network training

Click on **Start training** to complete the machine learning. Let us analyze the messages during the training cycles. [Figure 7.20](#) shows the messages. Under the Epoch 1/100 heading, results of the first epoch are printed. Each epoch has been split into 291 steps. The results are printed after completion of all the 291 steps. It is reported that it took roughly 31ms/step, thus taking about 9 seconds of compute for the epoch. Loss and Accuracy mentioned in the first half of the line is for the training set, and it shows 73% accuracy, while the accuracy on the validation set is 86%. At the start, the accuracy varies because the parameters change significantly between the steps. The important thing here is to note that, validation accuracy increased from 86% to 91% in the first three epochs, while the loss reduced from 0.3996 to 0.2660. This is a good start showing steady convergence. Refer to [Figure 7.20](#):

The screenshot shows a terminal window with the title "Training output". In the top right corner, there is a red "Cancel" button. The terminal displays the following text:

```
Scheduling job in cluster...
Job started
Splitting data into training and validation sets...
Splitting data into training and validation sets OK

Training model...
Training on 9384 inputs, validating on 2326 inputs
Epoch 1/100
291/291 - 9s - loss: 0.7245 - accuracy: 0.7310 - val_loss: 0.3996 - val_accuracy: 0.8672 - 9s/epoch - 31ms/step
Epoch 2/100
291/291 - 8s - loss: 0.4499 - accuracy: 0.8433 - val_loss: 0.3043 - val_accuracy: 0.8977 - 8s/epoch - 26ms/step
Epoch 3/100
291/291 - 7s - loss: 0.3699 - accuracy: 0.8721 - val_loss: 0.2662 - val_accuracy: 0.9186 - 7s/epoch - 25ms/step
Epoch 4/100
```

Figure 7.20: Messages shown at the start of the neural network training

[Figure 7.21](#) shows the accuracy and loss results around the 20th epoch:

The screenshot shows a terminal window titled "Training output". It displays a series of messages from epoch 18 to 291. The messages provide training and validation metrics every 7 seconds. The validation loss starts at 0.1841 and decreases to 0.1579. Validation accuracy starts at 0.9350 and increases to 0.9579. Training loss and accuracy are also shown. The terminal has a standard Windows-style interface with a title bar and a scroll bar.

```
Training output
Cancel ▾

Epoch 18/100
291/291 - 7s - loss: 0.1841 - accuracy: 0.9350 - val_loss: 0.1382 - val_accuracy: 0.9570 - 7s/epoch - 25ms/step
Epoch 19/100
291/291 - 7s - loss: 0.1856 - accuracy: 0.9362 - val_loss: 0.1436 - val_accuracy: 0.9561 - 7s/epoch - 25ms/step
Epoch 20/100
291/291 - 7s - loss: 0.1836 - accuracy: 0.9357 - val_loss: 0.1355 - val_accuracy: 0.9592 - 7s/epoch - 25ms/step
Epoch 21/100
291/291 - 8s - loss: 0.1714 - accuracy: 0.9410 - val_loss: 0.1386 - val_accuracy: 0.9566 - 8s/epoch - 26ms/step
Epoch 22/100
291/291 - 7s - loss: 0.1715 - accuracy: 0.9401 - val_loss: 0.1346 - val_accuracy: 0.9583 - 7s/epoch - 25ms/step
Epoch 23/100
291/291 - 7s - loss: 0.1648 - accuracy: 0.9436 - val_loss: 0.1348 - val_accuracy: 0.9592 - 7s/epoch - 26ms/step
Epoch 24/100
291/291 - 7s - loss: 0.1658 - accuracy: 0.9416 - val_loss: 0.1289 - val_accuracy: 0.9600 - 7s/epoch - 25ms/step
Epoch 25/100
291/291 - 7s - loss: 0.1579 - accuracy: 0.9431 - val_loss: 0.1353 - val_accuracy: 0.9579 - 7s/epoch - 25ms/step
```

Figure 7.21: Messages shown around 20th epoch

By the 20th epoch, the validation loss was reduced to 0.1355 from 0.2660, and accuracy improved from 91% to 95%. However, if you look closely, there is hardly much improvement from the 18th epoch to 25th epoch, as we are reaching the law of diminishing returns. As a matter of fact, the validation accuracy and loss are somewhat fluctuating around the terminal value. This is not necessarily a divergence, but simply shows that the convergence is almost reached.

By the 100th epoch, as shown in [Figure 7.22](#), the improvement is still measurable:

The screenshot shows a terminal window titled "Training output". It displays a series of messages from epoch 94 to 291. The validation loss starts at 0.0917 and decreases to 0.0876. Validation accuracy starts at 0.9690 and increases to 0.9678. Training loss and accuracy are also shown. The terminal has a standard Windows-style interface with a title bar and a scroll bar.

```
Training output

291/291 - 7s - loss: 0.0917 - accuracy: 0.9690 - val_loss: 0.1349 - val_accuracy: 0.9708 - 7s/epoch - 25ms/step
Epoch 94/100
291/291 - 7s - loss: 0.0939 - accuracy: 0.9696 - val_loss: 0.1407 - val_accuracy: 0.9652 - 7s/epoch - 24ms/step
Epoch 95/100
291/291 - 7s - loss: 0.0830 - accuracy: 0.9709 - val_loss: 0.1432 - val_accuracy: 0.9712 - 7s/epoch - 25ms/step
Epoch 96/100
291/291 - 7s - loss: 0.0842 - accuracy: 0.9704 - val_loss: 0.1418 - val_accuracy: 0.9669 - 7s/epoch - 25ms/step
Epoch 97/100
291/291 - 7s - loss: 0.0969 - accuracy: 0.9664 - val_loss: 0.1359 - val_accuracy: 0.9690 - 7s/epoch - 26ms/step
Epoch 98/100
291/291 - 7s - loss: 0.0885 - accuracy: 0.9701 - val_loss: 0.1403 - val_accuracy: 0.9647 - 7s/epoch - 25ms/step
Epoch 99/100
291/291 - 7s - loss: 0.0960 - accuracy: 0.9673 - val_loss: 0.1312 - val_accuracy: 0.9690 - 7s/epoch - 25ms/step
Epoch 100/100
291/291 - 7s - loss: 0.0876 - accuracy: 0.9708 - val_loss: 0.1333 - val_accuracy: 0.9678 - 7s/epoch - 25ms/step
Finished training
```

Figure 7.22: Messages showing at the end of 100th epoch.

The Edge Impulse studio does not show the convergence curves, but we have post processed the data, specifically the curves, to plot the trajectory. [Figure 7.23](#) shows the change in the loss values. As expected, for the training set, the loss continues to reduce. Both loss of the training set and validation set are plotted. The loss of the training set is less relevant as it will continue to get better and could be described as over-fitted. For the validation set, the loss reaches terminal value by the 40th epoch and beyond that, the loss remains almost flat. The validation set's loss is considered in evaluation of the quality of convergence. Refer to [Figure 7.23](#):

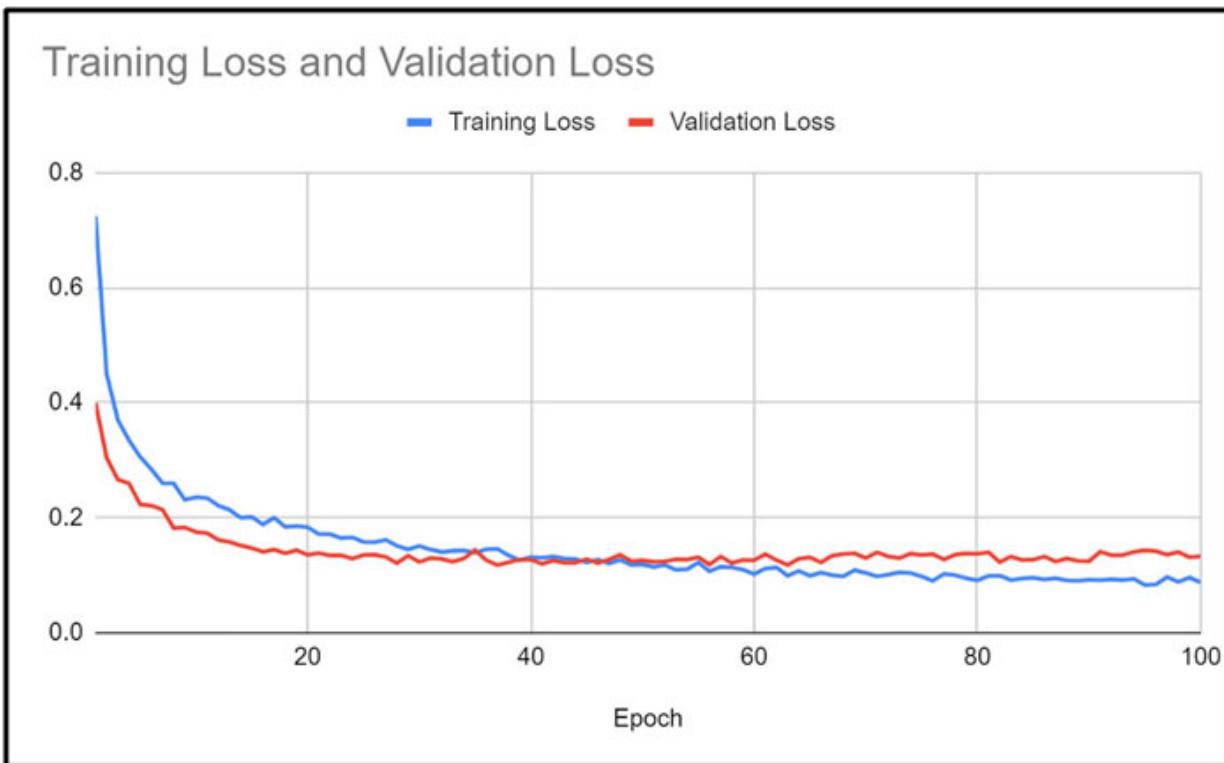


Figure 7.23: Reduction in loss value as a function of successive epochs during training

[Figure 7.24](#) shows the accuracy improvement over epochs:

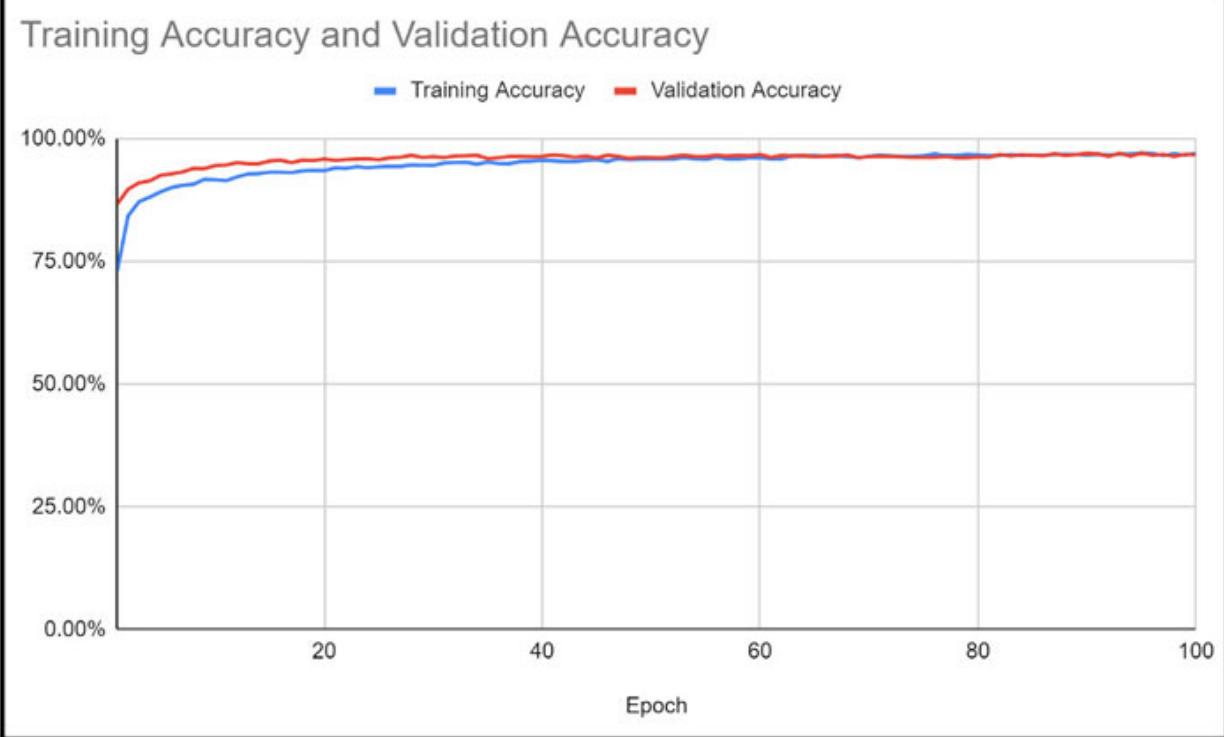


Figure 7.24: Reduction in loss value as a function of successive epochs during training

It is hard to see improvement in the accuracy on this scale, so percent error (100% accuracy) are plotted in [Figure 7.25](#):

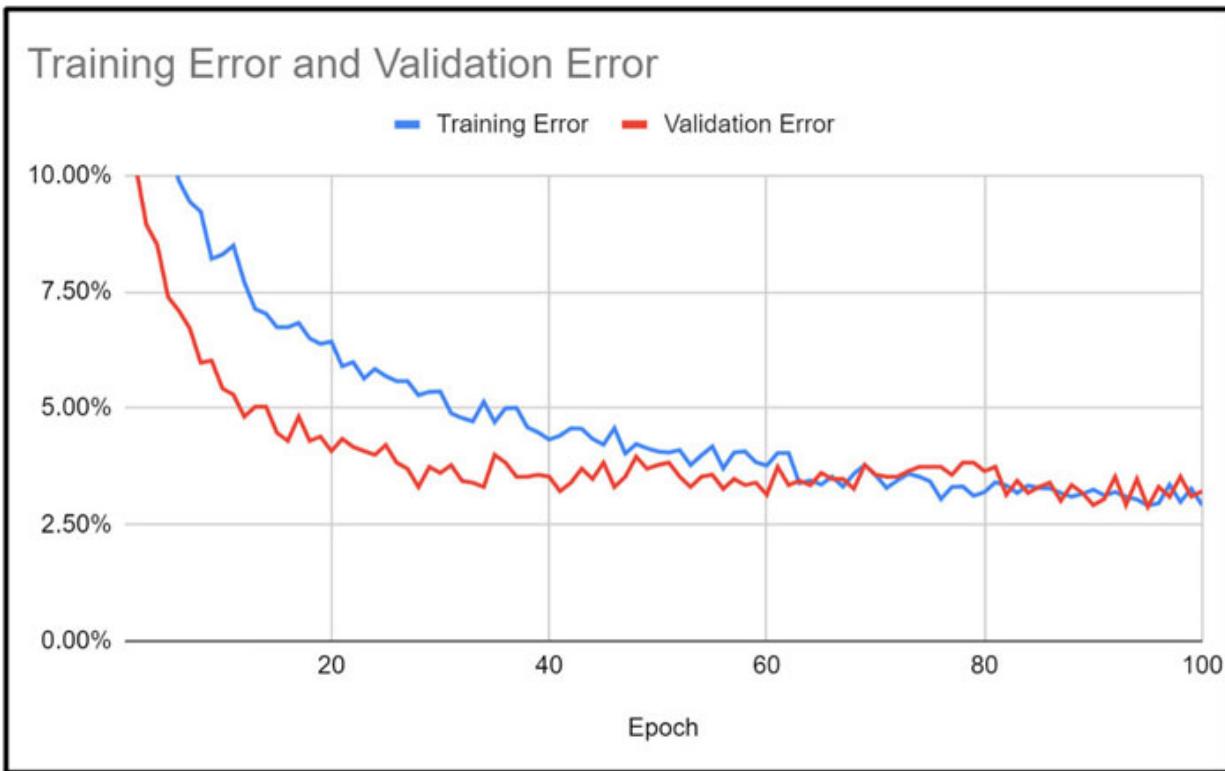


Figure 7.25: Error plotting instead of accuracy to see the improvement clearly

[Figure 7.25](#) zooms into the error. It seems like the errors were continuously improving but got stable between 80 and 100 epochs. The number of epochs it takes to achieve terminal values, depends on the variation in the data set, size of the training set, number of classifiers and so on. It is recommended to study the convergence for each application, as it cannot be generalized between different type of applications.

Now let us look at the confusion matrix shown in [Figure 7.26](#). The overall accuracy reported is 96.5% while loss is 0.16. “Marvin” has the worst accuracy of 92.9%, most likely due to a small sample set. “Go” is most likely to get maximum false alarm (2.8%, 1.6% and 3.1% respectively for “Marvin”, “Stop” and openset utterances) due to its small duration. Overall, the performance is better than 95%, so we will proceed to the next steps. However, this is not an acceptable production level performance. Refer to [Figure 7.26](#):



Figure 7.26: Confusion matrix results

There is another graph generated within the Edge Impulse studio (shown in [Figure 7.27](#)) on the bottom right corner, which shows the separation of the classifiers on the two axes. This is an interactive tool where one can click on the incorrect classification and hear the sample. This may give ideas why a particular tensor is misclassified. Refer to [Figure 7.27](#):

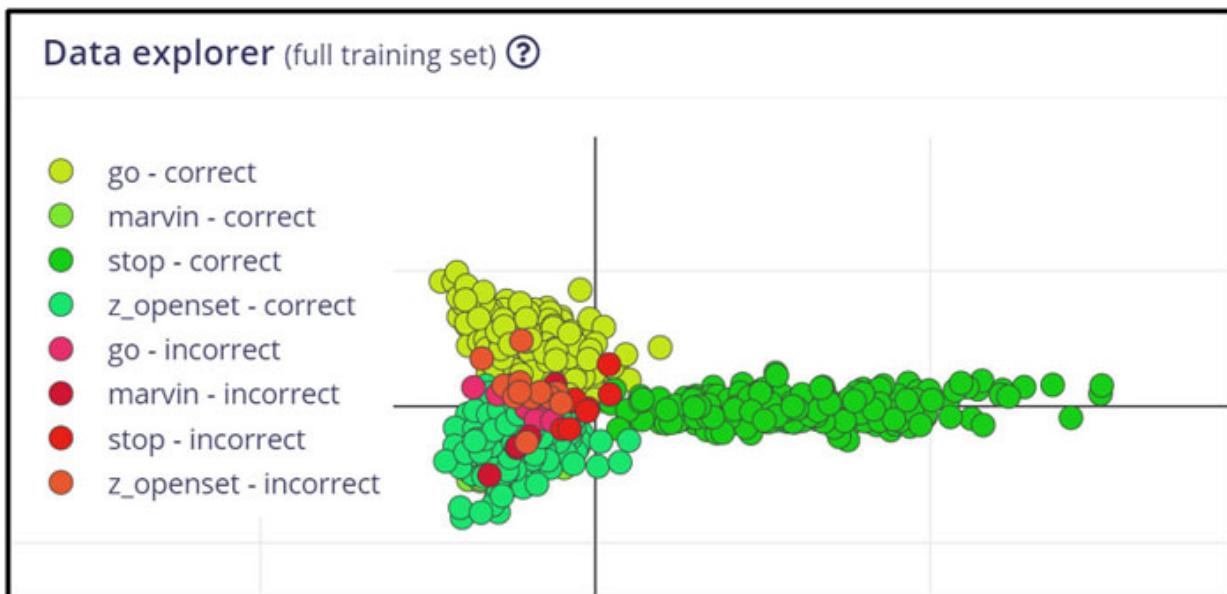


Figure 7.27: Interactive tool to see which wav files are misclassified

Model testing

The validation set gives one measure of accuracy where the validation set is randomly picked from the training set. Another set for testing could be used within the Edge Impulse studio. In our experiment, 100 samples were separated from the sanitized set for *go*, *marvin* and *stop* keywords. For the open set, 20 samples were chosen from utterances of five, six, seven, eight and nine. Let us see how the model fares against the model testing, as shown in [Figure 7.28](#):



Figure 7.28: Model testing using the testing bucket defined in the Edge Impulse studio

For *Go*, the accuracy was reasonable to about 90%. The uncertain category can be managed in the filter running over multiple consequent inferences. *Marvin* keyword had a bit of a poor performance, most likely due to the smaller set. *Stop* keyword had very good performance. Open Set with numbers uttered between five and nine had the worst performance. They had significant false acceptance for *Go* (~17%), *Marvin* (12%) and *Stop* (~12%). This implies that we need to put more of these words (five ... nine) in the openset.

However, if the next time someone may test for one to four, will we see the same performance? What about other keywords in the Kaggle speed command list, for example, bed, backward, forward and so on?

In real life, there will be more words which never made it to the open set.

This may seem like a never-ending job. As a matter of fact, this is the most difficult problem of machine learning. We will see how we can handle this

issue in the next chapter.

With this known limitation, we will further proceed and deploy the model to the TinyML board.

Deployment

To deploy the model, click on **Deployment** item on the left menu. A new panel will pop, as shown in [Figure 7.29](#):

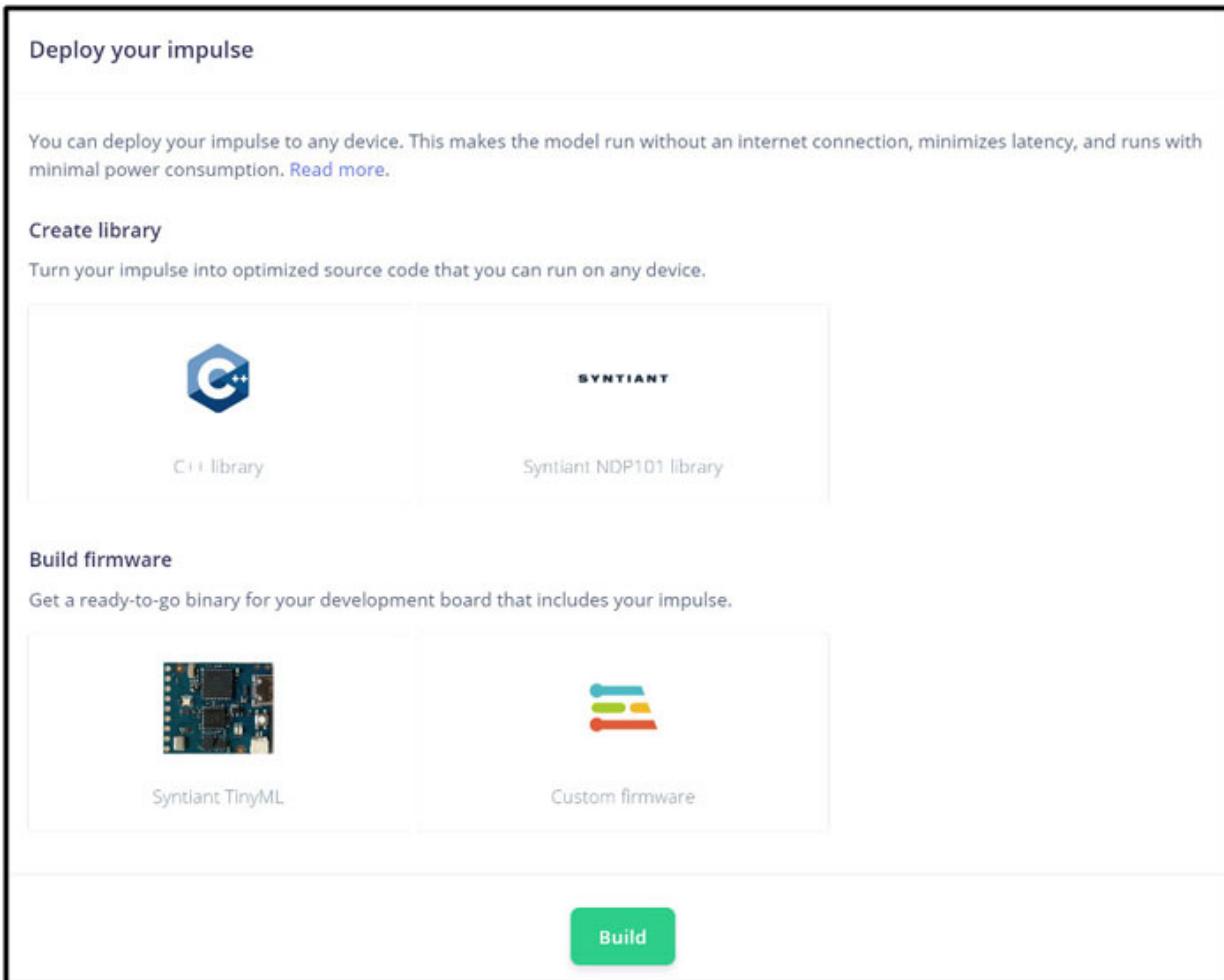


Figure 7.29: Deployment panel in the Edge Impulse studio

Click on the bottom left corner at the Syntiant TinyML board. More options will show up, as shown in [Figure 7.30](#):

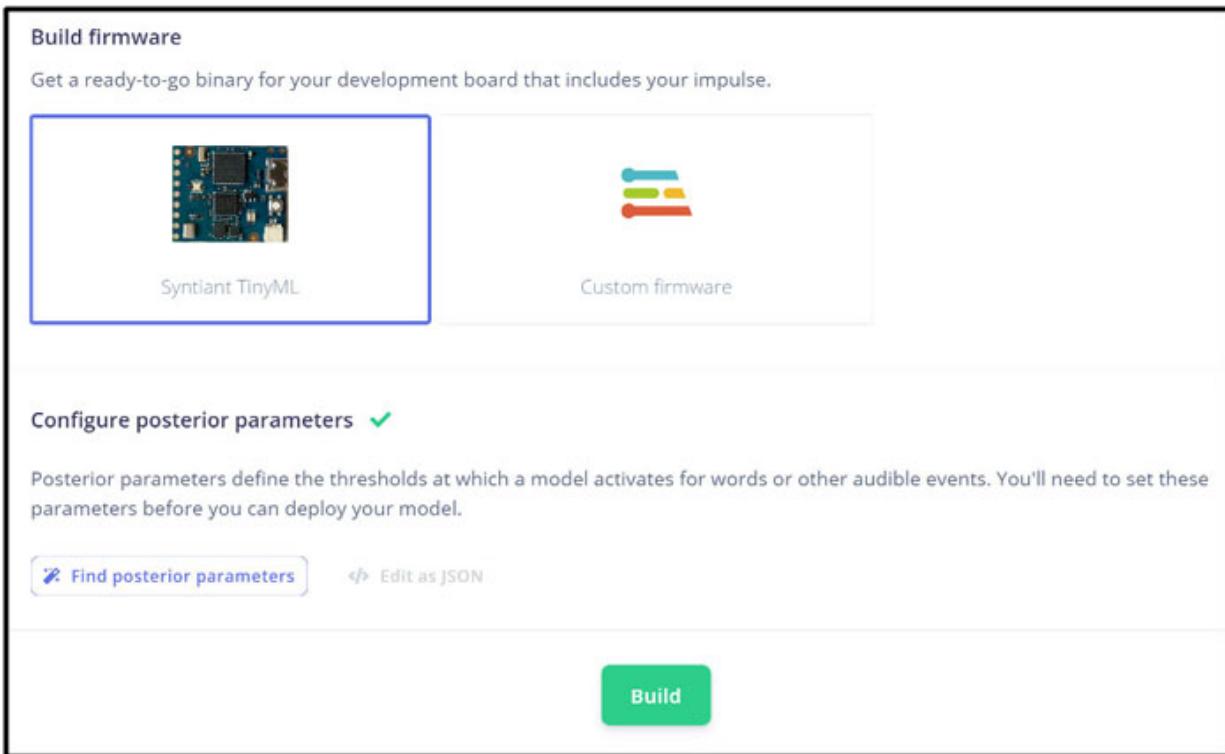


Figure 7.30: Finding posterior parameters for the deployment

Click on **Find Posterior parameters** in the bottom left corner. This will open a new panel, as shown in [Figure 7.31](#). Select all the target keywords except for the open set (`z_openset`). Then click on the **Find parameters** button. This will start a process of finding posterior parameters. Refer to the following figure:

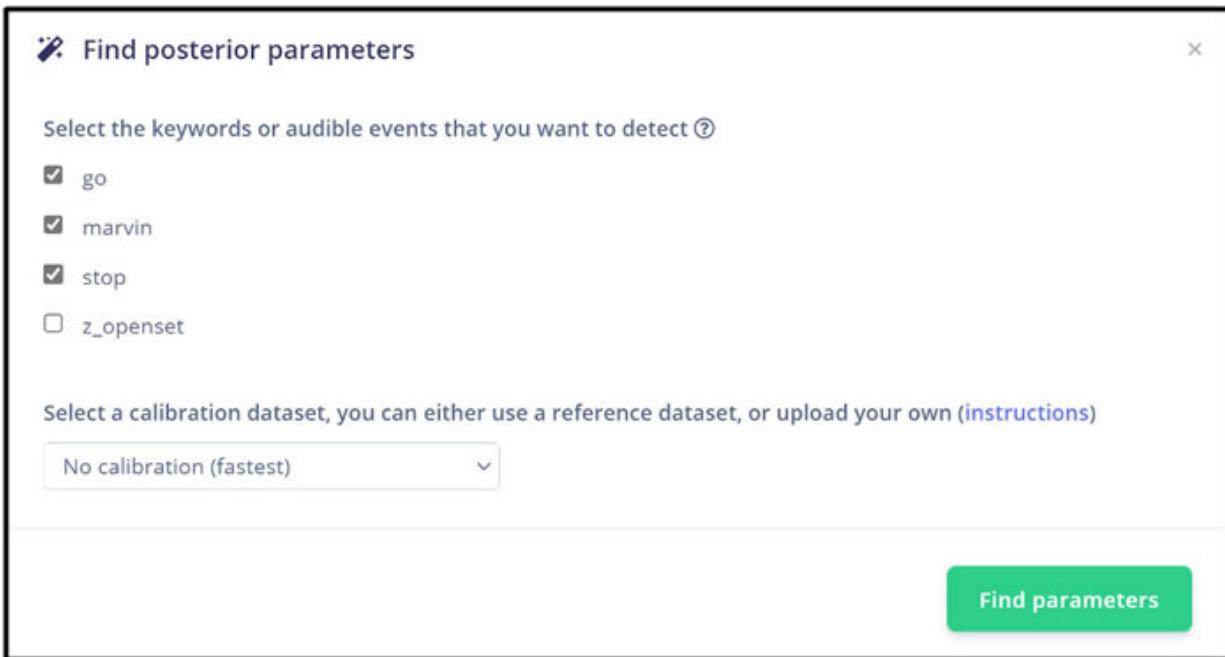


Figure 7.31: Choice of keywords for detection

Once the job is completed, close the pop-up panel. Now click on `</> Edit as JSON` which seems like a ghosted text written in light gray color. This will open a panel where you can edit the JSON file, which will be used to change some of the parameters.

Figure 7.32 shows that six parameters can be optimized for each classifier individually. `phaction` and `phaction_arg` are for advance usage and will not be discussed here. Leave their default values. Let us discuss the remaining four parameters.

The screenshot shows a JSON configuration for classifier states. The code is as follows:

```

1  {
2    "states": [
3      {
4        "timeout": 0,
5        "classes": [
6          {
7            "label": "go",
8            "phwin": 5,
9            "pht": 0.7,
10           "phbackoff": 20,
11           "phaction": 0,
12           "phaction_arg": 0,
13           "smoothing_queue_size": 1
14         },
15        {
16          "label": "marvin",
17          "phwin": 5,
18          "pht": 0.7,
19          "phbackoff": 20,
20          "phaction": 0,
21          "phaction_arg": 1,
22          "smoothing_queue_size": 1
23        },
24        {
25          "label": "stop",
26          "phwin": 5,
27          "pht": 0.7,
28          "phbackoff": 20,

```

A green "Save" button is located at the bottom right of the editor.

Figure 7.32: The optimized parameters can be reviewed and changed by clicking on </> Edit as JSON

smoothing_queue_size: This is the running averaging window with max value of 11. This parameter averages the softmax values produced for the given classifier. Each time a new inference is evaluated, the running window moves up.

pht: This is the threshold to trigger the internal counter which will be used for the **phwin** setting. The averaged value generated by the **smoothing_queue_size** filter, is compared against the **pht** threshold. If the **smoothing_queue_size** filtered value is less than the threshold, the counter is reset to zero value, each time. If the **smoothing_queue_size** filtered value is higher than the threshold, the counter is incremented by one, each time.

phwin: Once the counter equals or exceeds the **phwin** setting, the NDP chip triggers the interrupt signal on the dedicated interrupt pin. The host can read the SPI register to figure out which classifier triggered the interrupt signal. In low power systems, it is expected that the host is in sleep mode and the change in the logical value on the interrupt pin wakes up the host. This saves the power. A continuous polling would mean that the host has to be either up

all the time or frequently wake up, which will waste a lot of system power. The state machine is designed such that the interrupt is generated only once when the `phwin` condition is met. If the same condition persists and the counter continues to rise, more interrupts are not generated.

phbackoff: This parameter prevents any other classifier from getting triggered during the number of inferences specified here. Twenty inferences are about half of a second. For this setting, there will be no other classifier that will get triggered for the next half a second.

By the time the book is written, the optimization in the Edge Impulse studio is minimal and so, most often it comes with similar values. In the next chapter, we will see how these parameters can be optimized.

There is one more parameter that can be set in the JSON file. If users scroll down, they will see a setting, `audio_pdm_gain`. By default, it is 12. This can be changed between 8-15. It is reduced by 1 each time, and the gain of the microphone reduces by a factor of 2. The lower settings are good for detecting loud sounds, such as a gunshot or the breaking of glass. Similarly, by increasing it by 1, the gain is doubled, which is useful for detecting faint signals. Due to the large gain, users may see more noise and saturation of the signal.

Refer to [Figure 7.33](#):

The screenshot shows a JSON editor window titled "Set posterior parameters". The code is as follows:

```
37     null,
38     [
39         "uniform",
40         4,
41         "min"
42     ],
43     null,
44     [
45         "uniform",
46         4,
47         "min"
48     ],
49     null,
50     [
51         "uniform",
52         4,
53         "min"
54     ],
55     null,
56     [
57         "uniform",
58         4,
59         "max"
60     ],
61     null
62 ],
63 "audio_pdm_gain": 12
64 }
```

A green "Save" button is located at the bottom right of the editor.

Figure 7:33: Changing microphone gain setting by clicking on </> Edit as JSON

For now, save the json file without making any changes. Then click on the **Build** button at the very bottom of the panel. The process will be started, and will finish with the following screen message, shown in [Figure 7.34](#):

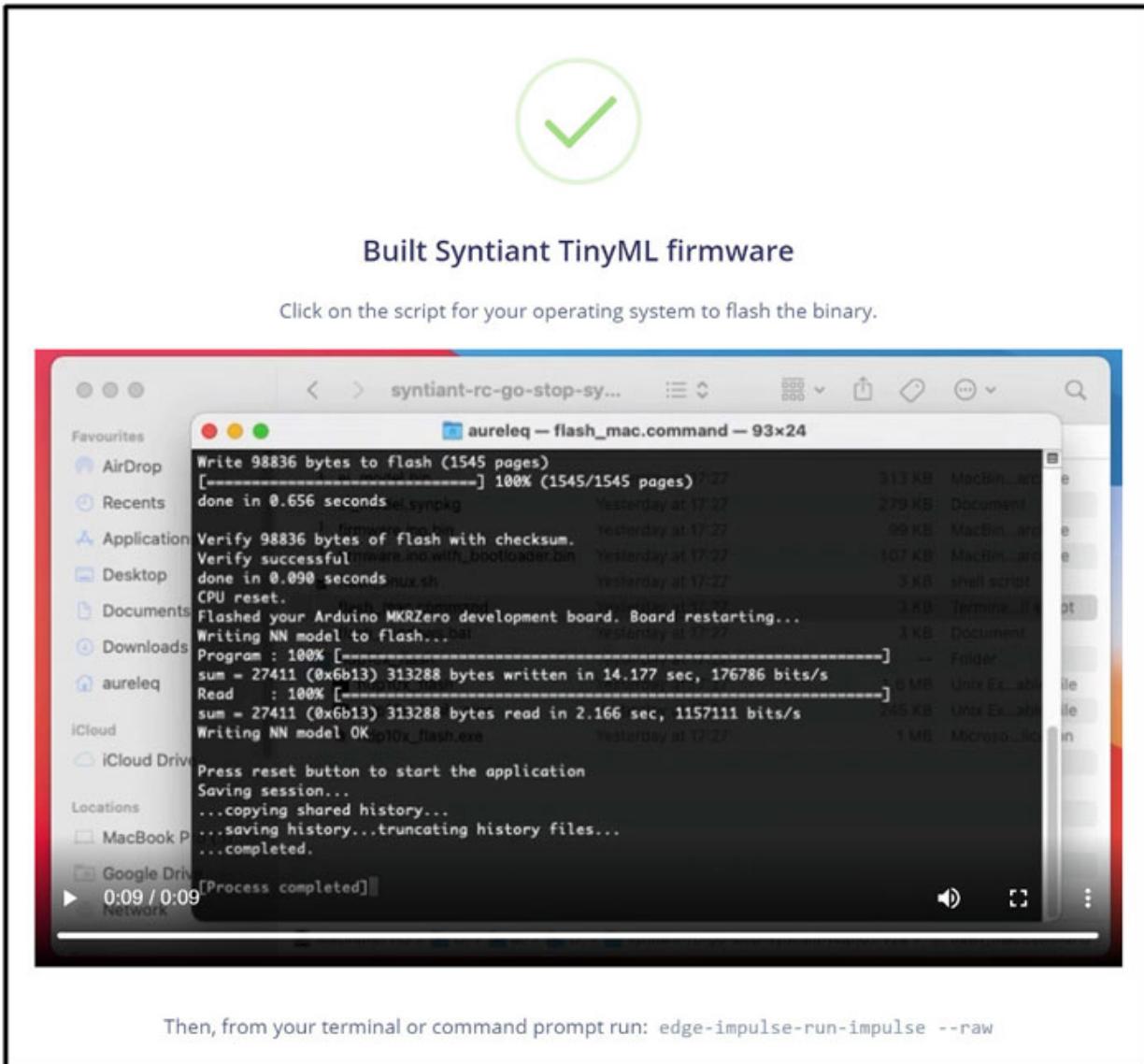


Figure 7.34: Generation and downloading of the zip file for the flashing the board

You will also notice that a new file is saved in your download directory. Follow the Windows or Mac tutorial provided in the www.syntiant.com/TinyML link. Scroll down to see the tutorial.

Now you can test the model. It will blink green for *Go*, and it will blink red for both *Marvin* and *Stop* keywords. The classifier can be seen by connecting it to the Serial Monitor of Arduino IDE.

In a quiet background, it will respond reasonably well. Significant false alarm will be observed. We will see how to characterize and improve performance in the next chapter.

Nicla Voice board can be programmed in the similar manner. Follow tutorials from Edge Impulse and Arduino to download the model into the board. There may be small differences in the flow which will be covered in the tutorials specifically composed for the Nicla Voice board.

Conclusion

In this chapter, an AI project is built from scratch. The project is built using a publicly available data set. The project is built on the free of cost developer license. All the steps have been described in this chapter, such that users can appreciate the rationale behind the step. Once the project is completed, it is downloaded on the TinyML board and tested in real life. The board is able to recognize the “go”, “Marvin” and “stop” keywords, when there is no background noise. Significant false alarms are observed. We will see how to deal with false alarms in the next chapter. We will also see how to make the model robust against the background noise”.

Key facts

- Typically, more than 2000 tensors are required per classifier for a prototype AI model.
- Users can write a Python script running TensorFlow. However, free of cost GUI studios are available for quick start.
- A simple four layer fully connected (dense) neural network is sufficient for complex keyword detector problems. It is recommended to start with a simpler neural network before trying complex neural networks.
- The convergence should be monitored during the training epochs.
- Confusion matrix should be analysed before proceeding.
- For a small set of data, do not expect production level performance in the first try.
- For production level performance, multiple iterations may be required.

Questions

1. For a deep neural network from scratch, 100 samples are more than sufficient per classifier for production level performance. True or

False?

2. Simple models using four to six layers of dense and/or convolution layers, are useless for complex applications such as keyword detection.
True or False?
3. 30 epochs are sufficient for all types of neural works and applications.
True or False?
4. Data science and machine learning are not interrelated. True or False?
5. Production models can be generated without much need of data or compute resources. True or False?
6. Machine learning is an iterative process. True or False?

References

1. **Speech_commands_v0.02 | Kaggle**
2. <https://www.edgeimpulse.com/>
3. <https://www.syntiant.com/tinyml>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 8

Continuous Improvement

Introduction

By now, you would have made a complete system and tested it. It is not surprising that the accuracy results may not have been as good as one would have hoped for. There are multiple steps that can be taken to improve the system level performance. We will discuss these steps one at time and with few examples, we will show how system level performance can be improved.

Structure

In this chapter, the following topics will be covered:

- Expectation gap
- Unique issues with audio application
- Handling anomalous behavior during target classifier testing
 - **Method 1:** Running window averaging
 - **Method 2:** Enriching target classifier
 - **Method 3:** Enriching open set classifier
- False Acceptance Rate testing
- Optimization of window size in running window averaging
- Phrase recognition constraints to improve system level performance
- FRR Testing under noisy conditions
- Improving FRR performance under noisy conditions
- Data collection for continuous improvement

Objectives

In this chapter, we will learn how to bridge the gap between what raw performance we get from neural networks and what is expected at the system level. It will be shown that the raw performance of the neural network is roughly off by a factor of 100,000, than what is expected by the end customer. However, it is possible to add a few steps to attain the expected system level performance. These steps recommended here are based on the traditional style of analyzing and writing code. Readers are encouraged to find equivalent solutions using artificial intelligence concepts.

Expectation gap

We may be expecting that the recognition of keywords will be better or similar to what we expect from a common person. Due to the fear of overfitting, most machine learning experts like to see an accuracy between 95% to 99%, even during training. In some cases, even 99% accuracy or 1% error rate is not good enough. The expectation gap needs to be bridged for AI technology to be widely successful. We will take the keyword detection example and show what we get from neural networks and what is expected.

Most people start their AI journey with image processing. For example, they learn how to classify if a cat or a dog is present in an image. An image is a finite amount of information which has well defined boundaries. It is a natural tensor which can be fed into a neural network for an inference. Audio/speech is a little bit more difficult. In the next section, we will see some nuances of speech recognition.

Unique issues about audio application

Unlike an image, an audio stream is an infinite stream of data with no particular start or end. During this stream, a keyword could be spoken without any demarcation. During training, we fed the neural network with audio clips of 1 second each. The keyword was present in that 1 second frame. In real life deployment, there is no pre-processing, which cuts out that 1 second of the frame when we expect a keyword. As a matter of fact, the sole purpose of the AI engine here is to continuously analyze the audio data and figure out when a keyword is spoken. It is possible to use traditional methods to find the envelope and carve out 1 second frame for inferences. This methodology can work in an environment where noise is not present. But when noise is present, the envelope method may not work. So, let us see

how the AI system reacts to a complete phrase spoken in about 3.5 seconds of the segment. For this illustration, the background noise was kept very low. Later in the chapter, we will see how to deal with noise.

Figure 8.1 shows the time domain waveform of the spoken phrase “Marvin go”:

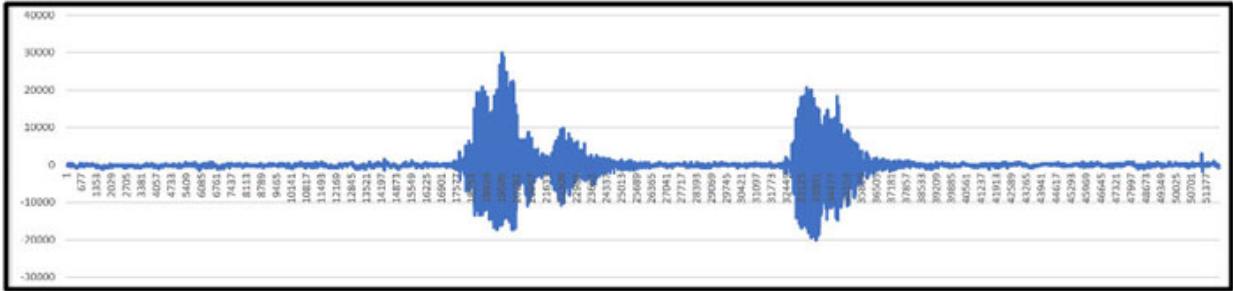


Figure 8.1: Time domain waveform of the spoken phrase “Marvin go”

The inferences are processed at every 24ms with a 968ms long frame. In a 5 second time frame, we will expect $3.5/0.024 = \sim 145$ inferences. Since the words “Marvin” and “go” are spoken only once in this time frame, we should expect one and only one assertion for these two key words.

Figure 8.2 shows the expected results. We will see that the neural network does not provide what is expected. In the next section, we will study what raw information we get from the neural network.

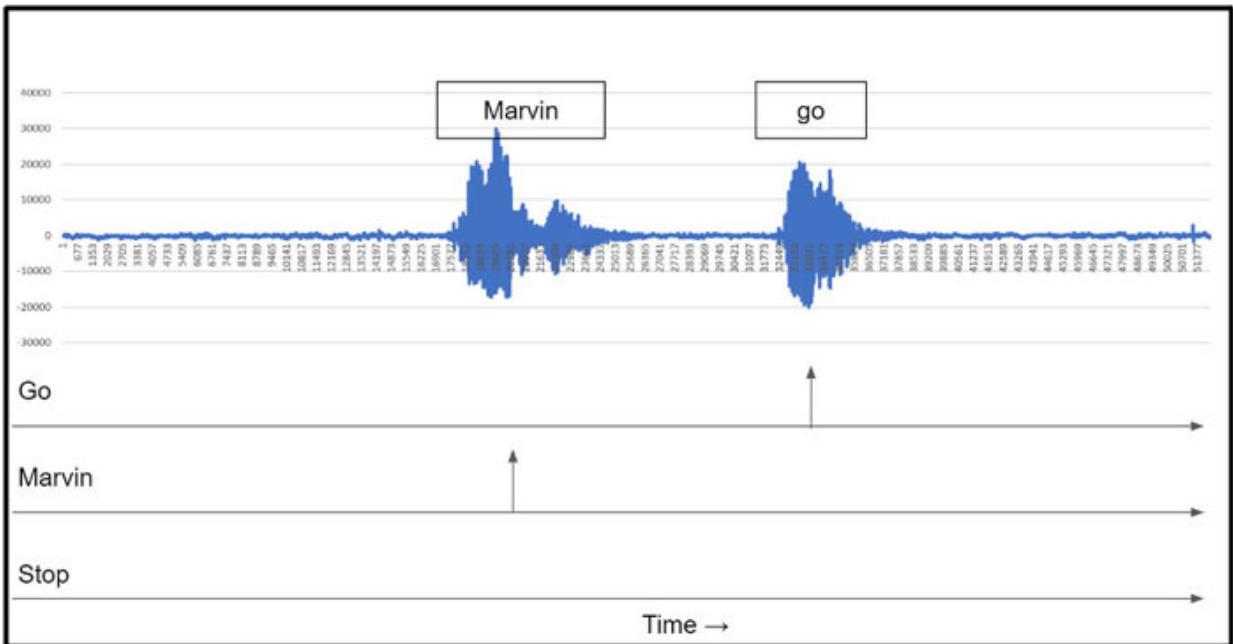


Figure 8.2: Expected inferences for the “Marvin go” phrase

Raw neural network output and softmax transformation

The raw value coming out from the chosen neural network implementation, are unsigned 8 bits (0-255). The softmax function is then applied, which forces only one of the classifiers to be the winner. The full-scale value is chosen to be unsigned 16 bits, (0-65535). With 4 classifiers, after the softmax transformation, the 0 value becomes 22 and 255 becomes 65464. For 0,0,0,255 classifier values, the equivalent softmax values are 22, 22, 22 and 65468.

Figure 8.3 shows the actual classifier results after the softmax processing. The summation comes out to be 65534, which is a little less than 65535, due to the truncation errors. We could normalize the softmax values with dynamic range between 0 and 1. However, this will generate a floating-point value and will be inefficient for processing in firmware. Thus, we are keeping the value at the integer levels. Please refer to the following figure:

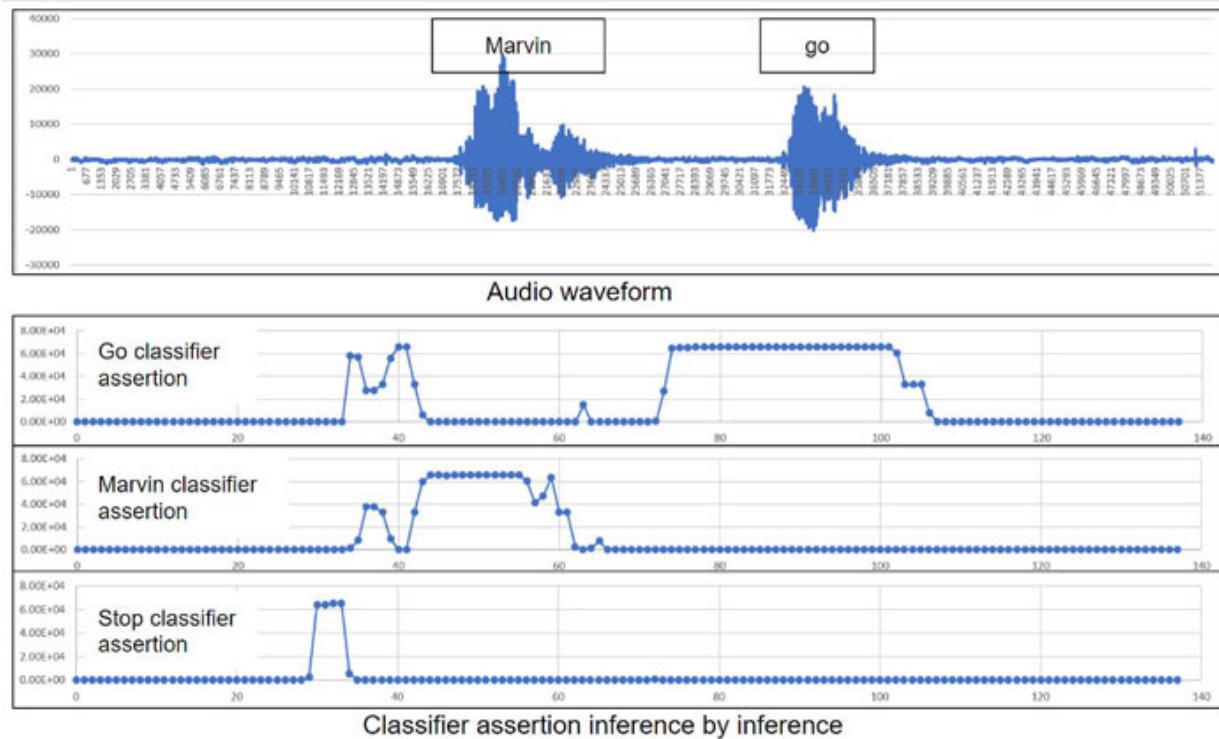


Figure 8.3: Actual softmax values inferred at every 24ms

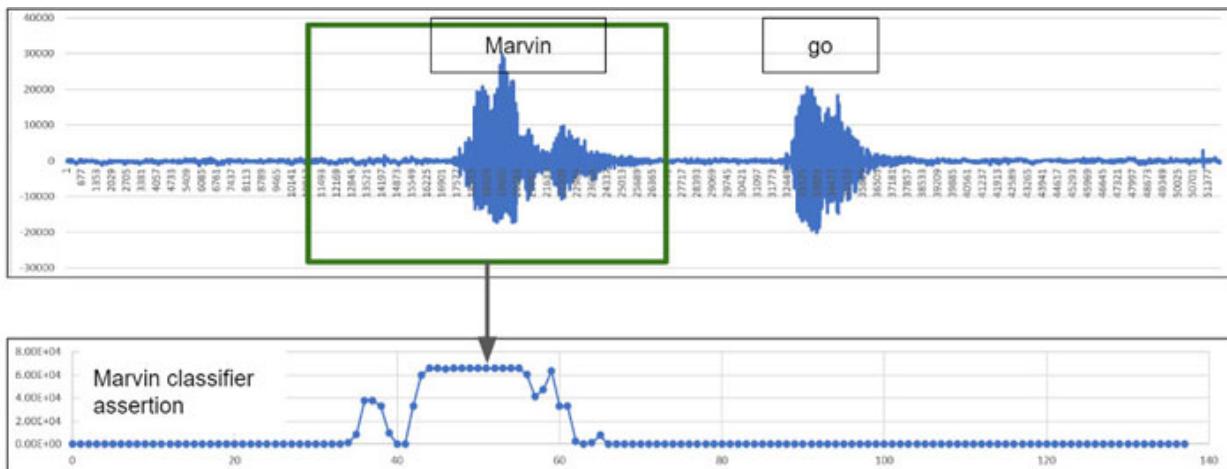
There are few surprises. There are many inferences which predict the correct classifier in the vicinity where the keywords (that is, “Marvin” and “go”) are

spoken. Between a 1.5 and 2.5 second time frame, the “go” keyword was predicted correctly 28 times. Similarly, the keyword “Marvin” was predicted correctly roughly 17 times. It is a good surprise that a correct prediction was made more than one time. This acts as redundancy which will provide robustness. We can reduce the expected number of correct inferences by aligning the data.

The ugly surprise is that the keyword “go” is inferred before “Marvin” is inferred. This is incorrectly predicted about 5 times. Similarly, “stop” was inferred 4 times when indeed it was not even spoken in this test phrase. So, if we transcribe this phrase, it will be “*stop stop stop stop go go Marvin Marvin Marvin go go go Marvin Marvin ... Marvin ... go go ... go*” which is obviously not correct. The most unexpected behavior we found is the presence of “stop” and “go” classifiers before the “Marvin” classifier was detected. In the next section, we will see the root cause of the behavior and how to handle it.

Handling anomalous behavior during target classifier testing

Figure 8.4 shows the green frame placed at the instant when the “Marvin” keyword is well centered in the frame, just like the way training frames were used. All the training set containing the keyword “Marvin” was aligned to be centered in the frame. The neural network predicts the right inference at that time:



A frame of 15488 samples, is inferred correctly as “Marvin” when it is well centered

Figure 8.4: Correct frame window for “Marvin” keyword

Figure 8.5 shows a frame with a red outline which was analyzed at a particular instant. The neural network is presented with a frame which was not part of the training set. Only a part of the “Marvin” keyword was present in this frame. In this case, since partial presence of “Marvin” was not part of the `z_openset` or Marvin classifier, the neural network ended up inferring it as the “go” classifier. This issue can be dealt with in three ways and readers are encouraged to study the advantages and disadvantages of all of them.

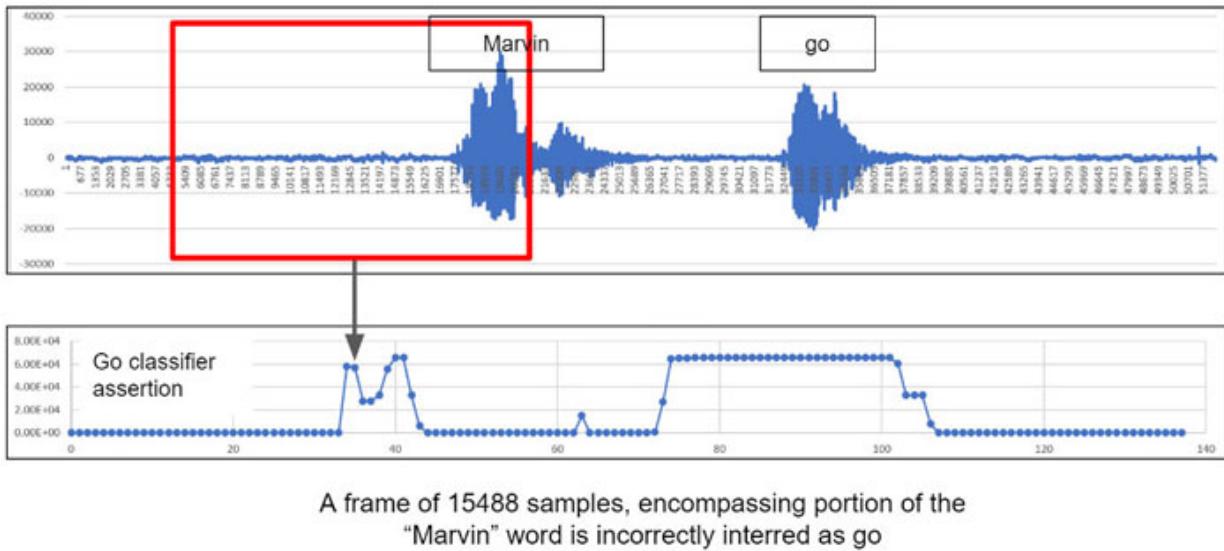


Figure 8.5: Root cause analysis of wrong keyword detection

Method 1: Running window averaging

Users will note that when the keyword “go” was spoken, 28 continuous inferences were detected, while there were only 5 inferences when it was not spoken. Since “go” was detected at the wrong place, it should be treated as **False Acceptance Rate (FAR)**. Noticing the contrast of 5 and 28 inferences, it is intuitive to somehow filter out an instant when only 5 inferences were asserted. The reader can try out different schemes to achieve a similar outcome; we will explore the most common linear filtering. Linear filtering is equivalent to finding a “mean” or “average” value. The mean function is a statistical function which is well behaved and studied. We can apply several statistical properties of the filtered signal to further analyze and optimize in a closed form manner. In running window averaging, a window is defined of n consecutive samples. Then one average value is saved, which is the average

of 1 to n input samples. Then the second value is saved, which is the average of 2 to n+1 input samples. The window is continuously moved by one, and an average value is generated. This is analogous to the low pass filtering concept.

We will apply running window averaging with different window sizes and compare the results. [Figure 8.6](#) compares 5 curves representing no filtering and varying length of running window averages. The running window average of the window size of 10, is shown in the orange curve. It can be seen that the first chunk of 8 wrong inferences is suppressed with a max value of about 42000. The filtered value still saturates to about 65000 when “go” was spoken. This suggests that if we choose a threshold of average value of 42,000 and 65,000, which is 53,500, then we can have the correct outcome.

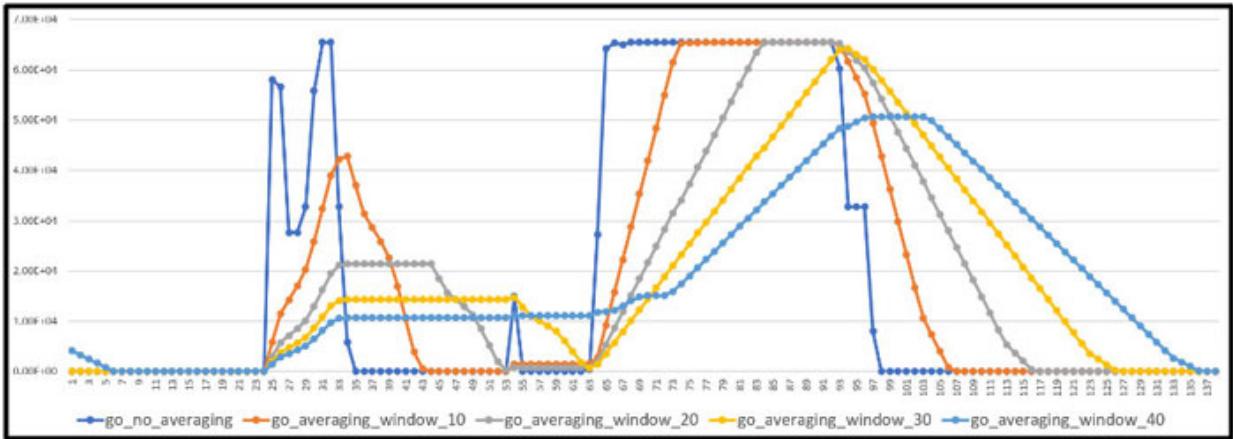


Figure 8.6: Illustrating how false alarms can be suppressed by running window averaging

The reader may be curious to find the optimum size of the running window averaging. To further develop intuition, the running averaging window is increased to 40, in increments of 10. The results are as expected. The contrast between the averaged value first increases and then decreases. [Table 8.1](#) shows the contrast values:

	Peak value at the unexpected time	Peak value at the unexpected time	Contrast
go_no_averaging	65461	65468	7
go_averaging_window_10	42808	65468	22660
go_averaging_window_20	21422	65468	44046
go_averaging_window_30	14780	64140	49360
go_averaging_window_40	11094	50638	39544

Table 8.1: The contrast between peak value at the expected and unexpected times for different window sizes of running window averaging

As we can see, the optimum value of the window size is about 30. Though 30 seems like a reasonable value, we will later show how to optimize the window size while taking **False Rate of Rejection (FRR)** and FAR into consideration.

Method 2: Enriching target classifier

We can create more training sets for the target classifier by intentionally selecting the frame where the keyword is not well centered. For example, as shown in the preceding [Figure 8.5](#), we can have a few training frames where the keyword “Marvin” is towards the right size and is only partially inside the window. Once we have significant training frames with this alignment, we expect the inferences to favor “Marvin” over “Go” for these types of frames. Correct inferences will be generated for longer times, thus providing opportunity for more running window averaging.

We will not study this method in this book, although the readers are encouraged to experiment with this method.

Method 3: Enriching open set classifier

In contrast to the second method, we can also enrich the open set classifier. For example, the misaligned frame when “Marvin” keyword is uttered, is towards the right and is only partially in the training frame and thrown in the open set. Effectively, we are telling the neural network that when we only see partial utterances of “Marvin”, to not trigger the “Marvin” classifier. This will be helpful in cases where a word which starts with the sound of “Mar”, for example, “marvelous”, “market” and so on, in random chatter, the classifier, “Marvin”, will not be triggered.

Once again, we will not study this method in this book, but readers are encouraged to see the advantages and disadvantages of these three methods.

Readers may be curious to know if they can analyze this behavior in Edge Impulse studio. In Edge Impulse studio, the wav file could be uploaded as a test file. Since it has both “Marvin” and “go” keywords, it should not be uploaded under the training category.

Figure 8.7 shows how the same audio segment with “Marvin” and “go” keywords uploaded. The label “`marvin_go_no_noise`” is also chosen to reflect what is in the wav file.

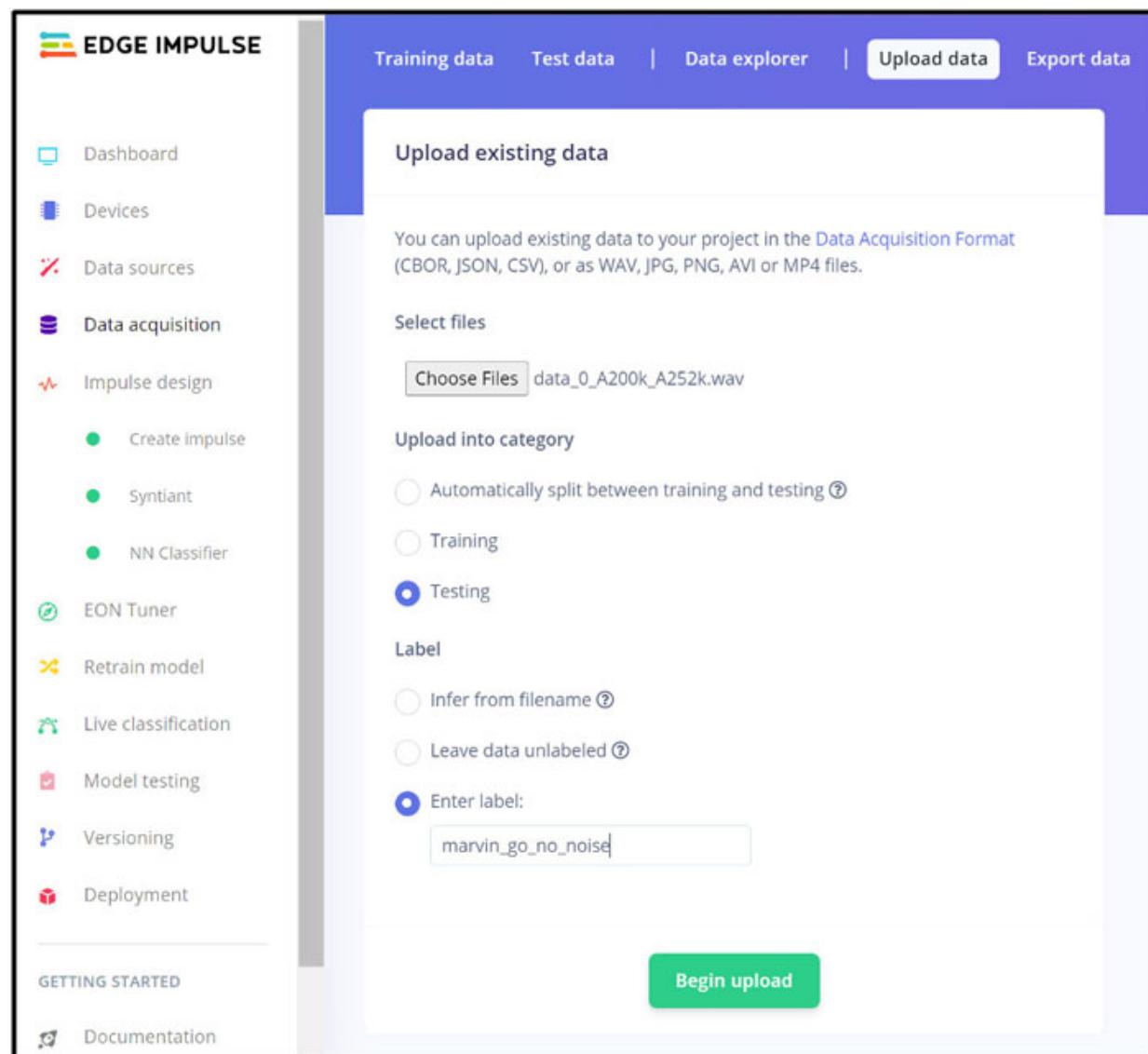


Figure 8.7: Uploading a wav file containing “Marvin” and “go” keywords

[Figure 8.8](#) shows how the data file can be located. To locate the file, follow these steps:

1. Select the **Test data** tab on the menu bar at the top.
2. Then click on the sorting funnel and unselect “**go**”, “**Marvin**”, “**stop**” and the **z_openset** labels. Only the relevant files will show.
3. The file can be selected and played in the studio to confirm if the file is correct. Label and file names should be chosen such that one could trace back the origin of the files.

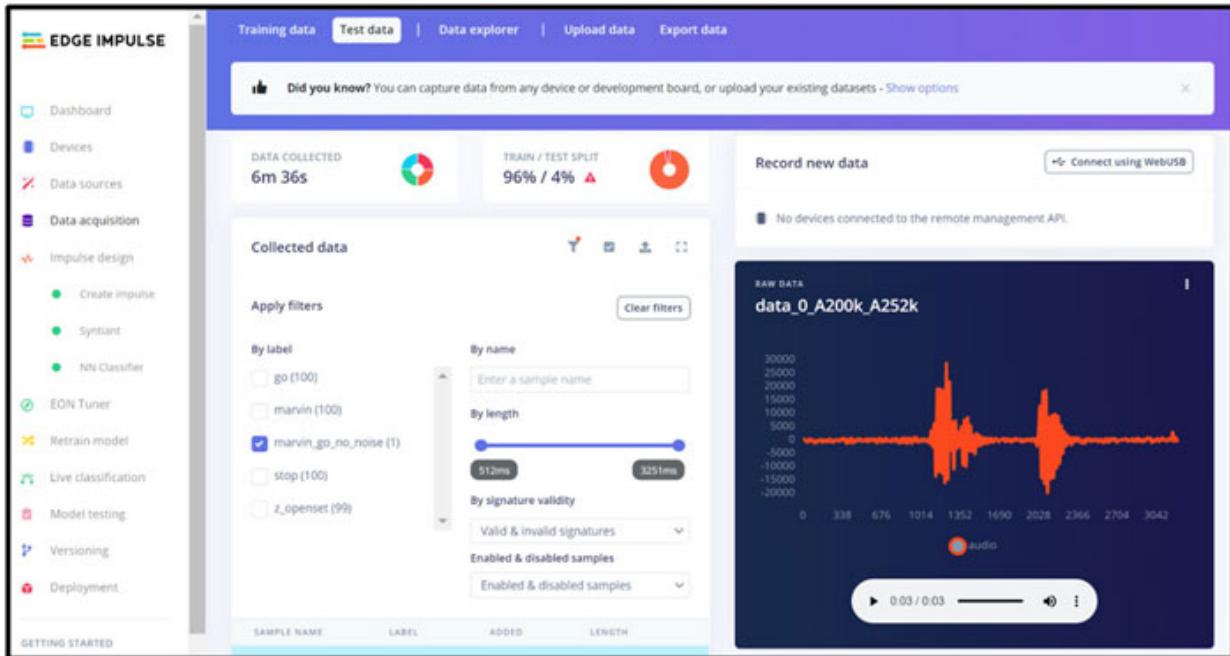


Figure 8.8: Locating and verified uploaded test wav file

Once the file is verified, software inference can be done by following the given steps:

1. Click on the **Live classification** tab on the left sidebar, as shown in [Figure 8.9](#).
2. On the main display panel, we will get the option to choose a file.
3. On clicking the **Load sample** button, the processing will start. When processing is done, the results can be reviewed in an interactive mode. Please refer to the following figure:

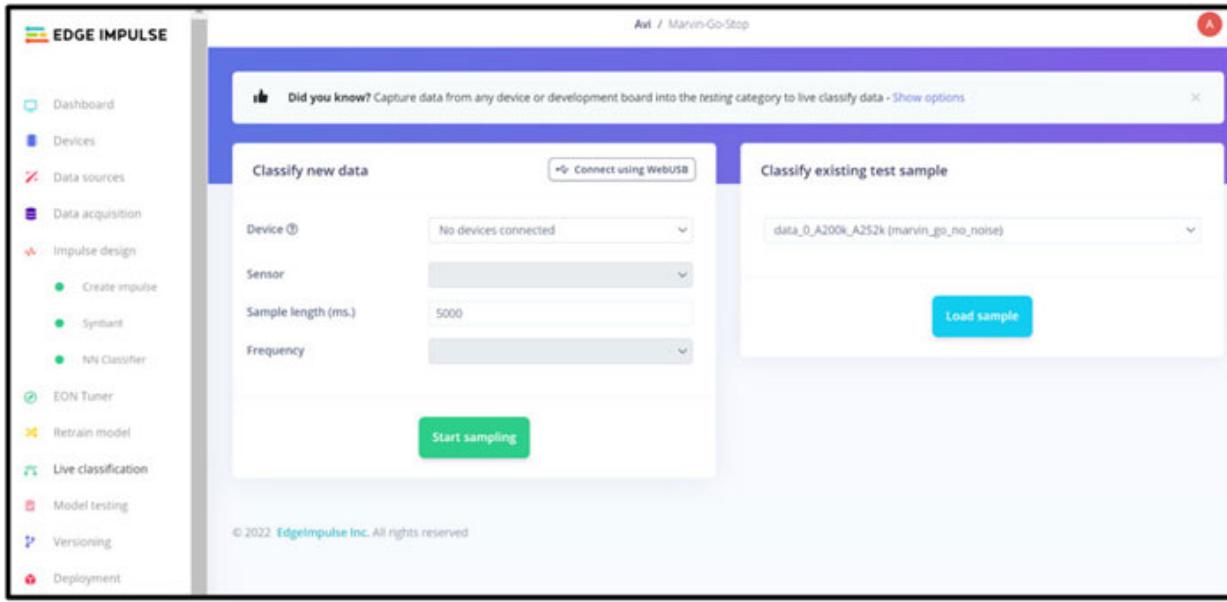


Figure 8.9: Loading file in the Live classification Menu

The Raw Data section on the screen displays the waveform, as shown in [Figure 8.10](#). There is a semi-transparent frame on top of the waveform, which can be moved horizontally. The length of the frame represents the frame size. The play button only plays the part of the audio section in the frame. Please refer to the following figure:

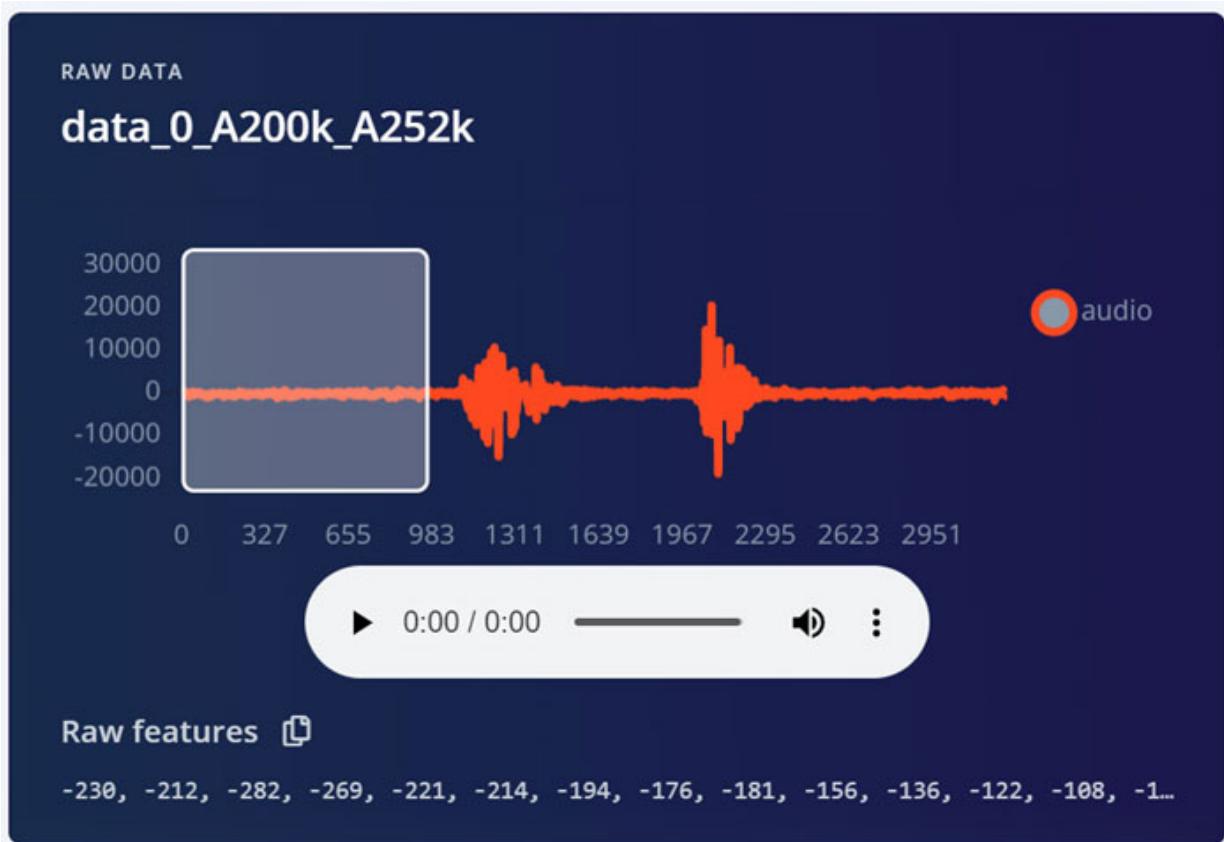


Figure 8.10: Waveform can be played and frame can be moved horizontally in this interactive window

The **Summary** section on the left side, as can be seen in [Figure 8.11](#), shows the results for the entire 3.5 second waveforms. It shows the summary of the classifiers. The results show a good number of classifiers for “**go**” and “**Marvin**” and “**z_openset**”. Seeing few assertions for “Stop” classifiers is similar to what was captured from the TinyML board. Refer to [Figure 8.11](#) for the summary of asserted classifiers:

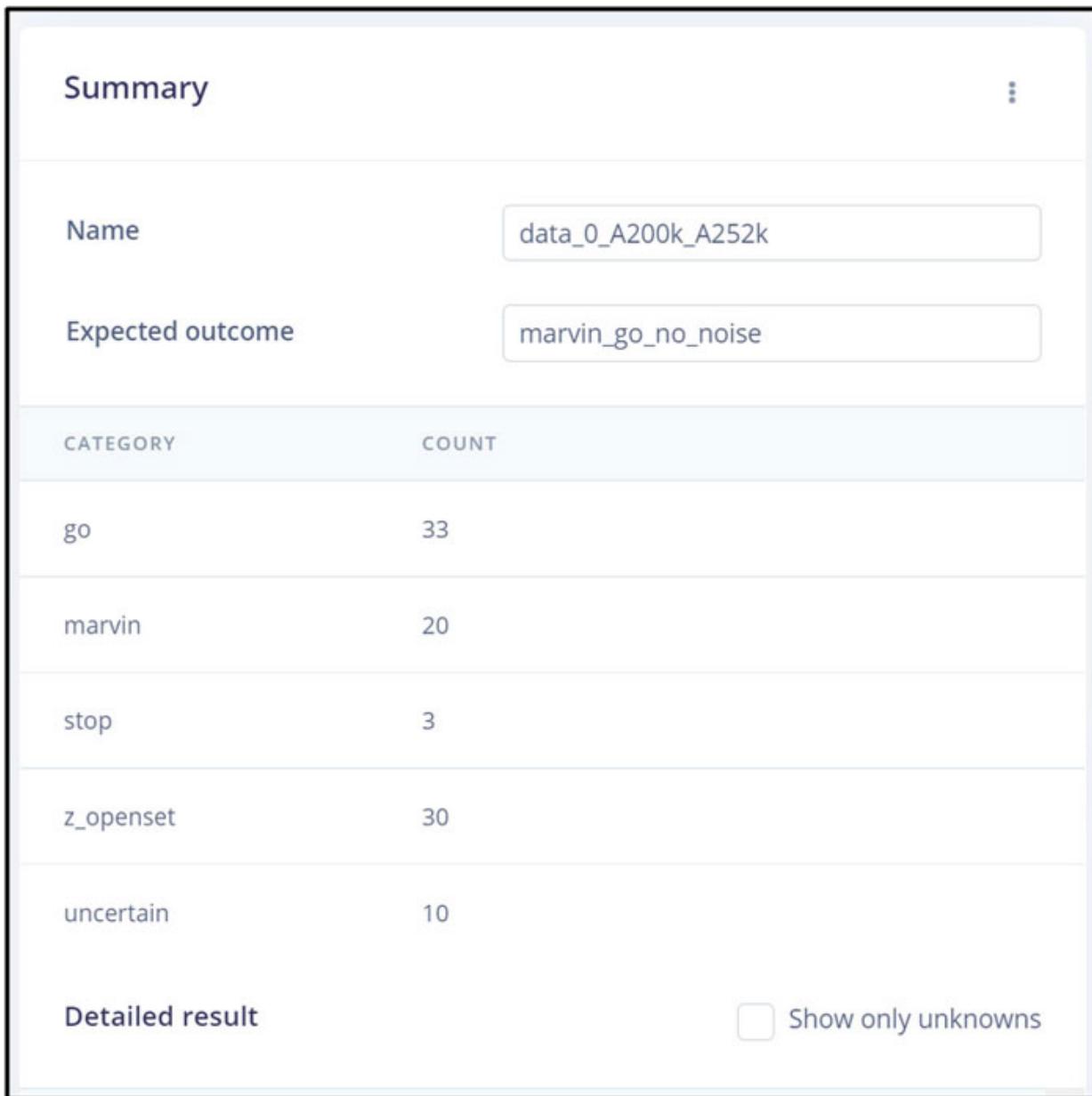


Figure 8.11: Summary result of Live classification using uploaded wav file

The section of the website on the left bottom side, shows tensor by tensor performance. Since there are approximately 200 inferences, we need to analyze the inferences in a few segments. The first segment shown in [Figure 8.12](#) starts with silence.

In this segment, the `z_classifier` is mostly triggered and thus, it is what is expected. If any classifier does not cross threshold of 0.8, then it is shown as uncertain. The rows are colored with an orange background for the inference where the softmax output does not cross 0.8.

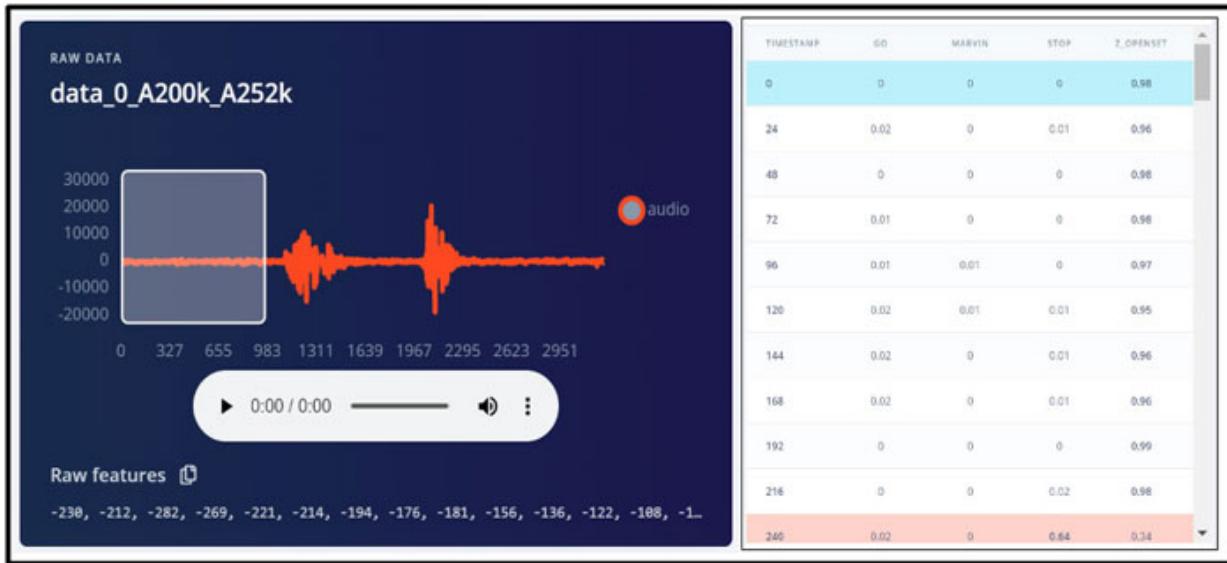


Figure 8.12: Segment with no signal

The “go” classifier starts getting triggered as the window overlaps on the utterance of “Marvin”, as shown in [Figure 8.13](#). The inference row, which is corresponding to the window, is shown with the light blue background. As expected, very few inferences are classified as “go”. The softmax values may not match exactly between the Live classification analysis within Edge Impulse studio and actual inference captured from the TinyML board. This is because the weights are truncated differently. The truncation is expected to make only cosmetic differences thus could be ignored for now.:)

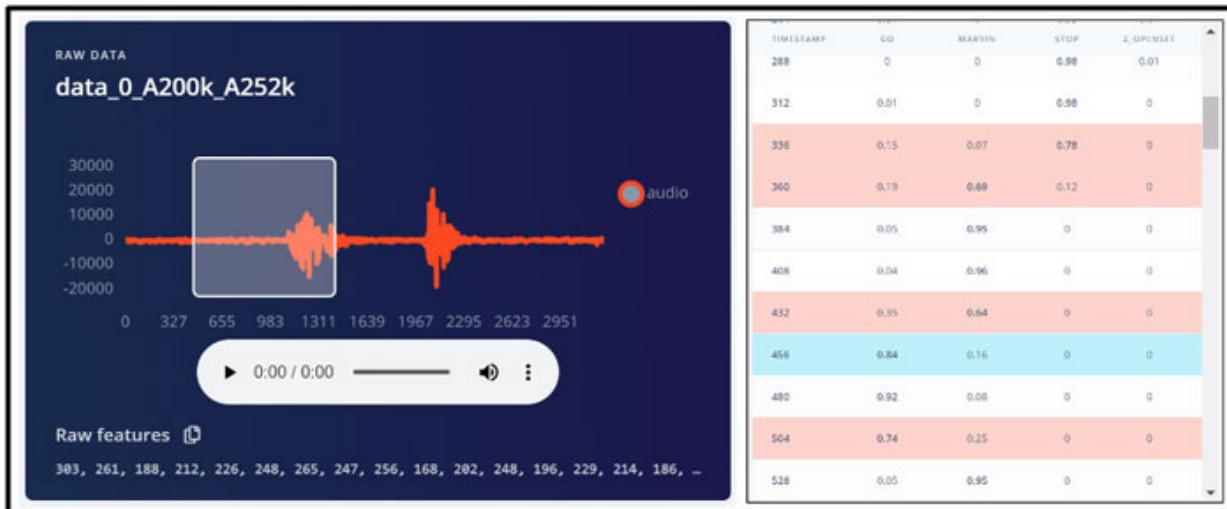


Figure 8.13: Segment with partial “Marvin” keyword classified as “go”

[Figure 8.14](#) shows the segment where the window is well centered around the utterance of the keyword “Marvin”. As expected, the correct classifier “Marvin”, is detected over several inference instances.



Figure 8.14: Segment where “Marvin” keyword is well centered

In [Figure 8.15](#), the window is placed over the utterance of the keyword “go”, and the correct “go” classifier is detected. The light blue background of the inference, shown in the right panel, corresponds to the selected window. This is a very interesting and interactive tool where users can hear what is spoken in the window and see its corresponding inference. If there is any unexpected result then the user can focus on it. Please refer to the following figure:

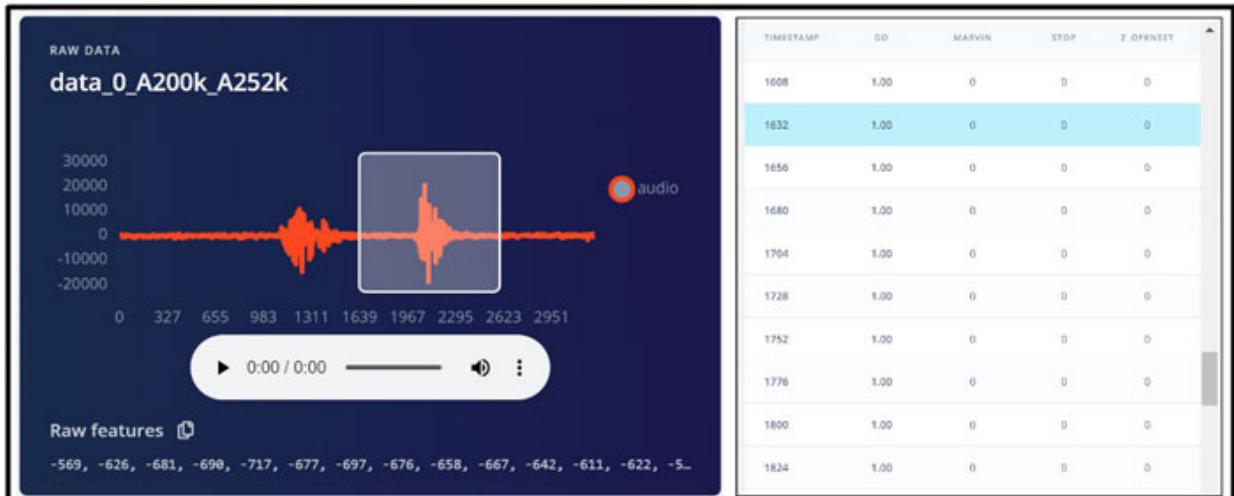


Figure 8.15: Segment where “go” is well centered

So far, we have studied the behavior of the neural network when the target classifier was present, and how the neural network reacted to it. If the neural network fails to recognize it, then it will be under the FRR. Next, we will study the FAR.

False Acceptance Rate testing

So far, we have seen the analysis where the intended keywords (target classifiers) were spoken and we analyzed the results. Now let us consider the case when the keywords are not spoken but these are rather, erroneously detected. This will be the **False Acceptance Rate (FAR)** characterization.

To test this case, 1 hour of NPR news was played from YouTube (9th Sept 2022 episode **PBS NewsHour full episode, Sept. 9, 2022 - YouTube**) and SoftMax classifiers were analyzed. For ease of data plotting and analysis in Microsoft Excel, one hour audio was recorded in 3.5 minutes long segments. 18 Files of 3.5 minutes were analyzed for about 1 hour of audio clip. It is assumed that “**Marvin**”, “**go**” and “**stop**” keywords were not uttered in this one hour of video clip. There is some chance that these words may get detected because they were either intentionally spoken during this episode or were part of another word. It is highly unlikely that the complete phrases “Marvin go” and “Marvin stop” were spoken during this one hour of recording. Carefully listening to each word during the long test audio could be a time-consuming task and was avoided. Users may consider how accurate they want to be in their testing but to some extent it does not matter in the big picture. This topic is brought up to alert users not to get sucked into literal accuracy which may be meaningless in the end deployment.

Out of the 18 segments, a segment was chosen, which showed significantly higher false alarms. *Figure 8.16* shows the first 10 seconds of the time domain waveform of the 3.5m audio segment:

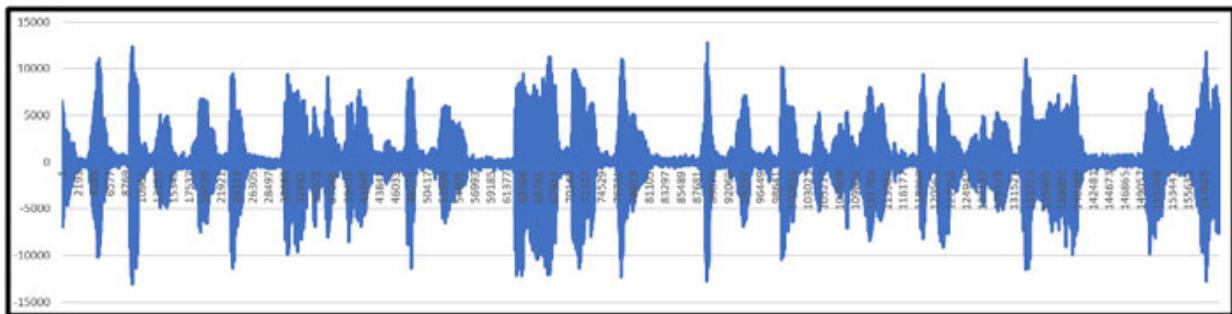


Figure 8.16: 10 seconds of segment which triggered significant false alarms

The preceding segment is uploaded into the Edge Impulse Studio. Since this is a longer sequence, it takes a few seconds. *Figure 8.17* shows the summary results:

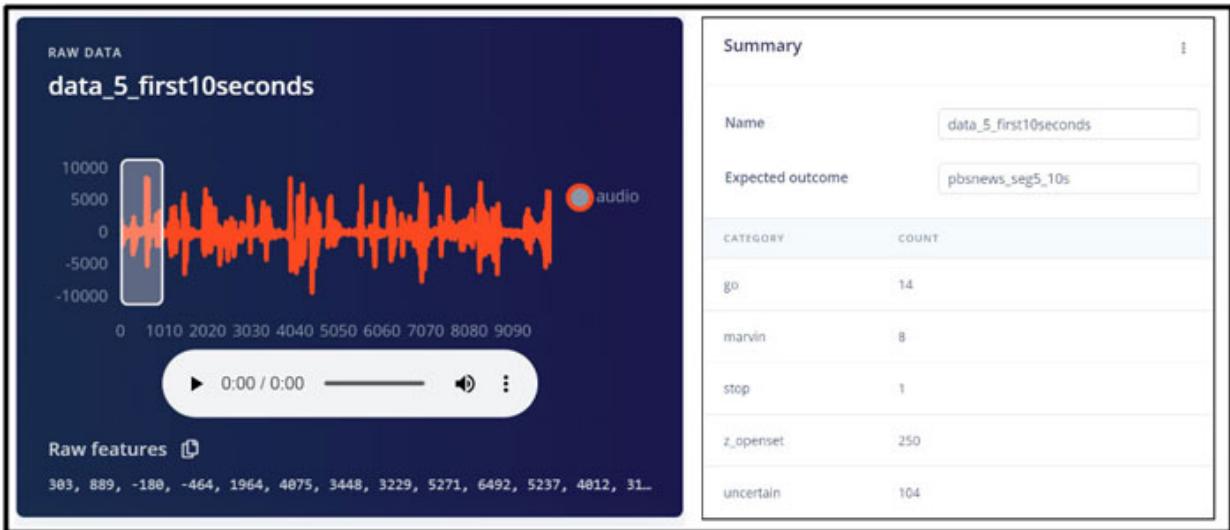


Figure 8.17: Live classification analysis of 10 second segment

As shown in *Figure 8.18*, by scrolling down, we will find the first false acceptance. By clicking on the inference row, the frame can be centered, which causes the false alarm. Users can play the segment which caused the false inference. Sometimes, the frame may be too small and we cannot understand the word without the context, and so, users may need to play longer audio segments. If indeed the keyword was spoken, then it will not be a false alarm.

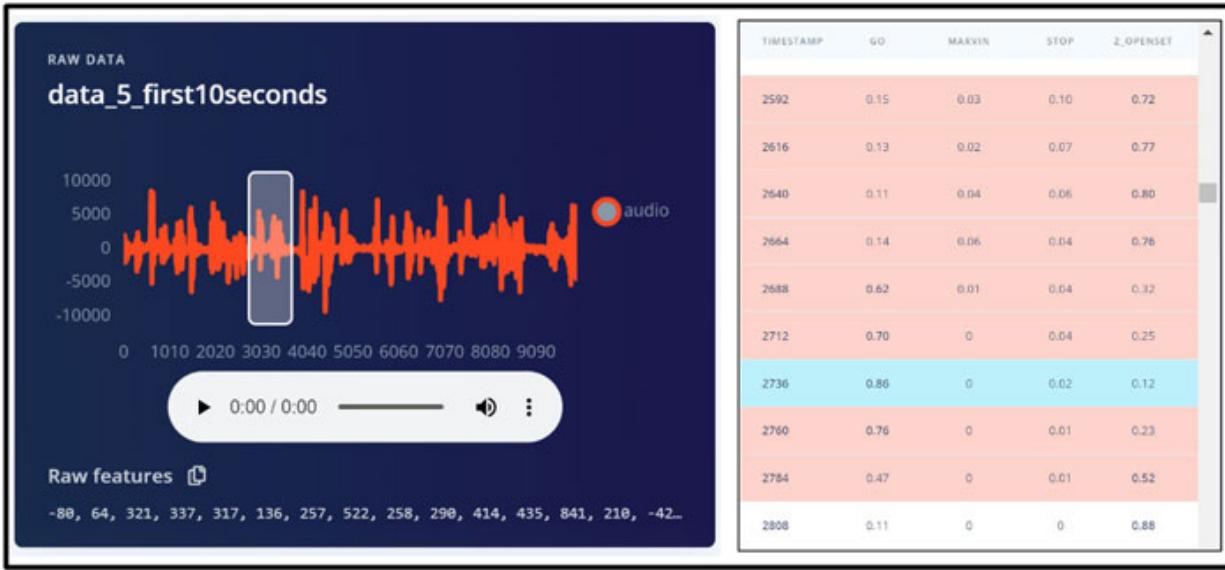


Figure 8.18: Scrolling down to find first false alarm

For further analysis, the data will be analyzed using Python scripts instead of Microsoft Excel. [Figure 8.19](#) analyzes the raw inferences captured for the “go” classifier in the 3.5-minute segment. It is underwhelming to see so many false alarms are triggered in only a 3.5-minute segment. However, we will see how we can deal with this situation. Then the running window averaging is applied with a window size of 30. The suppression of false acceptance is quite dramatic. In the example shown in the preceding [Figure 8.18](#), we arbitrarily chose a window size of 30. In next section, we will explore how to optimize window sizes for each classifier.

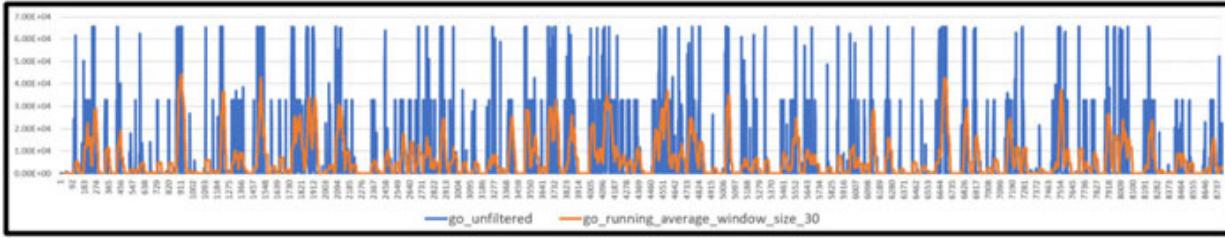


Figure 8.19: Application of running window averaging reducing the false acceptance instances

Optimization of window size in running window averaging

It is clear that we need some filtering to improve FAR. As we filter more, at some point FRR will start degrading. So, the question that arises is, how do

we find the optimum filtering which will provide a good balance between FAR and FRR. The window size of the running window averaging is swept from 1 to 80. Both FAR and FRR are plotted on the same x axis, representing the window size of running window averaging.

[Figure 8.20](#) shows FAR and FRR curves as a function of running average window size for the keyword “`go`”. Notice that errors are plotted on a log scale. If window size is chosen to be 30 (shown by green arrow), then there is hardly any FRR but FAR is about 33 per hour. So, it will be 792 per 24 hours. If the window size is chosen to be 39, then FRR will be 4% while the FAR will be 18 per hour, or 432 per 24 hours. An aggressive window size of 51 will reduce FAR to 9 per hour or 216 per 24 hours at the expense of 8% FRR. Both FRR and FAR can be improved by increasing the training data size. For further analysis, window size of 39 will be chosen for the keyword “`go`”.

Refer to the following figure:

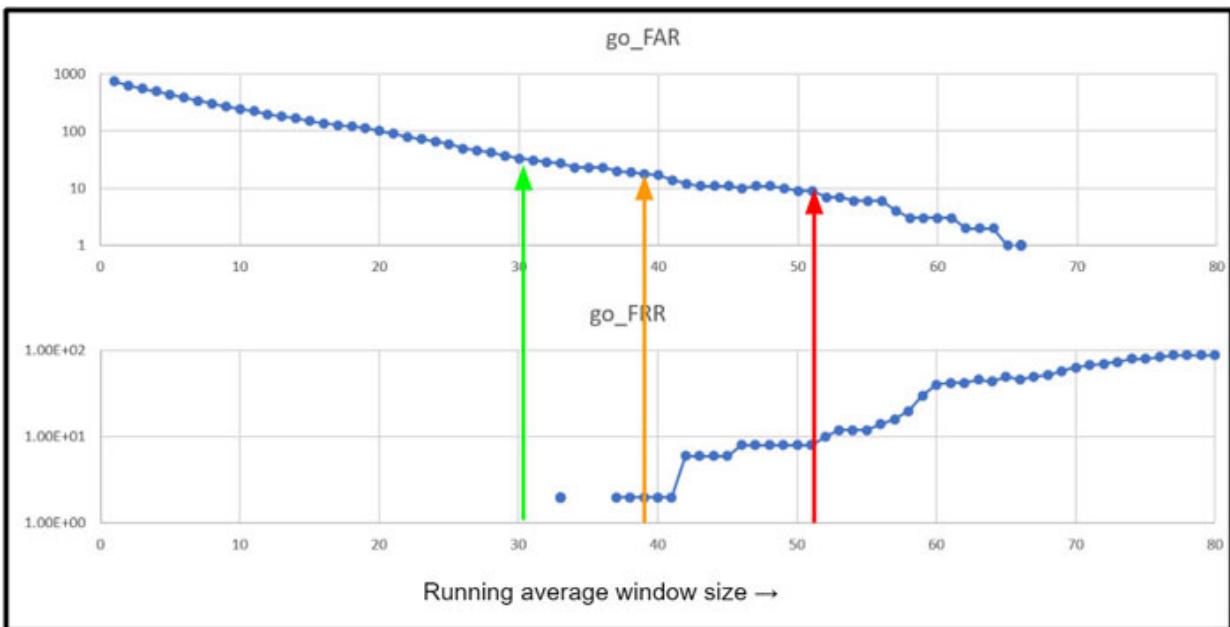


Figure 8.20: “`go`” keyword FAR in one hour of PBS news and FRR of 50 samples plotted as a function of window size of running window averaging filter

Plots for the keywords “`Marvin`” and “`stop`” are shown in [Figures 8.21](#) and [8.22](#) respectively. The optimum size of the window will be dependent on the complexity of the keyword, its uniqueness and the way alignment are done while preparing the training set. The optimum window sizes are significantly different for the keyword “`Marvin`” with respect to the “`go`” keyword. For

the keyword “**Marvin**”, the window of 10 seems optimum which will have about 4% inaccuracy in FAR and about 100 FARs in 1 hour or 2400 FARs in 24 hours. Similarly, the optimum window size of the keyword “**stop**” is chosen to be 29 with similar FRR and FARs. Refer to the following figure:

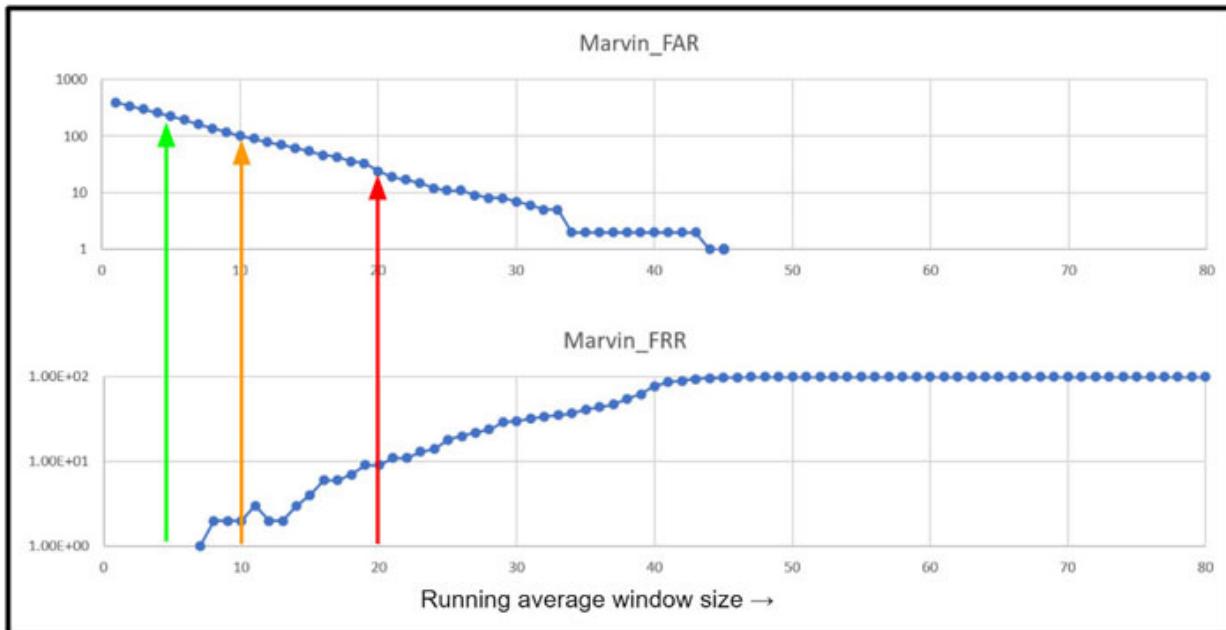


Figure 8.21: “go” keyword FAR in one hour of PBS news and FRR of 50 samples plotted as a function of window size of running window averaging filter

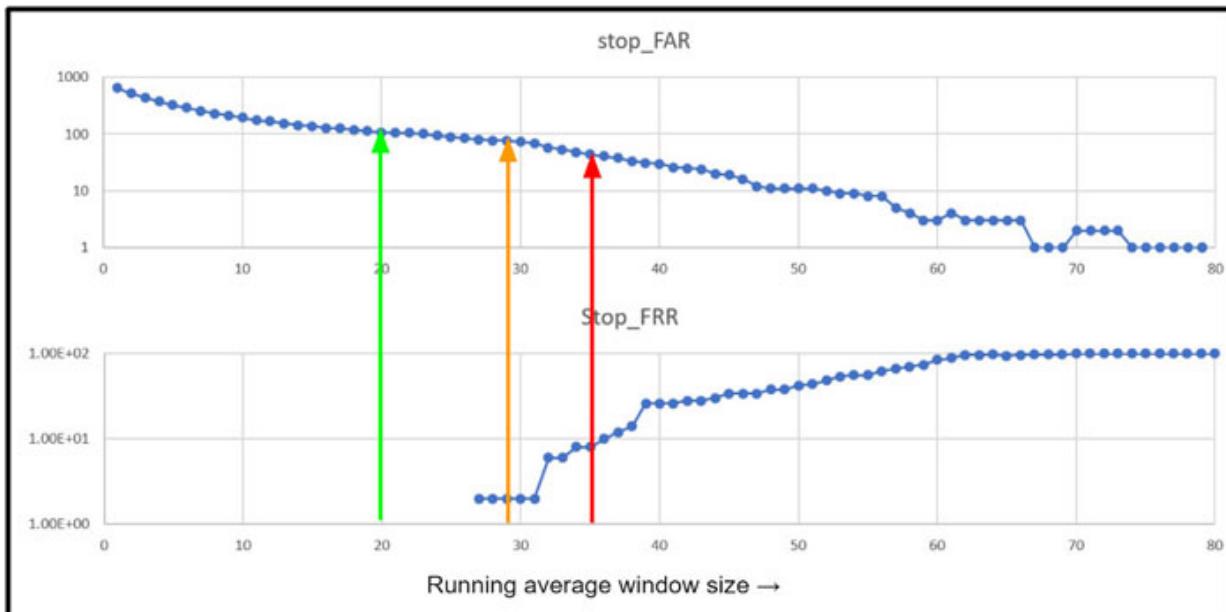


Figure 8.22: “go” keyword FAR in one hour of PBS news and FRR of 50 samples plotted as a function of window size of running window averaging filter

It should be noted that different applications may have different system level implications, and so the window size should be chosen considering system level considerations.

For a phrase, the probability of FRR will be dependent on an individual keyword's FRR. Since we are targeting low FRR (less than 10%), the net FRR of the phrase can be approximated to the sum of the probabilities. For example, if each keyword has 4% FRR, the combined phrase of two words is expected to have FRR of 8%. The FARs are expected to reduce significantly because the chances of two FAR happening at the same time will also be lower. We will study net results in the next section.

Figures 8.23, 8.24 and 8.25 show the raw and filtered inferences for the keywords “**go**”, “**Marvin**” and “**stop**” for optimum window sizes of 39, 10 and 29 respectively. Though the suppression of false alarms is quite significant, in this 3.5-minute clip, all three classifiers got triggered at least once.

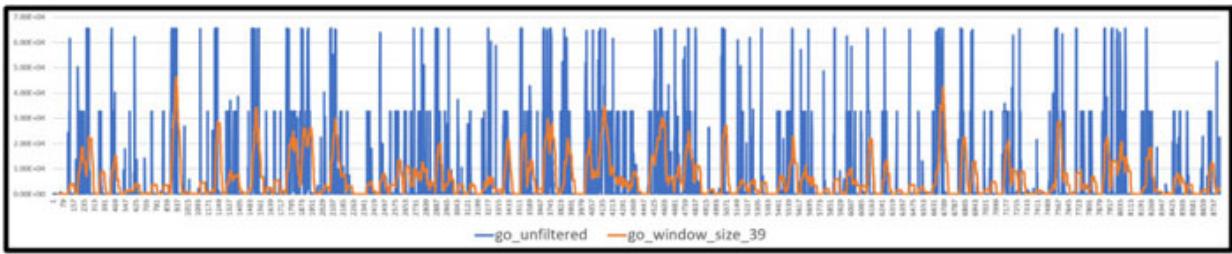


Figure 8.23: Running window averaging of window size of 39 for the keyword “**go**”

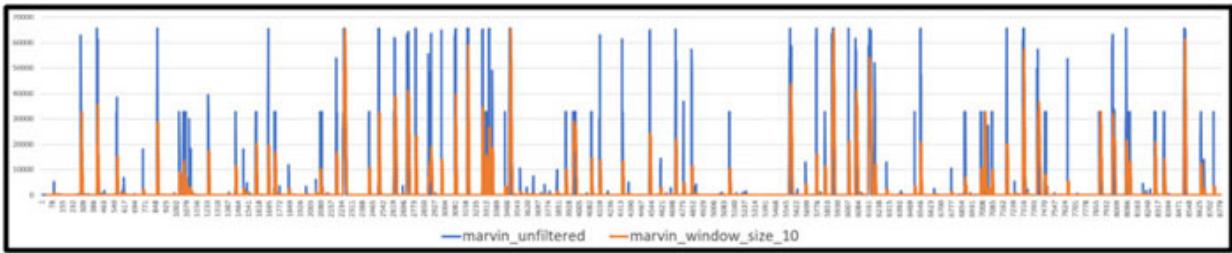


Figure 8.24: Running window averaging of window size of 10 for the keyword “**Marvin**”

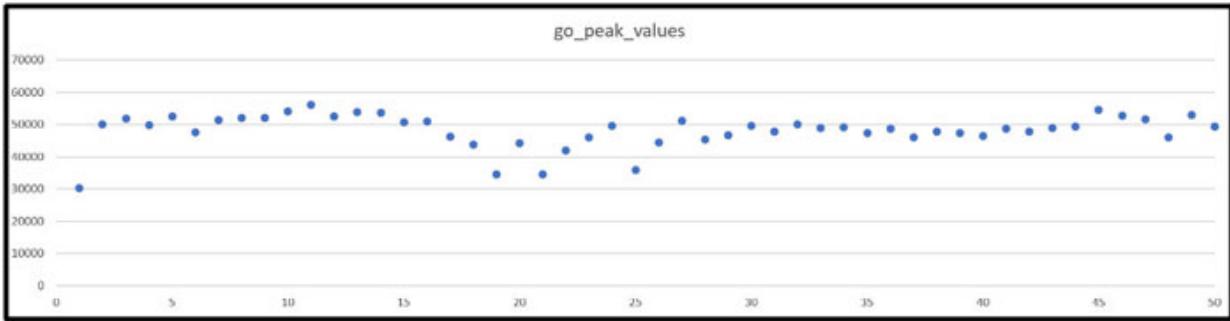


Figure 8.25: Running window averaging of window size of 29 for the keyword “stop”

Excel sheets are great for developing insight and intuition in analyzing the data. However, it can be used for analyzing data only for a limited amount of data collection. On the other hand, Python routines can analyze the data for hours and hours of data.

Some statistics are generated using Python routines. Optimum window size is used to find the peak values for the target classifiers in the 50 instances of “Marvin go” and “Marvin stop”.

First, we will analyze 50 files of “Marvin go” utterance. [Figure 8.26](#) shows the distribution of the peak value of the “go” classifier. The values are plotted for 50 utterances.

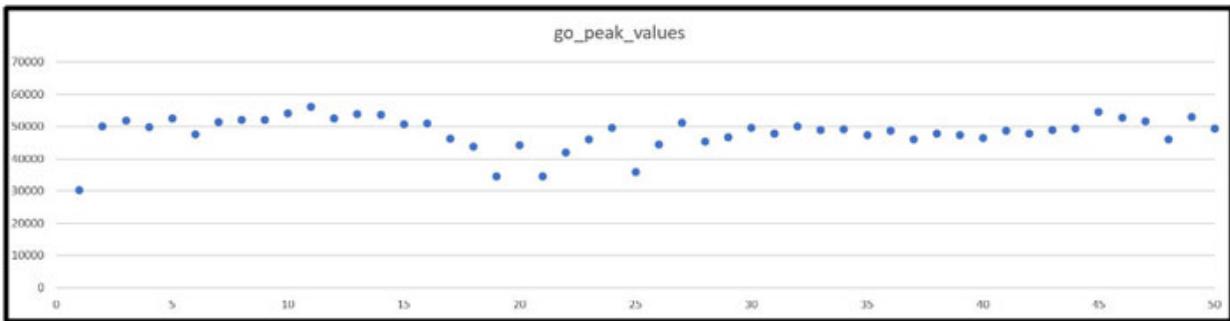


Figure 8.26: Peak values of “go” classifier after running through running window averaging in the phrase “Marvin go”

[Figure 8.27](#) shows the peak values of the “Marvin” classifier in the “Marvin go” phrase. Due to small averaging, most of the time, the value is saturated:

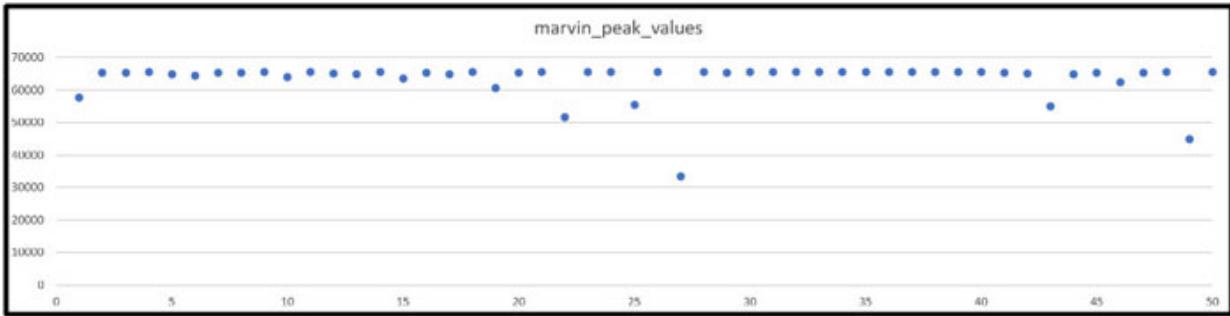


Figure 8.27: Peak values of “Marvin” classifier after running through running window averaging in the phrase “Marvin go”

Similarly, the [Figures 8.28](#) and [8.29](#) show a plot of the peak values for keywords “Marvin” and “stop” in the phrase “Marvin stop”:

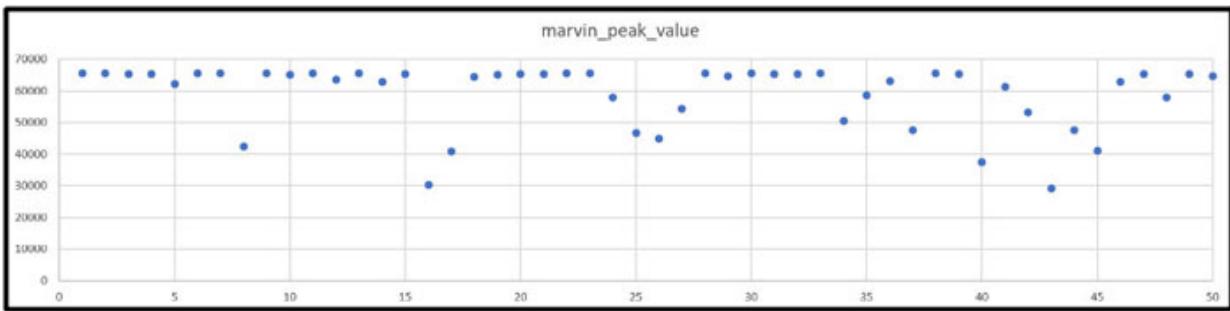


Figure 8.28: Peak values of “Marvin” classifier after running through running window averaging in the phrase “Marvin stop”

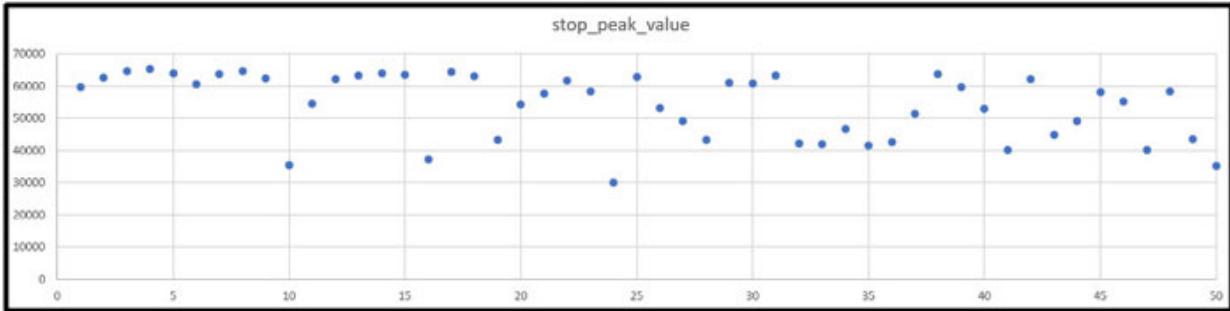


Figure 8.29: Peak values of “stop” classifier after running through running window averaging in the phrase “Marvin stop”

These charts provide good insight about the distribution of the peak values, which relates how easy or difficult it is for it to be detected by the current model. The utterances corresponding to the lower values can be played again. There could be several reasons why these utterances have lower peak value. Some of the reasons could be due to too high or too low volume, noisy background, different accent. Often, when data is captured for training,

volunteers use a neutral tone and pace. But in real life, the tone and pace could change significantly.

Let us assume that the “Marvin go” and “Marvin stop” is applied to a robot whose name is Marvin. And let us also assume that kids are playing games with robots. In this case, we do not expect kids to be using a neutral voice to instruct their robots. In the excitement of the game, their tone and pace will change significantly.

This is one of the reasons why models have poor field performance. Ideally, the data for training should be captured in the same setting as it is going to be used. But at the inception of the project, the device may not exist and so, it creates a chicken and egg issue. So, it is inevitable to start with data collected elsewhere and then deploy the system and then capture the data in the field and continuously retrain. At some point, the original data should be discarded to reduce its bias.

Now let us analyze a 60-minute news data. [Table 8.2](#) tabulates results for the 1 hour. The false alarms are in the range of 4.4% or the accuracy is about 95.6%. This is like the 96.5% accuracy observed during the training. If we extrapolate one hour of false alarms to 24 hours of false alarms, we get over 150 thousand errors. The expectation is in single digit. Now the reader can see the disconnect from raw neural network performance and what is the expectation. The discrepancy is 4-5 orders of magnitude! In the next section, it is shown how system level performance can be improved:

	Classifier Go	Classifier Marvin	Classifier Stop	Total False Alarm	% False Alarm
1hr of news play	2753	1162	2651	6566	4.38
Extrapolated to 24 hours	66072	27888	63624	157584	4.38

Table 8.2: Analysis of inferences during the playing back of 60 min PBS news

The Running window averaging is expected to reduce false alarms significantly. Using the same normalized threshold of 0.5 (or absolute threshold of 32768), the false inferences are reduced significantly as shown in [Table 8.3](#).

The errors reduce from over 150K to about 6K. This is about 1-2 orders of magnitude improvement. Though we just achieved dramatic improvement, we are still far away from what is expected. In the next section, we will apply the probability theory of multiple events to happen in certain order to reduce FARs.

	Classifier Go W.S. 39	Classifier Marvin W.S. 10	Classifier Stop W.S. 29	Total False Alarm
1hr of news play	33	100	107	240
Extrapolated to 24 hours	792	2400	2568	5760

Table 8.3: False alarm after the application of running window averaging

Phrase recognition constraints to improve system level performance

The error in [Table 8.3](#), shows detection of keywords “go”, “Marvin” and “stop” which may come at random times. At the system level, we can ignore “go” or “stop” if Marvin is not spoken. This means that the system level false alarms cannot be more than 2400, which will be limited by the keyword “Marvin”. Sometimes, the leading keyword is called wakeup word. “Marvin” is a wakeup word here. Now if we consider that the classifier “Marvin” should be spoken before “go” or “stop” within 1 second window, then the system level errors should drop even more.

To add a new constraint, we need to characterize the delays between keywords in normal phrases, after it has gone through the running window averaging. The delay between “Marvin” and “go” keywords are analyzed. Similarly, the delay between “Marvin” and “stop” keywords are also analyzed. A Python routine is used to find the delays between the peak point of the keywords, “Marvin”, “go” and “stop”. Threshold was adjusted from 0.5 to 0.3, to ensure that all the utterances are captured. [Figure 8.30](#) shows the delay between the peak point of the utterances “Marvin” and “go” keywords:

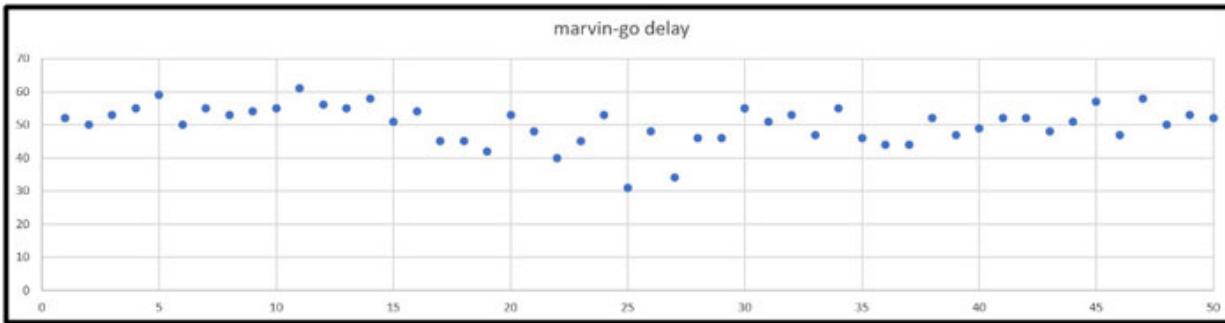


Figure 8.30: Delay between “Marvin” and “go” keywords

Similarly, [Figure 8.31](#) shows the delay between “**Marvin**” and “**stop**” keywords:

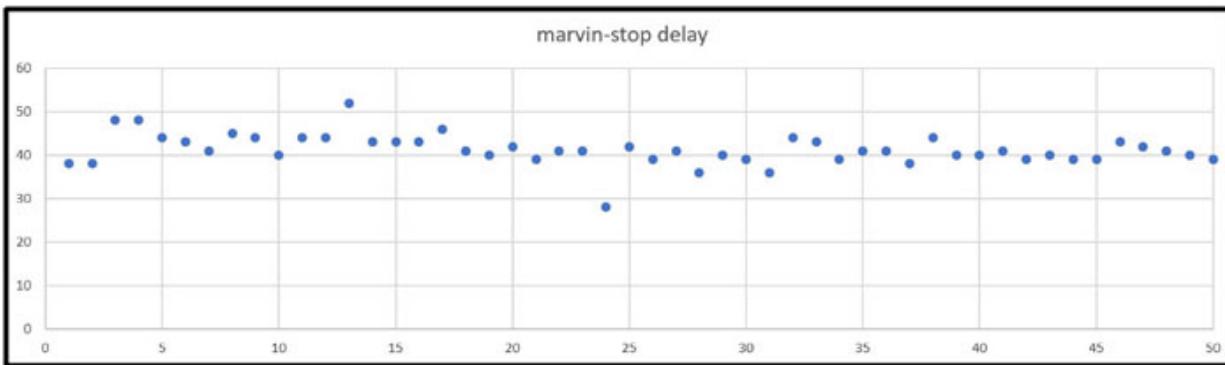


Figure 8.31: Delay between “Marvin” and “go” keywords

[Table 8.4](#) shows the statistical analysis to find the lower and upper bound in the delay using both min and max, as well mean and standard deviations. With limits of +/- 2 standard deviation, we target 95% of the population. The min and max of the population of 50 samples came close to +/-2 standard deviation points, which shows the normal distribution. It is always a good practice to check if the data is following a normal distribution or not. This can be checked by validating if the population in +/-1 std. dev. (68%), +/-2 std. dev. (95%) and +/-3 std. dev. (99.7%) following the normal distribution or not. Of course, for a +/3 sigma test, the sample size should be in the range of 1000. The final delay settings were based on worse case numbers coming from the min, max analysis and +/-2 standard deviations.

Refer to the following table for a statistical analysis to find optimum delay:

	Marvin-go delay Analysis	Marvin-stop delay Analysis
avg	50	41
std	9	7
std-2s	32	28
min	31	28
Recommended Min Delay	31	28
std+2s	68	55
max	61	52
Recommended Max Delay	68	55

Table 8.4: Statistical analysis to find optimum delay between the wakeup word “Marvin” and keyword “go” and “stop”

This constraint dramatically reduces FAR with little impact on the FRRs. In 1 hour of the PBS News, there was only one instant when a false error occurred matching the complete phrase “Marvin go”. If we extrapolate for 24 hours, we will expect about 24 FARs, which is not too far from where production grade expectation lies, which is single digit FAR in 24 hours. During development, it is preferred to extrapolate numbers for lengthy experiments to save time and cost between multiple iterations. However, for signing off on the specifications, the results should be presented with a certain statistical confidence interval, for example, 95%. The test duration should be increased and multiple tests should run in parallel to provide results with statistical confidence. [Table 8.5](#) shows how the FAR reduces as we apply post softmax filtering and add constraints:

	Raw softmax inferences	Running window averaging	Phrase delay constraint
1hr of news play	6566	240	1
Extrapolated to 24 hours	157584	5760	24

Table 8.5: FAR improvement with suggested filtering and adding constraints

We need to check FRR under the same filtering and constraints. The FRR tested under no noise conditions comes out to be 4% (96% accuracy) for “Marvin go” phrase and 6% (94% accuracy) for “Marvin stop”. [Table 8.6](#) shows tabulated results. By now, we have achieved reasonable FAR and FRR performance under clean conditions. In the field, it is expected to have background noise present. We will explore the performance under noisy conditions in the next section:

	“Marvin go” phrase		“Marvin stop” phrase	
Phrase FRR	4%		6%	
FRR analysis				
Fall out due to delay constraint	2%		0%	
Individual word	Marvin	Go	Marvin	Stop
FRR	0%	2%	4%	2%

Table 8.6: FRR analysis with same filtering and constraints used for 1 hour PBS news testing

FRR testing under noisy conditions

Before we further optimize, we should test the target keywords under noisy conditions. We will see how a small amount of noise reduces the system performance significantly. We will use Edge Impulse studio to validate the

results. The original wav files are mixed with noise. This will keep the signal to noise ratio consistent, which is difficult to maintain in a real-life environment. The real-life noise was captured with a table fan running full speed. The TinyML board was kept at the back of the fan, to avoid microphone and air flow induced artifacts. The intent was to capture the noise of the fan only. [Figure 8.32](#) shows the signal with and without noise in the time domain. The noise seems relatively small with respect to the signal. Please refer to the following figure:

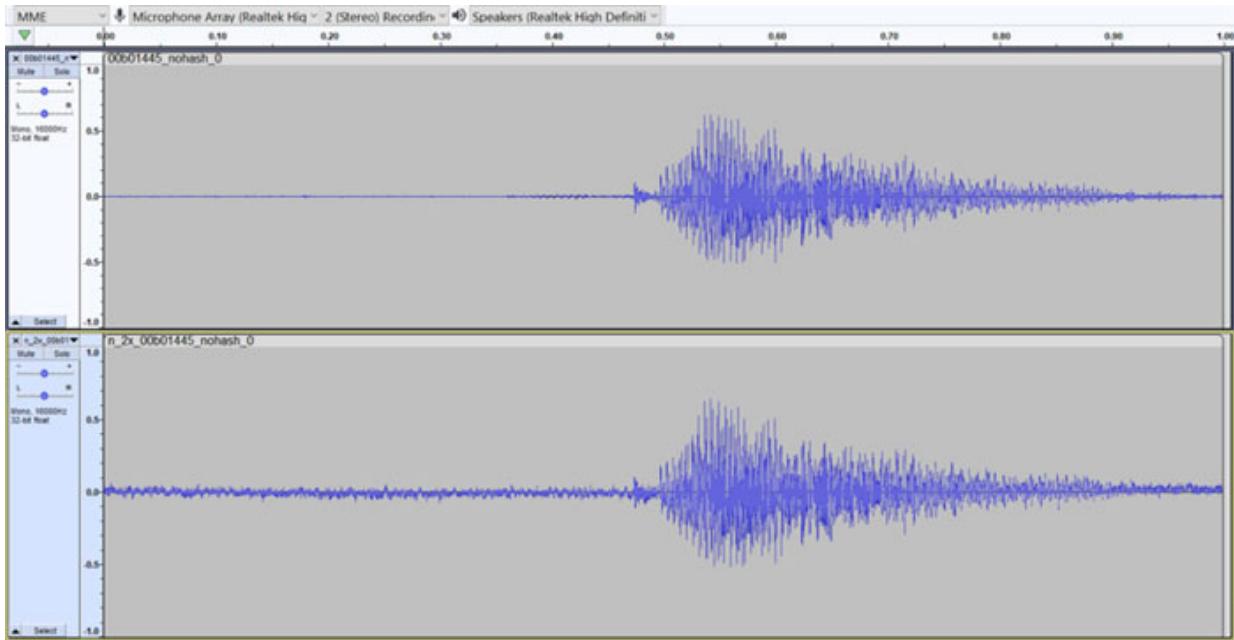


Figure 8.32: The top waveform is the original wav file without fan noise. The bottom waveform is the same wav file but mixed with the fan noise. The noise can be seen between 0 and 0.40 second period.

In the spectral view, as seen in [Figure 8.33](#), the noise appears a little bit more dramatically due to the log conversion in the Mel scale transformation. This is the spectrogram which is seen by the neural network and thus we expect small noise to cause big degradation.

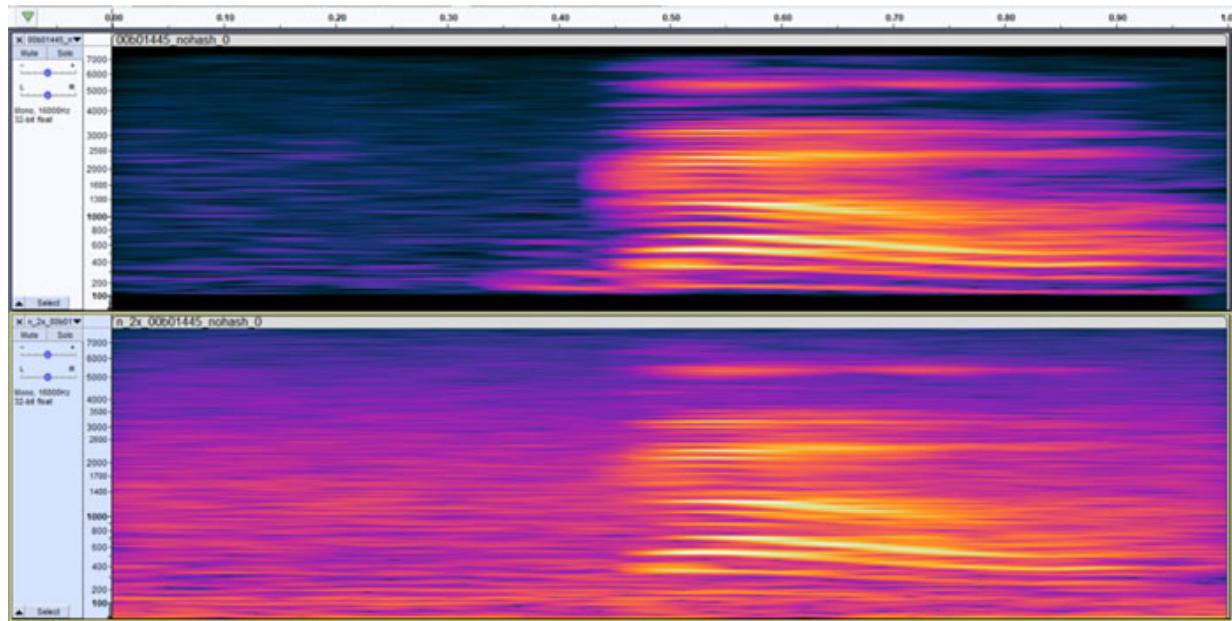


Figure 8.33: The top spectrogram is corresponding to the original wav file without fan noise. The bottom spectrogram is for the same wav file but mixed with the fan noise

We are referring to this level of noise as 2x. Mixing is done by adding up the noise by multiplying it by a factor (in this case 2) and then summing it with the original signal sample by sample using a Python script. While mixing the noise, the noise segments should be randomly taken for each waveform from the captured wav file.

This study is done to show how the model performs before it is re-trained with the noise. The testing is done in the **Model testing** step of the Edge Impulse studio. The same set of utterances were used for all the classifiers except the open set while adding the noise. The open set was not added with the noise because it is expected that noise may reduce the false accepts which is not the intention and user may draw wrong conclusions. It is surprising to see a dramatic drop in performance when the noise was added. [Figure 8.34](#) compares the results where the neural network was trained on the original data set but tested with a data set with noise and without noise:

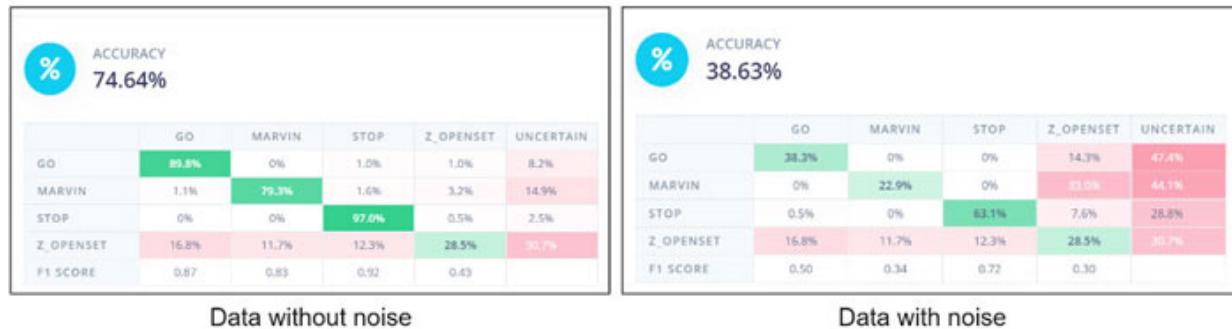


Figure 8.34: Model testing to show the impact when small noise is added

We can see the accuracies dropped from 75% to 39% by small introduction of noise. In the next topic, we will see how we can deal with noise impairment.

Improving FRR performance under noisy conditions

Neural networks need to be trained for expected input. The background noise is an input which was not seen by the neural network in our first training. Therefore, it is not surprising that the model generated in the first training performed poorly. The solution is quite simple: add more data with noise. We can still use the old data and mix noise with different amounts. This step comes under data augmentation step. It is always a good idea to capture data under real conditions. However, data augmentation is the next best thing and it can be done in trivial amount of cost and time. Nonetheless, a few things should be taken into consideration. When adding the noise segments, do not use only one noise segment and mix it with all the data. Rather collect the noise of a longer period and then mix with unique segments of the noise for each data set. If the same noise segment is used, the neural network will expect that particular noise pattern for the match.

Not only the segment of the noise but the amount of the noise can also be varied. The data can grow very quickly after just a few augmentation permutation combinations. Therefore, it may make sense to randomly select noise segments and data sets. Once the data size grows, the cost of the computation becomes a concern. To illustrate how to handle noisy background, in our example, we doubled the data set with adding fixed 2x noise. Reader is encouraged to apply different levels of noise. Since we have fewer “Marvin” instances, the “Marvin” data was augmented with 1x, 2x

and 3x noise. The z_openset was not augmented for the noise. [Figure 8.35](#) shows the new data set:

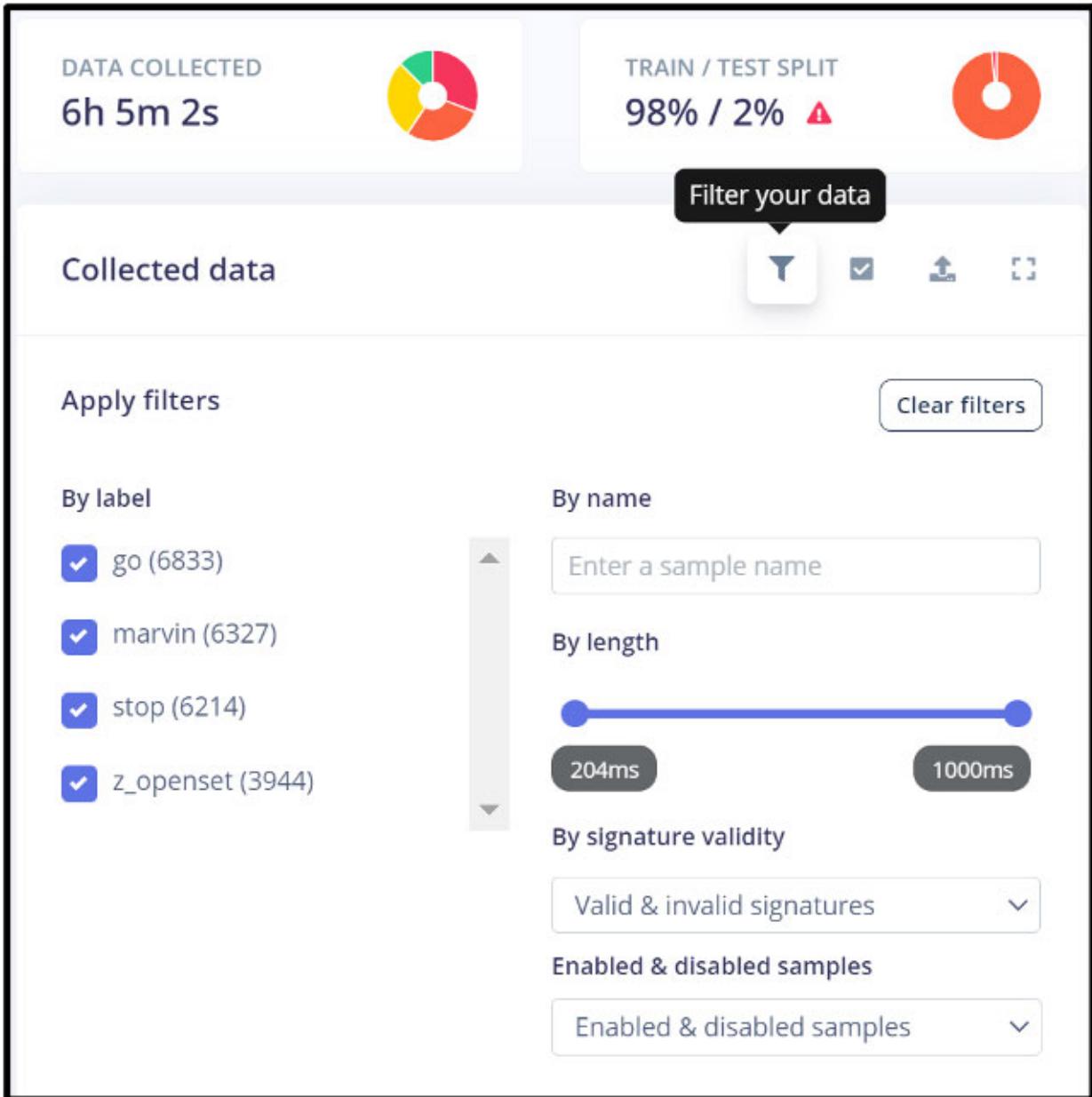


Figure 8.35: New set of data with time domain noise augmentation

By adding augmented data with noise, the net data collected is a little over 6 hours as shown in the top left corner of [Figure 8.35](#). At this time, we saw it was reaching the limit for some of the processing can be done in the developer license of Edge Impulse which is free of cost. As a matter of fact, the number of epochs were reduced from 100 to 70, to fit within the compute budget. [Figure 8.36](#) shows the result of the re-training. It was suspected that

fewer epochs and noisy data will reduce the accuracy, however, we saw the accuracy improved slightly. At least we can say the performance did not drop while adding noisy data. Small improvement and degradation may be within variation due to random initialization.

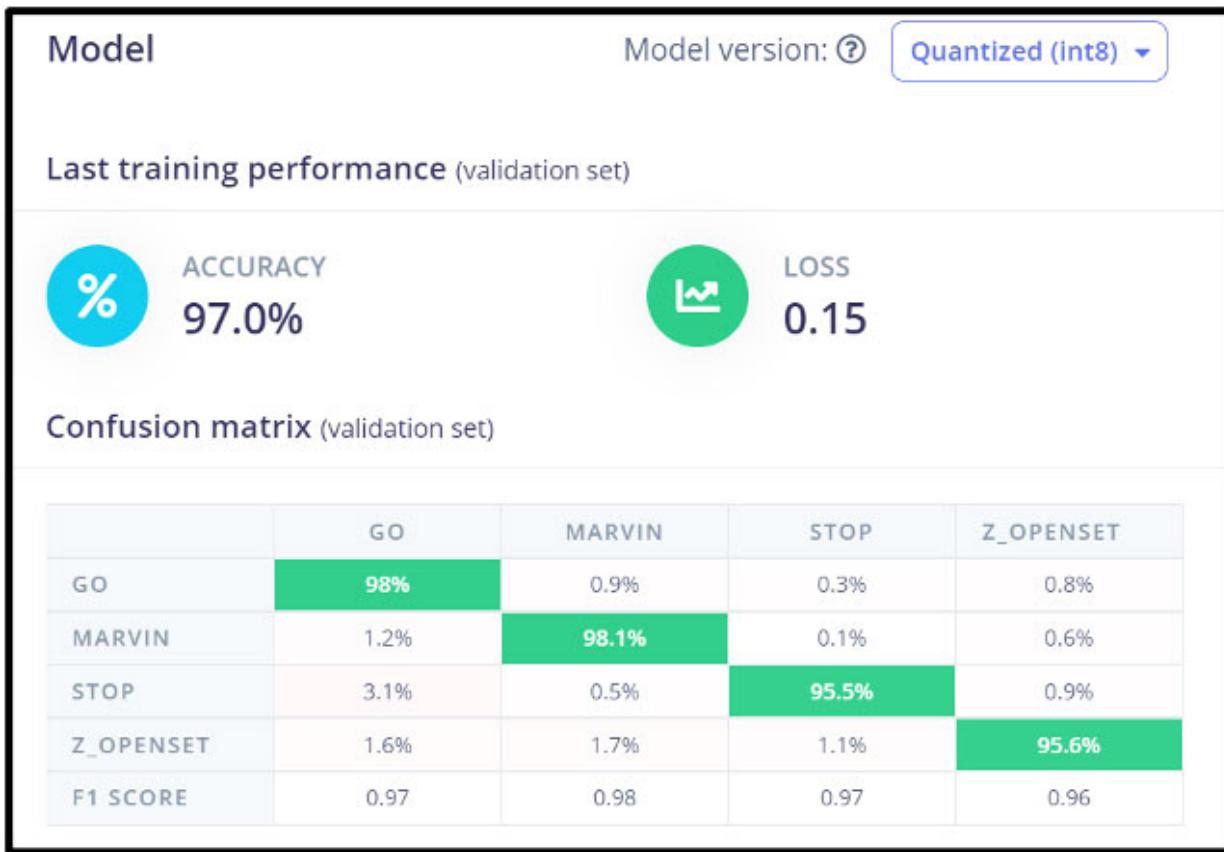


Figure 8.36: Re-training accuracy results with noisy data in the training set

[Figure 8.37](#) compares two models which were trained with and without noisy dataset. In this test, we used the same data set for testing. Now the results improve and are at the same level as before. This means that this neural network size can handle the diversity increase due to noise. Users are encouraged to add different type of background noise in both training set and test set and find the limit of a particular neural network. Please refer to the following figure:



Figure 8.37: Test accuracy improves with test vectors with noise and the training is done with some data set with noise

As seen in [Figure 8.37](#), the performance of `z_openset` was not good. The `z_openset` test set contains utterance of 20 words each of “five”, “siz”, “seven”, “eight” and “nine”. This is a hard test as the `z_openset` did not contain any of these words explicitly. To improve the performance, roughly 50 to 100 keywords from each of the Kaggle speech command sets were added in the `z_openset` along with some segments of noise totaling about 2000 new tensors. The dataset in the new `z_openset` training data included words which are also in the `z_openset` test samples (“five”, “siz”, “seven”, “eight” and “nine”). This increased the dataset of the `z_openset` as shown in [Figure 8.38](#) and now all four classifiers are more balanced. The accuracy dropped a little overall because many of the keywords are difficult to differentiate from each other:

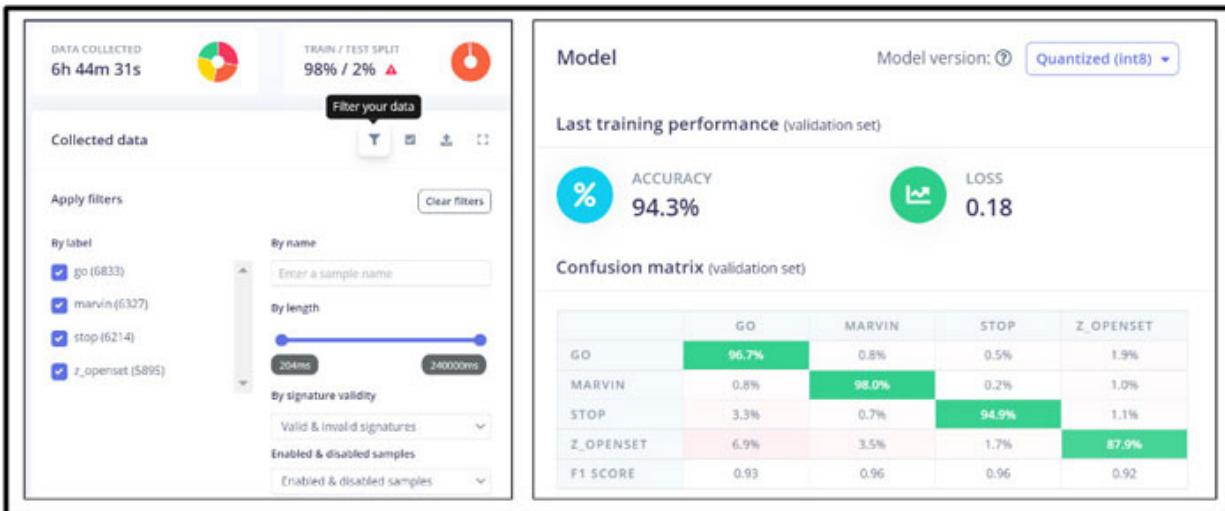


Figure 8.38: Training result while using balanced dataset in which `z_openset` includes more keywords from the Kaggle speech command

Though the training accuracy dropped, the testing accuracy improved as shown in [Figure 8.39](#). The result dramatically improved for the open set from 37% to 67%. There is a little bit of a penalty in accuracy for the target classifiers. By adding more keywords in the open set, 67% accuracy can further be improved. Please refer to the following figure:

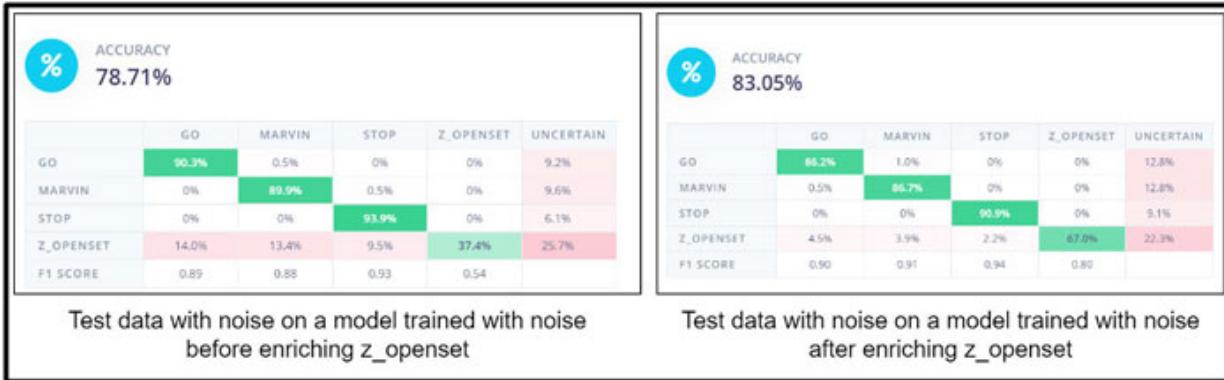


Figure 8.39: z_openset accuracy improves with enriched data set

By now, it is shown that the poor results can be analyzed and something can be done to improve the accuracy. As the data set grows, it takes more and more compute time and improvement reaches terminal value. In the next section, we will see how to collect data from the field instead of just augmenting the data we started with. It may be advantageous to discard augmented data when real field data is present to keep computation under control. By using data with the targeted background noise, accent, pace, tone and so on, the field performance can be improved significantly.

Data collection for continuous improvement

The AI system should be designed such that it can collect the data in the field, which can then be used in the future for better models. The system should have enough buffer, such that when unexpected inference occurs, the historical data can be saved, as it may not be possible to recreate the same situation again.

Let us take audio as an example: if the inference is being done on 1 second tensor, then the data collection should be about 2 seconds to account for some delay in the processing and data collection. Longer data collection will always be beneficial.

When data is collected, its file name should be unique and it should be tagged properly so its uniqueness can be tracked. The tag should be descriptive so it is easy to trace back.

Privacy laws should be observed while collecting and distributing data. The process of retraining should be continued over the life of the product for continued better performance.

Conclusion

In this chapter, we learned that it is very likely that the raw accuracy or error performance from the first neural network training is not acceptable. In this chapter, we showed some tricks and techniques to improve the system level performance.

It is also shown how the training data set can be strategically enriched to improve the error rate. Since AI is data driven, continuous increment in the training data is a norm for continuous improvement of the model. Most AI systems deployed today are improved over time after first deployment. Some pointers are proposed on how to collect the data from the field for continuous improvement to keep computation cost under control.

Key facts

- The raw performance of a neural network may not be acceptable due to overfitting concerns.
- Traditional techniques could be used to bridge the gap between the raw neural network performance and customer expectation.
- Both the shelf utilities, for example, Edge Impulse studio and custom Python scripts, can be used for the analysis.
- Optimization should be done keeping both FRR and FAR in mind.
- Data augmentation is a widely used method to generate more data which will provide immunity to certain environmental issues, for example, background noise, reflections, and so on.
- Ultimate performance could be improved by the field data. Therefore, a methodology should be planned to collect field data, especially which is prone for errors while obeying local privacy laws.

Questions

1. Neural networks are perfect and always produce superior results than natural human intelligence. True or False?
2. There is nothing that can be done to improve neural network performance. True or False?
3. If the data is not collected with environmental variables, such as background noise or reflection, then nothing can be done except collecting more data. True or False?
4. FRR and FARs can be individually optimized. True or False?

References

1. <https://www.edgeimpulse.com/>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



CHAPTER 9

Conclusion

Introduction

This book is intended for someone who is new to this field, and it is aimed not only to give people an overview what the AI is all about, but also takes concrete examples where the readers can make a meaningful project.

The AI field is in its infancy, and it is rapidly growing. It will take several books to capture material which covers various aspects of AI. As the field is maturing, several academic papers are presented every day. While the AI field is ever evolving, the practical use of AI is not that complex. The ecosystem is forming and is reducing the complexity for an average user, so that they can be productive right away.

In this chapter, we will review what we have learned so far and we will also go over some of the topics in which reader will be interested next.

Structure

In this chapter, the following topics will be covered:

- Review of material covered in this book
- Advanced topics
 - Different type of neural networks
 - Neural network optimization
 - Zero-shot, One-shot or Few-shot learning
 - Federated learning
 - Tuning pretrained networks
 - MLOps

Objectives

In this chapter, we will first review what we have learned in the preceding chapters. Then we will take some time to discuss some of the advanced topics which were not in the scope of this book. The premise of AI is to automate intellectual work in the same way that manufacturing today is automated. In this chapter, we will see what gets in the way of deploying AI and how to deal with it.

Review of material covered in this book

This book introduces deploying AI models at the edge. The book is divided into 8 chapters which discuss different topics leading ever improving AI deployment. Let us recap what we learned in different chapters.

Chapter 1

In this chapter, we learned what to expect in an AI system flow. We learned that AI systems run on neural networks which are inspired from the biological brain. Despite orders of low complexity, the mathematical model of neural networks starts showing attributes of intelligence. This intelligence is limited to pattern recognition which can be applied to some problem solving. We discussed some of the use cases of deploying AI on the edge.

Chapter 2

In this chapter, we learned the traditional methods to solve complex problems, different types of ML categories and its performance metrics being used. We have also learned popular DL algorithms being used for solving problems in computer vision and other domains. We learned different tools, libraries and frameworks used for development and deployment of ML models, on various embedded devices and microcontrollers. We also learned the differences between the learning and inference using ML models at different levels of deployments.

Chapter 3

In this chapter, we learned about the major and important TinyML hardware and software platforms. We understood the differences among CPUs, GPU, TPU, Raspberry Pi boards and servers at data Centers. We also learned about the different types of Raspberry Pi boards, Microcontrollers and AI accelerators and their advantages for MC boards are studied. We learned about the different TinyML hardware boards and software tools, explored different categories from multiple vendors and their main important features. We have also understood why ML model compression is needed and the benefits of model compression techniques.

Chapter 4

In this chapter, we learned about **Embedded Machine Learning (EML)**, its characteristics, different types and sample example systems. We learned in detail the Edge Impulse and Arduino IDE tools usage and different components in it. We studied different approaches in data collection steps from sensors and TinyML tools. In data engineering, we learned about the topics such as data cleaning, and transformations required on the data. Model training using EON compiler is studied. Model compression, especially pruning and knowledge distillation topics are focused, and model conversion is understood in detail with the understanding of different quantization techniques. Model inferencing and deployment techniques on different hardware boards is understood with the help of Edge Impulse Studio.

Chapter 5

In this chapter, we took a detailed look at multiple use cases where AI is already deployed. We also presented some ideas where AI could be applied. The use cases are broken down into seven categories. Some ideas could be applied among many categories, for example, predictive maintenance for equipment could be applied in agriculture, industrial or automotive settings. Different use cases are presented to inspire readers for their own ideas. The AI is in infancy and so we expect to see several new use cases to emerge in next two decades.

Chapter 6

In this chapter, we learnt about implementing air gesture digit recognition using Arduino Nano RP2040 board. A brief introduction about the Nano RP2040 TinyML board is given. We implemented the use case from scratch and focused mainly on the steps involved from the data collection using Nano RP2040 board and Arduino IDE for different gestures, cleaning the dataset and creating a project in Edge Impulse platform, uploading the datasets for training and testing purpose. After uploading the datasets, we selected the required architecture (setting the development framework) for the classification and performed the training activity on the datasets. We tested the algorithm and built the final model that is to be deployed in the Nano RP2040 board for the final inference. The required modifications are performed for the built code to deploy it on the Nano RP2040 board.

The final inference results are observed in real time for different air gesture digits for the deployed code on the board.

Chapter 7

In this chapter, we learned about some advance platforms where heavy duty AI can be deployed without writing a single line of code. AI is data driven which means that the code does not have to be written using traditional rule-based logic. Since most of the routines will be used across the multiple use cases, it does not make sense for users to be writing the code in isolation. The echo system is developing, where utilities could be either used as a free service or with small subscription fees. It is encouraged to use the tools available and augment them, if need be, as compared to writing the code from scratch.

A concrete example was taken to show all the steps to deploy a model in the hardware without writing a single line of code.

Chapter 8

In this chapter, we took a deeper dive to see intricacies of working towards a production level AI model. Though it is a child's play to develop a demo model, it is not trivial to make a model which can be called a production model. A production model needs to be driven by specifications around false rate of acceptance and false rate of rejection.

The model should be tolerant to expected background noise. It shows how the data should be collected and be augmented. It is an acceptable practice to improve the performance of the model once the model is deployed, based on the field data. It is recommended to keep provision to collect the data from the field and improve the model while respecting privacy. We will see how federated learning is used to improve the model while providing the privacy later in this chapter.

Advanced topics

By now, it is expected that the reader could make a complete keyword recognizing system from scratch. The same methodology can be applied for image recognition, video analytics, sensor analytics and so on. However, the user may find that the performance is not adequate. The advanced topics will give some insight about all that can be done to further improve the performance.

Different types of neural networks

So far, we have used the very basic neural network which is fully connected. This is simple to understand and implement, and works quite well when inputs are under ten thousand parameters. Consider a high-definition image with 4K UHD resolution. 4K UHD resolution is specified as 3840 numbers of horizontal pixels and 2160 numbers of vertical pixels. Considering red, green and blue colours, the number of input values are $3 \times 3840 \times 2160 = 24,883,200$. If we want to have similar numbers of outputs, then we are looking at over 600 billion parameters in just one layer. This does not seem manageable. To reduce the number of parameters, **Convolution Neural Networks (CNN)** are used. In a CNN, a small area of the input tensor is processed and part of another tensor is created. For colored images, smallest

convolution layer could utilize 3 (color) x (3 x pixel) x 3 (y pixels) input. Then the area is swept in X/Y with certain stride. If a stride of 2 is used, this will reduce the next tensor size to be roughly one forth. In each successive layer, the tensor size will become smaller and smaller. When the size of tensor becomes 100 x 100, then the dense layer could be used.

In certain applications where the time history is important, for example, long speech signal, then a **Long Short-Term Memory (LSTM)** network is utilized. Due to recursive feedback, LSTM is a **Recurrent Neural Network (RNN)**. **Gated Recurrent Unit (GRU)** is another type of RNN.

Sometimes, multiple types of neural networks are used to combine their results. These are termed as ensemble neural networks.

An encoder and decoder combination of neural network could be used for the anomaly detector. In this neural network, outputs are matched to the input for all expected inputs. Then a difference is calculated between input and output. For an unknown input, the output will be unknown and thus will be different from input. The difference between input and output could be detected and with a threshold, anomaly alarm can be triggered.

[Figure 9.1](#) features the Neural Network Zoo showing the different types of cells and layer connectivity styles:

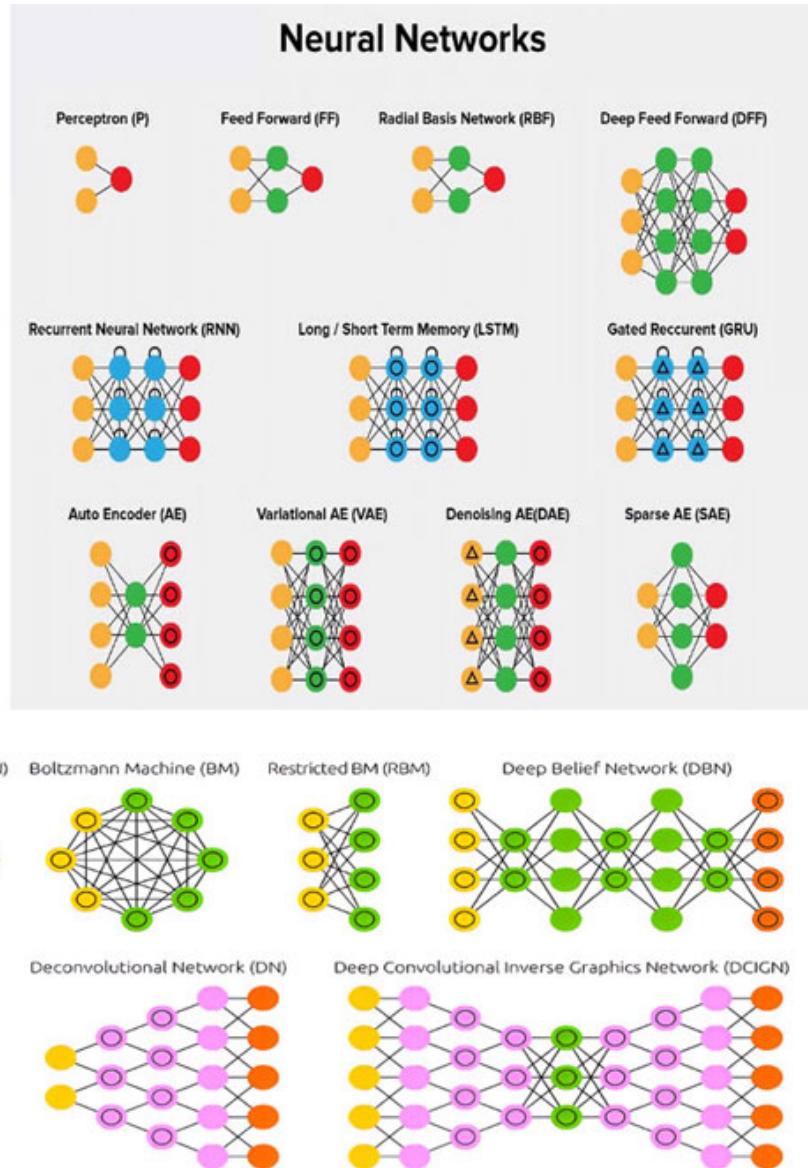


Figure 9.1: The Neural Network Zoo^[3] shows different types of cells and various layer connectivity styles for Neural Network Algorithms

Neural network optimization

The process of modifying the parameters of a neural network in order to minimize the error between the network's predictions and the ground truth is referred to as neural network optimization. The optimization process begins with the initialization of the network's weights and biases. Weights and biases are then iteratively changed based on the difference between the network's predictions and the ground truth. A loss function is often used to determine the error, which measures the difference between the anticipated

output and the ground truth. The optimization algorithm then computes the gradient of the loss function with respect to the weights and biases, and updates them in the opposite direction of the gradient. This process is repeated until either the error is minimized or the optimization algorithm reaches a stopping point.

After neural network is trained, it can be pruned. When the parameters are sparse, then parameters with zero value or near zero values are removed, thus reducing the parameters and compute time.

Parameters can also be compressed to save space to store them.

Truncating the parameters to fixed point reduces the compute time at the expense of small and sometimes unnoticeable inaccuracy. In some extreme cases, the parameters are converted to binary parameters, thus reducing the computation significantly.

Zero-shot, One-shot or Few-shot learning

Zero-shot learning is a type of machine learning where the model is able to classify new objects into categories that it has never seen before, based on a description or representation of these categories. This is done by training the model on a set of classes and a set of attributes that describe these classes. At test time, the model is presented with an unseen object and a description of the object's attributes, and it must predict the class of the object based on the attribute description and the training data.

Figure 9.2 provides a Zero-shot learning example:

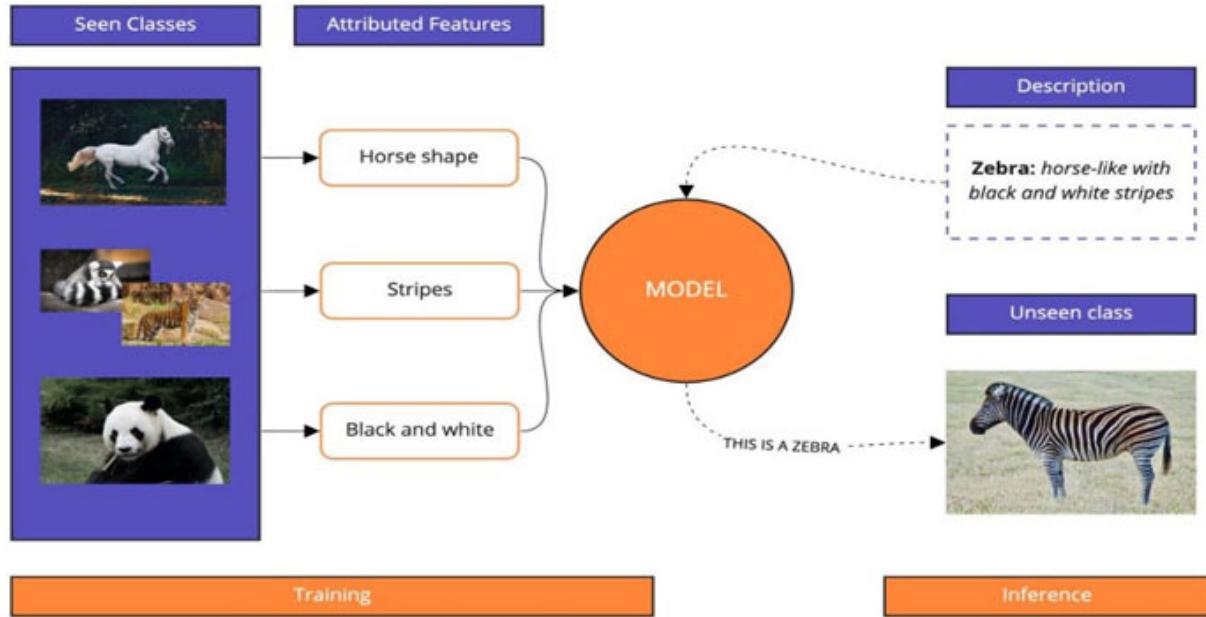


Figure 9.2: Zero-shot learning example [2]

One-shot learning and few-shot learning are ways to teach machine learning models how to learn from a small number of examples and make predictions. When there is not much data to train with, these methods are useful because they can still get good results with a small amount of data. One-shot learning is when a model is trained on a single example or a small number of examples, and then this information is used to make predictions about data it has never seen before. Most of the time, this is done by learning a similarity function between the examples given and the new data, and then making predictions using this function. To train a model to recognize different sorts of animals, for example, we would only submit a single image of each species as training data. In contrast, classic machine learning approaches often require a significant amount of labelled training data to perform successfully.

Few-shot learning is like one-shot learning, but it involves making predictions about new data based on a small number of examples (usually less than 10). There are many ways to do this, such as using metric learning or a model with a lot of parameters that can be fine-tuned, based on a small number of examples.

Both one-shot and few-shot learning are useful when it is hard or impractical to collect a lot of training data, such as in areas where data is difficult to obtain or costs a lot to get. They are also useful for tasks where the

underlying distribution of the data may change over time because the model can quickly adapt to new data with just a few examples. The amount of training examples given for each class is the primary distinction between few-shot and one-shot learning.

When training is done from scratch, it requires a lot of data and compute power. It is possible to use multilayer neural networks with lower layers already pretrained. Then the last few layers can be trained with few shots.

In general, zero-shot, one-shot, and few-shot learning are all methods of learning from very few or even a single example. These methods are useful in situations where it is difficult or impractical to gather a large dataset for every class that the model needs to recognize.

Federated learning

Federated Learning (FL) is a machine learning technique that lets models be trained on decentralized data, like data that is spread across multiple devices or locations. This can be helpful when it is not possible or practical to collect and centralize data for training, such as when the data is sensitive or the devices do not have enough resources.

We have seen in [*Chapter 6, Practical Experiments with TinyML*](#), and in [*Chapter 8, Continuous Improvement*](#), that models can be improved over time. In some cases, either it is not possible, or it is not preferred to transmit the data back to cloud to get the next version of the model, due to privacy reasons. However, the models can be improved in a distributed manner where they are deployed. Once the models have been improved locally, the models are sent to cloud where an aggregate model can be generated. In this manner, privacy is preserved while learning is accomplished from the data.

In the case of TinyML, which is machine learning on small devices, FL can be especially helpful because it lets the model be trained on data that is created and collected directly on the devices, instead of having to send the data to a central server. This can save time and protect your privacy because it cuts down on the amount of data that needs to be sent and processed.

FL can also help TinyML models work better by letting them be trained on bigger and more diverse datasets. This can be especially important when the data are noisy or not balanced, because it can help the model work better in the real world.

An example of federated learning in image processing would be training a model to classify animal images. Suppose you have a dataset including photographs of different animals, but the data is spread over multiple devices and places. Instead of trying to gather all of the data into a single place and build a model on it, you could utilize federated learning to immediately train a model on the distributed data.

Here is one possible scenario: you would begin by selecting a collection of devices or locations that have data relevant to the classification of animals. The model would then be sent to each of these devices or locations. Each device or location would utilize its own data to locally train and update the model's weights and biases. After a predetermined number of training cycles, each device or location would transmit its modified model to a central server. The central server would then collect the updates from all devices or locations and utilize them to update the global model. The procedure would then be repeated, with the revised global model being transmitted back to the devices or locations for additional local training. This process would continue until the model converges, at which point it would be capable of making accurate predictions based on new data. [Figure 9.3](#) features federated learning along with an example for animal classification:

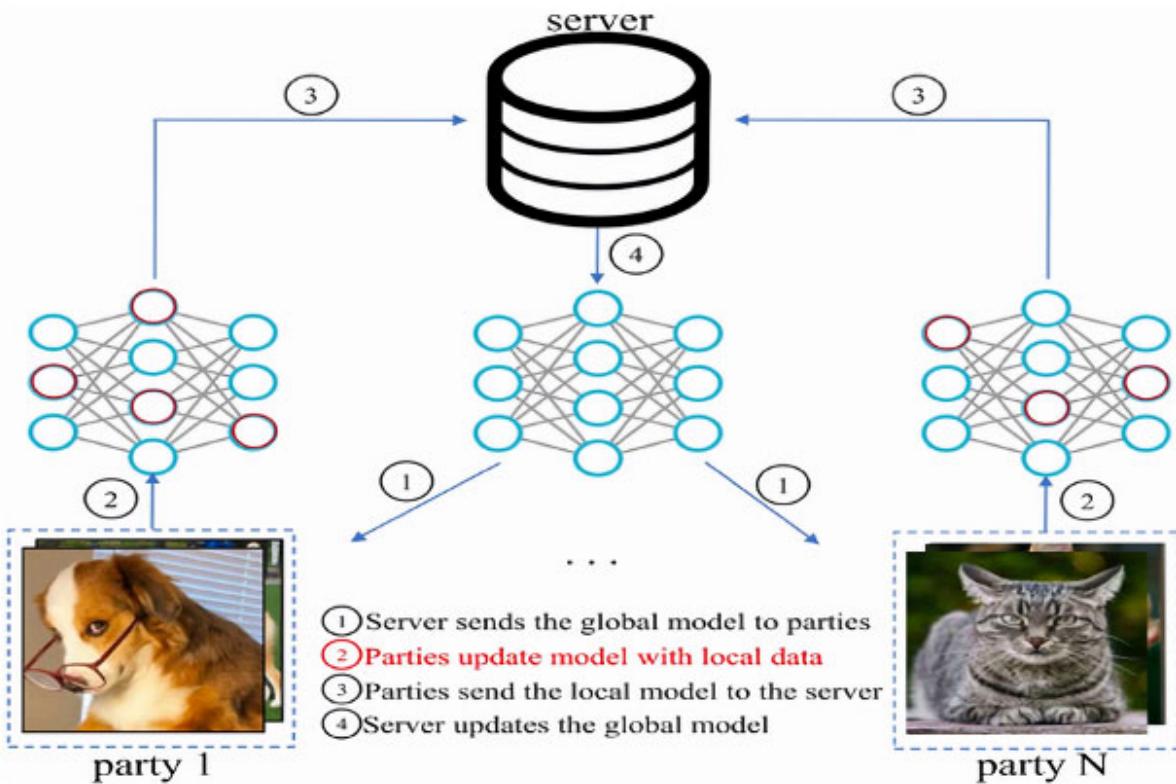
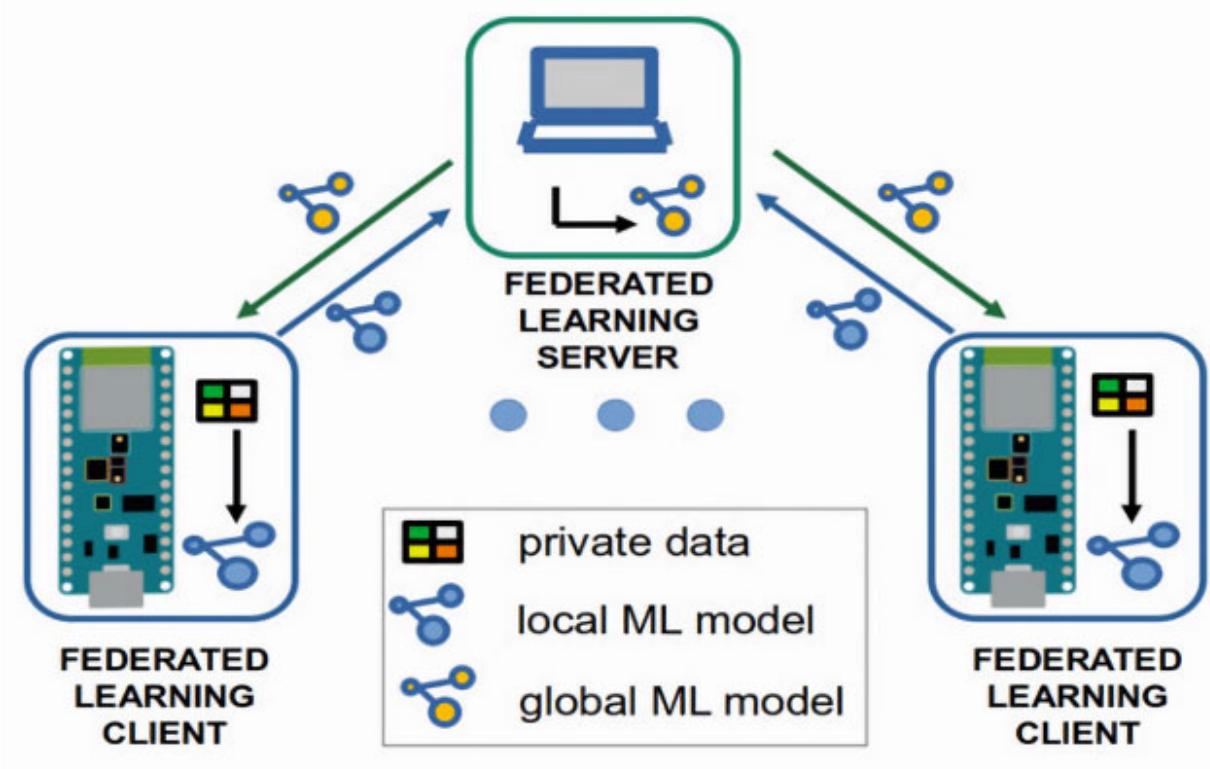


Figure 9.3: Federated Learning and its example for animal classification

Overall, FL has the potential to be a powerful tool for making TinyML applications that work well and efficiently. This is especially true when data privacy and limited resources are important factors.

Federated learning has various advantages:

- **Privacy:** Because FL does not require data to be shared or centralized, enterprises can keep their data private. This is particularly beneficial for firms that deal with sensitive data, such as healthcare providers and financial institutions.
- **Data sovereignty:** FL enables enterprises to maintain control over their data, which is critical for complying with data sovereignty laws and regulations.
- **Improved model performance:** FL can boost a machine learning model's performance by allowing it to learn from a bigger and more diverse dataset.
- **Reduced communication costs:** Because each party just needs to communicate model changes to the central server rather than the complete dataset, federated learning can reduce the quantity of data that needs to be transmitted across a network.
- **Improved scalability:** FL improves scalability by allowing businesses with huge datasets to train machine learning models without the requirement for powerful hardware and infrastructure.

Transfer learning

Sometimes a pretrained model needs to be fitted in a smaller parameter/compute footprint. To accomplish this, learning is transferred from a bigger model to the smaller model.

In transfer learning, a model trained on one task serves as the basis for a model trained on a different task. The concept is that the lower layers of the model, which learn more general features, can be reused as-is, and the upper layers, which acquire more task-specific features, can be trained or adjusted to the current task.

Here is an illustration of transfer learning in image processing task as an example.

Imagine you have a large collection of images of animals, and you want to develop a model capable of categorizing the animals into several groups (for example, dogs, cats, and birds). You could train a deep **Convolutional Neural Network (CNN)** from scratch on this dataset, but it would take a significant amount of time and data to achieve high-quality results.

As an alternative, you might utilize a CNN model that has already been trained on a big image dataset (such as ImageNet) as the basis for your model. This pre-trained model has already learned to recognize many fundamental image features (such as edges, textures, and patterns), and therefore, its lower layers can be used as a feature extractor for your own dataset. The higher layers of the model would then be trained or fine-tuned using your animal dataset to learn task-specific attributes for animal classification.

Using a pre-trained model as a starting point for your own model can significantly minimize the quantity of data and time required to train the model, and can often result in superior performance. This is particularly true when the new assignment resembles the original task for which the pre-trained model was developed.

Figure 9.4 features a comparison between traditional and transfer Machine Learning/ Learning.

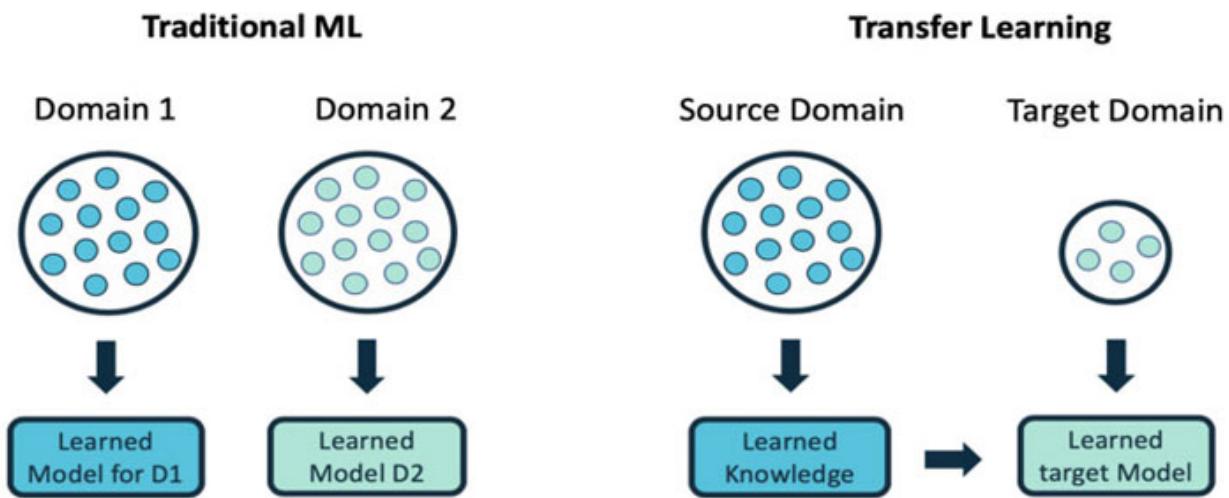


Figure 9.4: Comparison of Traditional ML vs Transfer Learning

Tuning pretrained networks

Tuning a pre-trained network means fine-tuning the weights of a model that has already been trained on a new dataset. This is done to make the model work better with the new data set, by adjusting the weights of the model that was already trained to fit the new data.

There are many ways to fine-tune a network that has already been trained. Transfer learning is one way to do this. With this method, the weights of a model that has already been trained are used as the starting point for training a new model. Only the final layers are trained from scratch. Fine-tuning all the layers of a model that has already been trained is another option. This is called "fine-tuning from scratch."

The learning rate is an important thing to think about when fine-tuning a network, that has already been trained. When fine-tuning a pre-trained network, it is usually best to use a slower learning rate. This is because the weights of the pre-trained model already contain a lot of information about the problem, and we do not want to make big changes to these weights.

Another important thing to think about is which pre-trained model to use. Most of the time, it is best to choose a model that has already been trained and is similar to the problem at hand. This lets you get the most out of the transferred knowledge. For example, if the new dataset is made up of images, it would make sense to start fine-tuning with a model that has already been trained to classify images.

Assume you wish to train a neural network to recognize photographs of dogs and cats. To begin, you may use a pretrained network that has already been trained to classify a wide range of images. This pretrained network would have previously learned image classification features including edges, textures, and patterns. To fine-tune this pretrained network for your specific goal of classifying dogs and cats, you would "freeze" the pretrained network's weights so that they do not change during training. Then, at the network's end, you'd add a couple more layers that would be randomly initialized. These extra layers will be trained to recognize the precise characteristics that separate dogs from cats.

Finally, using a dataset of images of dogs and cats, you would train the entire network, freezing the pretrained layers and updating the newly added layers. Because the pretrained network has previously acquired features suitable for a wide range of image classification tasks, fine-tuning the weights of a

pretrained network can result in higher performance than training a network from scratch.

Consider another example; a **FaceNet** neural network could be trained with a large population in one country. Then the network can be tuned to work in another country where skin tone and some of the facial features may be different. This methodology not only takes less data and compute power, the model is very robust, as shown in [*Figure 9.5*](#):

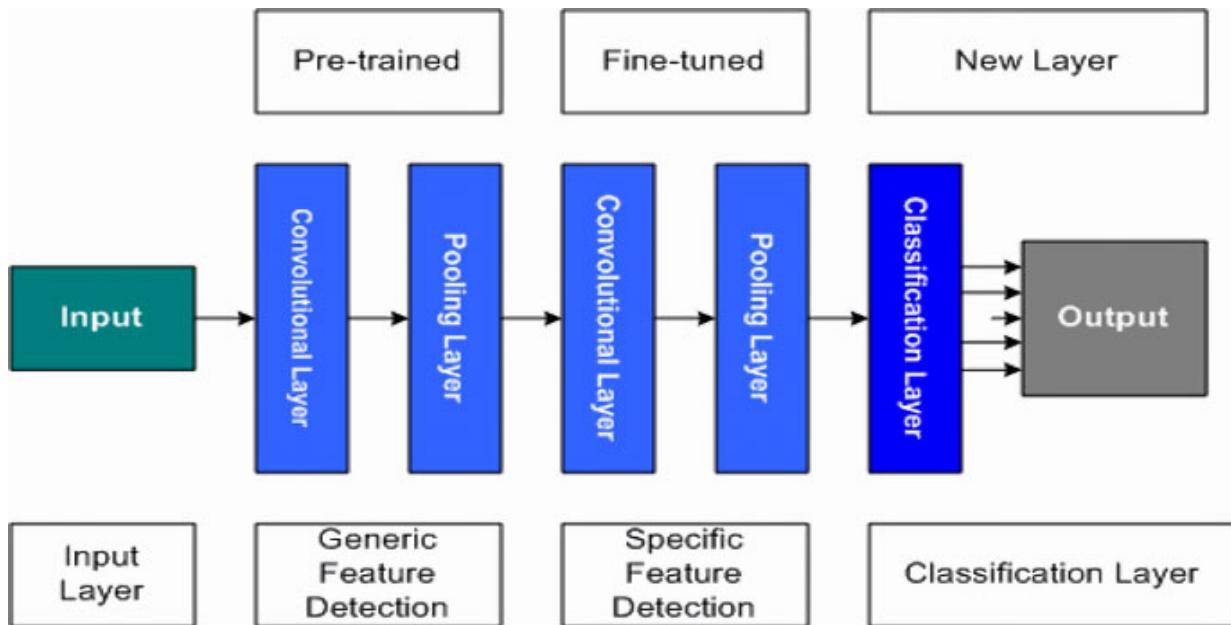


Figure 9.5: Fine-tuning the specific feature extraction layers

In conclusion, fine-tuning a pre-trained network is a good way to improve a model's performance on a new dataset. This method requires careful thought about the learning rate and the choice of the pre-trained model.

MLOps

MLOps, which stands for **Machine Learning Operations**, is a practice that tries to standardize and automate the process of building testing, deploying, and maintaining machine learning models. It is the practice of merging the ideas of software engineering and data engineering, in order to increase the collaboration, communication, and productivity of teams working on machine learning projects. One example tool for MLOps is MLflow which is an open-source platform for managing the end-to-end machine learning lifecycle, including experiment tracking, reproducibility, and deployment.

Key facts

- Different types of neural networks could be better suited for different application. For example, LSTM network could be better suited for speech recognition, while CNN is better suited for image processing.
- Neural networks could be pruned to reduce the complexity.
- It is possible to use pretrained networks to optimize for slightly different use case with relatively small amount of data.
- The primary distinction between zero-shot learning, one-shot learning, and few-shot learning is the number of examples of each class that the model is trained on. This is true for all three types of learning.
- Federated learning is a way to improve the models over time while providing privacy of the users.
- MLOps aims to simplify the process of building, deploying, and maintaining machine learning models in a production environment.

Questions

1. What is the biggest bottleneck in deploying an AI model?
2. What is the difference between a demo model and a production model?
3. Can a model be improved over time once it is deployed? What are the key activities required to improve the model?
4. What is the key concern in collecting more data to improve the model?
5. How could a smaller set of data be required to produce production level model?
6. Is it true that one neural network will be best for different use cases?
7. Why we need to deploy AI on the edge, even though it can be easily deployed in the cloud? Give three reasons.
8. What is MLOps? List the tools used for MLOps.

References

1. **Types of Neural Networks | Top 6 Different Types of Neural Networks** ([educba.com](https://www.educba.com/types-of-neural-networks/)), [https://www.educba.com/types-of-neural-](https://www.educba.com/types-of-neural-networks/)

networks/

2. <https://modulai.io/blog/zero-shot-learning-in-nlp/>
3. How do zero-shot, one-shot and few-shot learning differ?:
<https://analyticsindiamag.com/how-do-zero-shot-one-shot-and-few-shot-learning-differ/>
4. <https://www.asimovinstitute.org/neural-network-zoo/>
5. <https://developer.ibm.com/articles/transfer-learning-for-deep-learning/>

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

<https://discord.bpbonline.com>



Index

A

accuracy [11](#)
Acoustic Event Detection (AED) [68](#)
Adafruit Circuit Playground Express [59](#)
Adafruit EdgeBadge [58](#)
Adafruit Feather [61](#)
AI Model Efficiency Toolkit (AIMET) [38, 39](#)
Analog to Digital Converter (ADC) [197](#)
anomalous behavior, target classifier testing
 handling [229, 230](#)
open set classifier, enriching [232-237](#)
running window averaging [230, 231](#)
target classifier, enriching [231](#)
Arduino [59](#)
Arduino Create [70, 72](#)
Arduino IDE [109, 110](#)
 advantages [114](#)
 Arduino Driver Installation [111-114](#)
 disadvantages [115](#)
Arduino Nano [33](#) BLE Sense [14, 15, 58, 59](#)
 components [60](#)
Arduino Nicla Sense ME [58-60](#)
Arm Cortex M4F microprocessor [62](#)
Arm Cortex-M microcontroller-based machine learning [70](#)
Artificial Intelligence (AI) [1-3](#)
 hypothetical problem example [3-5](#)
 paradigm, changing [5-7](#)
audio application
 unique issues [226, 227](#)
AU-ROC [31](#)
Autonomous Vehicle (AV) [44](#)

B

backpropagation method [11](#)
Berkeley AI Research (BAIR) [39](#)
Bluetooth Low Energy (BLE) [13, 59](#)

C

Central Processing Unit (CPU) [52](#)
Clustering [29](#)
CMix-NN [89](#)

code writing
continuum [3](#)
Comma-Separated Values (CSV) file [119](#)
Complex Instruction Set Computer (CISC) [18](#), [54](#)
Computer Vision (CV) [40](#), [51](#)
confusion matrix [30](#), [31](#)
 AU-ROC [31](#), [32](#)
 F1-score [31](#)
 False Negative (FN) [31](#)
 False Positive (FP) [31](#)
 precision [31](#)
 recall [31](#)
 True Negative (TN) [31](#)
 True Positive (TP) [31](#)
Convolutional Architecture for Fast Feature Embedding (Caffe) [39](#), [40](#)
Convolutional Neural Networks (CNNs) [40](#), [261](#)
Coral Edge TPU [58](#)
Core ML [40](#), [41](#)

D

data center [52](#)
data collection
 for continuous improvement [255](#)
 from Arduino board [116](#), [117](#)
 from multiple sensors [115](#), [116](#)
 from Syntiant board [117](#)
data deduplication [119](#)
data engineering [117](#)
data engineering, for TinyML
 data cleansing [118](#)
 data organization [119](#)
 data transformation [119](#)
Data Engineering Frameworks
 Edge Impulse [78](#), [79](#)
 Qeexo AutoML [81](#)
 SensiML [79](#), [80](#)
data transformations
 data deduplication [119](#)
 encoding [119](#)
 feature extraction [119](#)
 imputation [119](#)
 normalization [119](#)
Deep Learning (DL) [32](#), [33](#)
 algorithms [34](#)
Digital Signal Processing (DSP) algorithm [184](#)
Digital Signal Processing (DSP) block [199](#)
DL model training
 versus, inference [44](#)
Do-It-Yourself (DIY) projects [54](#)

E

Edge Computing [45](#)
Edge Impulse [70](#), [78](#), [99](#), [100](#), [199](#)
advantages [102](#)
Arduino Nano RP2040 Connect [101](#)
Edge Impulse CLI [101](#)
Edge Impulse Studio [100](#)
features, for model compression [125](#), [126](#)
input block [177](#)
learning block [177](#)
ML model compression [126](#)
ML model deployment [130](#)
ML model testing [128](#)
processing block [177](#)
Syntiant TinyML boards [101](#)
usage [102-107](#)
Edge Impulse Machine Learning (ML) accelerator [62](#)
EdgeML (Microsoft) [72](#)
EloquentML [72](#)
Embedded Machine Learning (EML) [42](#), [43](#)
Embedded ML [95](#), [96](#)
advantages [97](#), [98](#)
disadvantages [98](#)
examples [96](#), [97](#)
features [96](#)
end-to-end TinyML deployment [93-95](#)
EON Compiler (Edge Impulse) [73](#), [120](#)
model training, with [120](#)
epochs [11](#)
expectation gap [226](#)

F

False Acceptance Rate (FAR) [230](#)
testing [238-240](#)
False Rate of Rejection (FRR) [231](#)
performance improving, under noisy conditions [251-255](#)
testing, under noisy conditions [249-251](#)
feature extraction [119](#)
Federated Learning (FL) [264-266](#)
advantages [266](#), [267](#)
Federate Learning (FL) [45](#)
few-shot learning [264](#)
Field-Programmable Gate Arrays (FPGAs) [70](#)

G

Gated Recurrent Unit (GRU) [261](#)
Glyphosate Based Herbicides (BGHs) [137](#)

Google Colaboratory [36](#)
Google Coral [59](#)
Google Coral Edge TPU [63](#)
 features [64](#)
Graphical User Interface (GUI) [199](#)
Graphics Processing Unit (GPU) [52](#)

I

Impulse Design [203-208](#)
 auto balance setting [210](#)
 data augmentation [210, 211](#)
 epochs setting [209](#)
 learning rate setting [209](#)
 neural network architecture [211, 212](#)
 neural network training [212-216](#)
 validation data set setting [209, 210](#)
imputation [119](#)
Inertial Measurement Unit (IMU) [60, 142](#)
inference phase [43](#)
inferencing [126](#)
 performing [127](#)
Input Holding tank [199](#)
integrated circuit (IC) [55](#)
Intel Curie [67](#)
 features [68](#)
Intelligent Internet of Things (IIoT) [13](#)
Intelligent IoT system
 Arduino Nano [33](#) BLE Sense board [14, 15](#)
 key applications [19, 20](#)
 limited compute resources [15](#)
 Nicla Voice board [17](#)
 TinyML board [16](#)
 versus cloud based IoT system [13, 14](#)
Internet of Things (IoT) [12, 25, 51](#)

J

Jetson Nano [62](#)
 features [63](#)
Jupyter Notebook [36](#)

K

Keras [86](#)
key applications, Intelligent IoT system [19, 20](#)
 smart agriculture [20](#)
 smart appliances [20](#)
 smart cities [20](#)
 smart health [21](#)

smart homes [21](#)
smart industry [21](#)
knowledge distillation [83](#), [84](#), [123](#), [124](#)
advantages [123](#)

L

learning phase [43](#)
limited compute resources
 battery power limits [15](#)
Long Range Radio (LoRa) [13](#)
Long Short-Term Memory (LSTM) network [261](#)
loss [11](#)
low-ranked approximation [83](#)
low-rank factorization [83](#), [123](#)

M

machine learning (ML) [11](#), [12](#)
 interface phase [43](#)
 learning phase [43](#)
 life cycle [25](#)
 performance metrics [30](#)
 reinforcement learning (RL) [30](#)
 supervised learning [29](#)
 unsupervised learning [29](#)
Machine Learning Operations (MLOps) [270](#)
Metavision Intelligence Suite 3.0 [78](#)
microcontrollers [55](#)
 8-bit microcontrollers [56](#)
 16-bit microcontrollers [56](#)
 32-bit microcontrollers [56](#)
 advantages [55](#), [56](#)
 input/output (I/O) peripherals [55](#)
 with AI accelerator [55](#)
Microcontroller Unit (MCU) [14](#)
MicroPython [70](#)
Microsoft NNI [87](#)
ML algorithms [27](#), [28](#)
ML model compression techniques
 knowledge distillation [83](#), [84](#)
 low-ranked approximation [83](#)
 pruning [83](#)
 quantization [82](#)
ML model compression tools
 CMix-NN [89](#)
 Microsoft NNI [87](#), [88](#)
 OmniML [89](#)
 QKeras [86](#)
 Qualcomm AIMET [86](#), [87](#)

STM32 X-CUBE-AI [85](#)
TensorFlow Lite [84](#), [85](#)
ML model deployment
in Edge Impulse [130](#)
ML models training
versus inferencing [44-46](#)
ML model testing
in Egde Impulse [128](#)
mobile CPU [53](#), [54](#)
model compression
Egde Impulse, using [124](#)
knowledge distillation [123](#), [124](#)
pruning [122](#), [123](#)
model conversion [124](#)
quantization [124](#)
model deployment
in TinyML hardware board [128](#)
model pruning [83](#)
model quantization [82](#)
activation quantization [82](#)
weight quantization [82](#)
model training
in TinyML software platforms [120](#)
with EON Compiler (Edge Impulse) [120](#)

N

NanoEdge AI Studio [74](#)
Nano RP2040 TinyML board experiment [163-169](#)
accelerometer, testing [164](#)
air gesture digit detection [165](#), [166](#)
Arduino IDE, setting up [163](#)
dataset, loading in Edge Impulse [170-173](#)
development framework, setting up [174](#)
gyroscope, testing [164](#), [165](#)
Nano RP2040 board, testing [163](#), [164](#)
neural network classifier design [174](#)
Narrow band IOT (NB-IOT) [13](#)
Natural Language Processing (NLP) [40](#), [51](#)
NDP101 architecture [196-198](#)
NDP120 architecture [199](#)
neural network [7-10](#)
types [261](#)
neural network optimization [262](#), [263](#)
Neural Network Zoo [262](#)
Neuton TinyML [77](#)
Nicla Sense ME [60](#)
normalization [119](#)
Nvidia Jetson Nano [62](#), [63](#)
NVIDIA Jetson Nano [59](#)

NXP i.MX 8M [65](#), [66](#)

O

OmniML [89](#)
one-shot learning [264](#)
OpenMV [70](#), [76](#)
Open Neural Network Exchange (ONNX) [41](#)
OpenVINO [41](#), [42](#)

P

phbackoff parameter [220](#)
phrase recognition constraints
for improving system level performance [246-248](#)
practical implementation and deployment
data, uploading [201-203](#)
Impulse Design [203-208](#)
model deployment [217-221](#)
model testing [216](#), [217](#)
project, creating [199](#), [200](#)
practical TinyML experiment
air gesture digit recognition experiment [166-169](#)
inferencing/prediction of results with RP2040 [188-192](#)
model deployment, in Nano RP2040 board [184-187](#)
model testing, with collected data [183](#), [184](#)
model training, in Edge Impulse platform [177-182](#)
Nano RP2040 TinyML board [163](#)
pre-trained network
tuning [268-269](#)
Principal Component Analysis (PCA) [29](#)
pruning [122](#)
low-rank factorization [123](#)
structured pruning [122](#)
unit pruning [123](#)
weight pruning [122](#)
Pulse Density Modulation (PDM) interface [197](#)
PYNNQ [70](#), [74](#), [75](#)
Python [36](#)
Pytorch [42](#)
PyTorch Mobile [42](#)

Q

Qeexo AutoML [81](#)
QKeras [86](#)
Qualcomm AIMET [86](#), [87](#)
Qualcomm QCS605 [64](#)
features [65](#)
quantization [124](#)

activation quantization [125](#)
dynamic fixed-point quantization [125](#)
quantization-aware training [125](#)
static fixed-point quantization [125](#)
weight quantization [124](#)

R

Random Access Memory (RAM) [14](#)
Raspberry Pi [59](#)
Raspberry Pi [4](#) [58](#)
Raspberry Pi boards [54](#)
components [55](#)
Raspberry Pi [2](#) Model B [54](#)
Raspberry Pi [3](#) Model A+ [54](#)
Raspberry Pi [3](#) Model B [54](#)
Raspberry Pi [4](#) Model B [54](#)
Raspberry Pi 400 [55](#)
Raspberry Pi Model B [54](#)
Raspberry Pi Model B+ [54](#)
raw neural network output [228](#)
Recurrent Neural Network (RNN) [261](#)
Reduced Instruction Set Computer (RISC) [18](#), [54](#)
reinforcement learning (RL) [30](#)
RoadBotics [143](#)
running window averaging
window size optimization [240-246](#)

S

SensiML [76-79](#)
servers [52](#)
servers, at data centers
CPUs [52](#)
GPUs [52](#)
TPUs [53](#)
smart agriculture [135](#)
acoustic insect detection [138](#)
agribots [136](#), [137](#)
agriculture video analytics [135](#), [136](#)
animal husbandry [139](#)
crop intruder detection [136](#)
crop yield prediction and improvement [136](#)
insect detection [137](#)
pesticide reduction [137](#)
weedicides elimination [137](#), [138](#)
smart appliances [139](#)
Audio AI, for appliances [141](#), [142](#)
Sensors based AI, for appliances [142](#)
Vision AI, for appliances [139-141](#)

smart automotive [156](#)
advance collision detection [156](#)
drowsy driver alert [156](#)
smart cities [142](#)
bridges, monitoring [144](#)
city maintenance [143](#)
non-smoking enforcement [145](#)
parking enforcement systems [144](#)
safe and secure city [142](#), [143](#)
traffic management [144](#)
smart health [146](#)
boxing moves detector [148](#)
cataract detection [146](#)
cough detection [148](#)
fall detection [147](#)
mosquito detection [149](#)
sleep apnea detection [150](#)
snoring detection [150](#)
smart home
glassbreak detection [151](#), [152](#)
person detection, at door [151](#)
smart baby monitoring [152](#)
voice recognition, for home automation [153](#)
smart industry [153](#)
defect detection [155](#)
railway track defect detection [153](#), [154](#)
telecom towers defect detection [154](#)
softmax formula [11](#)
softmax transformation [228](#)
SparkFun Edge [58-62](#)
spectral analysis [174](#)
state-of-the-art (SOTA) [35](#)
STM32Cube.AI [73](#)
STM32 X-CUBE-AI [85](#)
STMicroelectronics [73](#)
STMicroelectronics STM32L4 [66](#)
 features [67](#)
structured pruning [122](#)
supervised learning [29](#)
Syntiant's purpose built AI chips
 >10x parameters [18](#)
 >20x Throughput [18](#), [19](#)
 >200x Power advantage [18](#)
Syntiant TinyML [59](#), [68](#)
 features [68](#), [69](#)
system level performance
 improving, with phrase recognition constraints [246-248](#)
System on a Chip (SoC) [64](#), [65](#)

T

TensorFlow Lite [37](#), [70](#)
advantages [37](#), [38](#)
for model compression [84](#)
TensorFlow Lite Micro [38](#)
TensorFlow Lite Micro (Google) [70](#)
TensorFlow (TF) [36](#)
Tensor Processing Units (TPU) [53](#)
TF Lite [36](#)
TinyML [14](#)
at scale, Venn diagram [98](#)
inferencing [126](#)
Venn diagram [98](#)
TinyML ecosystem [19](#)
TinyML hardware boards [57](#)
Adafruit Feather [61](#)
Arduino Nano [33](#) BLE [59](#)
benefits [58](#)
Google Coral Edge TPU [63](#)
Intel Curie [67](#)
Jetson Nano [62](#), [63](#)
model deploying [128](#), [129](#)
Nicla Sense ME [60](#)
NXP i.MX 8M [65](#)
practical experiments [161](#)
Qualcomm QCS605 [64](#)
SparkFun Edge [62](#)
STMicroelectronics STM32L4 [66](#)
Syntiant TinyML [68](#), [69](#)
TinyML Model Compression Frameworks [81](#), [82](#)
TinyML Software Suites [69](#), [70](#)
Arduino Create [72](#)
EdgeML (Microsoft) [72](#)
EloquentML [72](#)
EON Compiler (Edge Impulse) [73](#)
Metavision Intelligence Suite 3.0 [78](#)
NanoEdge AI Studio [74](#)
Neuton TinyML [77](#)
OpenMV [76](#)
PYNNQ [74](#), [75](#)
SensiML [76](#), [77](#)
STM32Cube.AI [73](#)
STMicroelectronics [73](#)
TensorFlow Lite Micro (Google) [70](#)
uTensor (ARM) [71](#)
traditional problem-solving approaches [26](#)
transfer learning [35](#), [267](#), [268](#)

U

unsupervised learning [29](#)

uTensor (ARM) [71](#)

V

VineScout [136](#)

W

weight pruning [122](#)

Z

Zephyr Project [70](#)

zero-shot learning [263](#)