O'REILLY®

AI at the Edge

Solving Real-World Problems with Embedded Machine Learning

Free Chapters

compliments of

EDGE IMPULSE

Daniel Situnayake
& Jenny Plunkett
Foreword by Pete Warden

EDGE IMPULSE

IMAGINE

The Edge ML Conference

September 27, 2023

https://edgeimpulse.com/imagine

AI at the Edge

Solving Real-World Problems with Embedded Machine Learning

Daniel Situnayake and Jenny Plunkett

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

AI at the Edge
by Daniel Situnayake and Jenny Plunkett

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Nicole Butterfield
Development Editor: Angela Rufino
Production Editor: Elizabeth Faerm
Copyeditor: nSight, Inc.
Proofreader: Charles Roumeliotis
Indexer: WordCo Indexing Services, Inc.
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

January 2023: First Edition

Revision History for the First Edition
2023-01-10: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098120207* for release details.

978-1-098-12020-7

[LSI]

Table of Contents

Foreword

It's the start of 2023, and it suddenly feels like *everyone* is talking about artificial intelligence. A convergence of new and existing technologies has created novel capabilities, and business and technical leaders are applying them to build mind-blowing products and experiences. The public is taking note.

While much of the attention is on generative AI, with its gigantic models that output natural-sounding text and images, there's an even larger revolution happening at a far smaller scale. Advances in software and hardware mean that embedded systems—the tiny, hidden computers that run our world—have become capable of hosting powerful AI algorithms that can read and interpret sensor data. With 28 billion microcontrollers produced *every year*, the opportunity is tremendous.

What started in the labs of companies like Google, where I first worked on edge AI, is now bringing new capabilities to every corner of our world. From the phone in your pocket to the factory floor, AI and machine learning are making it possible for devices to recognize voice commands, identify dangerous situations, monitor health conditions, and protect endangered species. The most innovative companies are using edge AI to reduce costs, improve customer experiences, and build products that were never possible before. The rest are still catching up.

Despite this potential, the unfortunate fact is that most AI projects fail. Edge AI is at the intersection of two challenging disciplines: AI and machine learning, and embedded engineering. Very few orgs have the skills to be productive with both. Much of the success of AI projects depends on building the right team, and giving them access to the right tools and skills for the job.

At Edge Impulse, where I lead the machine learning team, we build tools that help organizations be successful with edge AI. We enable a team's ML experts to make sense of sensor data and to adapt their models to run on embedded systems, and we empower the experienced embedded engineers to develop their own algorithms, no prior knowledge required.

I started at Edge Impulse as the first full-time employee, three years ago. We've since grown to a large, multidisciplinary team, including experts from across many industries. This book, like the tools we've built, represents the knowledge we've collectively uncovered about how to build successful projects in the edge AI space. It draws insight from the thousands of real world projects that companies have built with our product.

Our goal as a company is to enable your existing developers to succeed with edge AI—whether they're ML practitioners or embedded software engineers—without having to hire or train a whole new team. We use our deep organizational knowledge to help you navigate the pitfalls and challenges of developing an AI product, lower your time to market, and help your engineers feel immediately productive. As these chapters show, our tools can both lower the costs of development and reduce the risks associated with delivering AI projects.

As a truly transformative technology, edge AI represents vast potential for impact. Some of those who read this book will go on to create products that transform their respective industries, create huge economic value, and deliver improvements in quality of life to millions of people. We hope that with our help, it's your team that does it.

— Daniel Situnayake
Head of ML at Edge Impulse,
coauthor of AI at the Edge *and* TinyML

CHAPTER 1

A Brief Introduction to Edge AI

Welcome on board! In this chapter, we'll be taking a comprehensive tour of the edge AI world. We'll define the key terms, learn what makes "edge AI" different from other AI, and explore some of the most important use cases. Our goal for this chapter is to answer these two important questions:

- What is edge AI, anyway?
- Why would I ever need it?

Defining Key Terms

Each area of technology has its own taxonomy of buzzwords, and edge AI is no different. In fact, the term *edge AI* is a union of two buzzwords, fused together into one mighty term. It's often heard alongside its siblings, *embedded machine learning* and *TinyML*.

Before we move on, we better spend some time defining these terms and understanding what they mean. Since we're dealing with compound buzzwords, let's deal with the most fundamental parts first.

Embedded

What is "embedded"? Depending on your background, this may be the most familiar of all the terms we're trying to describe. *Embedded systems* are the computers that control the electronics of all sorts of physical devices, from Bluetooth headphones to the engine control unit of a modern car. *Embedded software* is software that runs on them. Figure 1-1 shows a few places where embedded systems can be found.

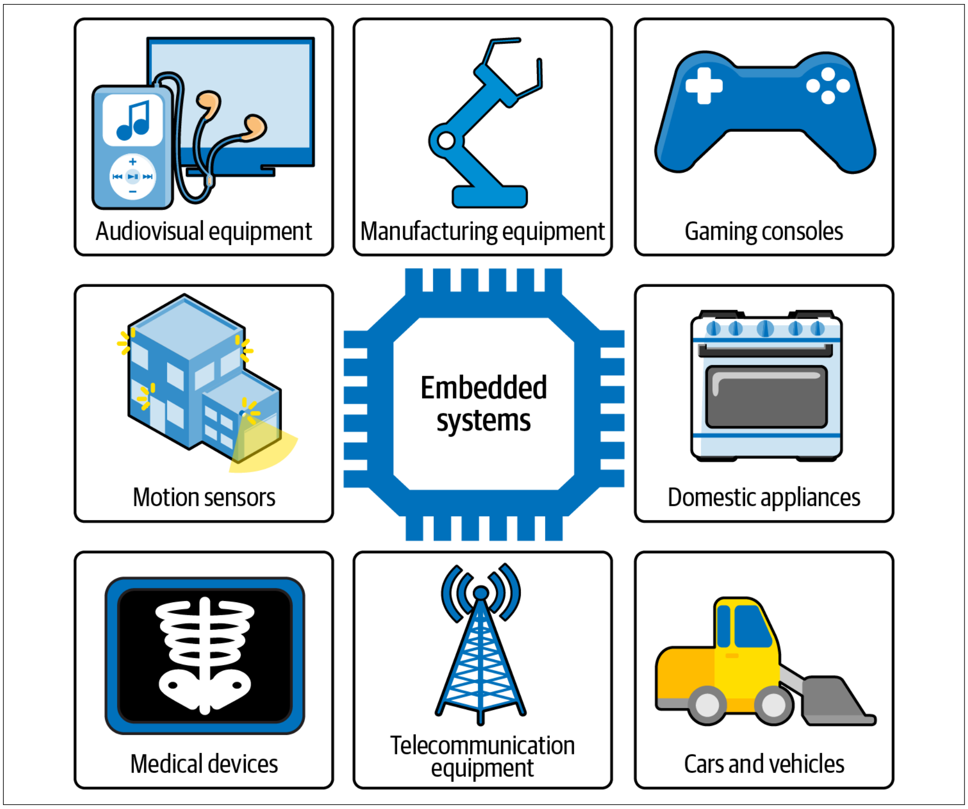


Figure 1-1. Embedded systems are present in every part of our world, including the home and the workplace

Embedded systems can be tiny and simple, like the microcontroller that controls a digital watch, or large and sophisticated, like the embedded Linux computer inside a smart TV. In contrast to general-purpose computers, like a laptop or smartphone, embedded systems are usually meant to perform one specific, dedicated task.

Since they power much of our modern technology, embedded systems are extraordinarily widespread. In fact, there were over 28 billion microcontrollers shipped in the year 2020[1]—just one type of embedded processor. They're in our homes, our vehicles, our factories, and our city streets. It's likely you are never more than a few feet from an embedded system.

1 As reported by Business Wire.

It's common for embedded systems to reflect the constraints of the environments into which they are deployed. For example, many embedded systems are required to run on battery power, so they're designed with energy efficiency in mind—perhaps with limited memory or an extremely slow clock rate.

Programming embedded systems is the art of navigating these constraints, writing software that performs the task required while making the most out of limited resources. This can be incredibly difficult. Embedded systems engineers are the unsung heroes of the modern world. If you happen to be one, thank you for your hard work!

The Edge (and the Internet of Things)

The history of computer networks has been a gigantic tug of war. In the first systems—individual computers the size of a room—computation was inherently centralized. There was one machine, and that one machine did all the work.

Eventually, however, computers were connected to terminals (as shown in Figure 1-2) that took over some of their responsibilities. Most of the computation was happening in the central mainframe, but some simple tasks—like figuring out how to render letters onto a cathode-ray tube screen—were done by the terminal's electronics.

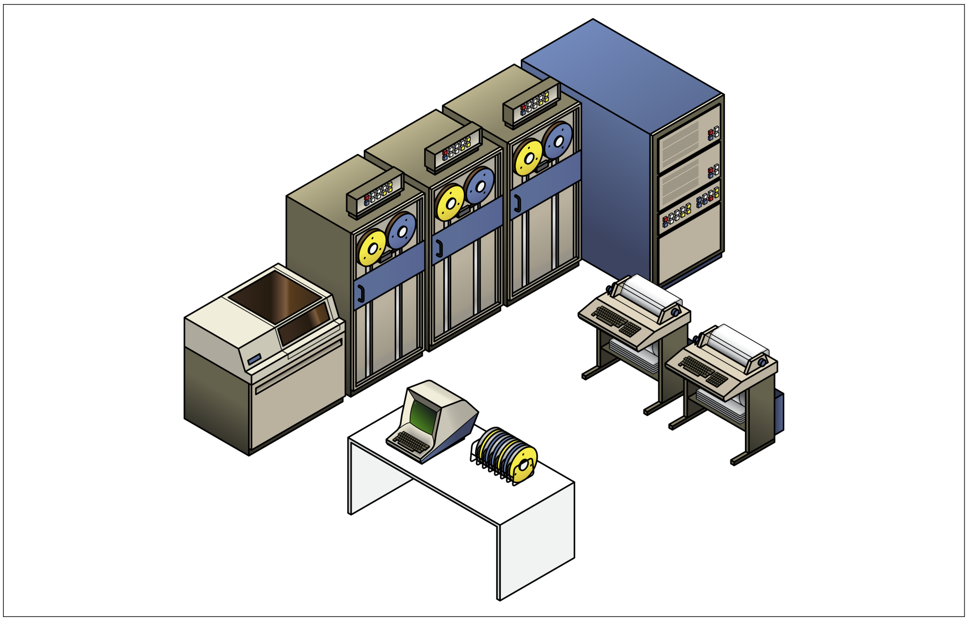


Figure 1-2. Mainframe computers performed the bulk of the computation, while simple terminals processed input, printed output, and rendered basic graphics

Over time, terminals became more and more sophisticated, taking over more and more functions that were previously the job of the central computer. The tug-of-war had begun! Once the personal computer was invented, small computers could do useful work without even being connected to another machine. The rope had been pulled to the opposite extreme—from the center of the network to the *edge*.

The growth of the internet, along with web applications and services, made it possible to do some really cool stuff—from streaming video to social networking. All of this depends on computers being connected to servers, which have gradually taken over more and more of the work. Over the past decade, most of our computing has become centralized again—this time in the "cloud." When the internet goes down, our modern computers aren't much use!

But the computers we use for work and play are not our only connected devices. In fact, it is estimated that in 2021 there were 12.2 billion assorted items connected to the internet,[2] creating and consuming data. This vast network of objects is called the Internet of Things (IoT), and it includes everything you can think of: industrial sensors, smart refrigerators, internet-connected security cameras, personal automobiles, shipping containers, fitness trackers, and coffee machines.

The first ever IoT device was created in 1982. Students at Carnegie Mellon University connected a Coke vending machine to the ARPANET—an early precursor to the internet—so they could check whether it was empty without leaving their lab.

All of these devices are embedded systems containing microprocessors that run software written by embedded software engineers. Since they're at the edge of the network, we can also call them *edge devices*. Performing computation on edge devices is known as *edge computing*.

The edge isn't a single place; it's more like a broad region. Devices at the edge of the network can communicate with each other, and they can communicate with remote servers, too. There are even servers that live at the edge of the network. Figure 1-3 shows how this looks.

2 Expected to grow to 27 billion by 2025, according to IoT Analytics.

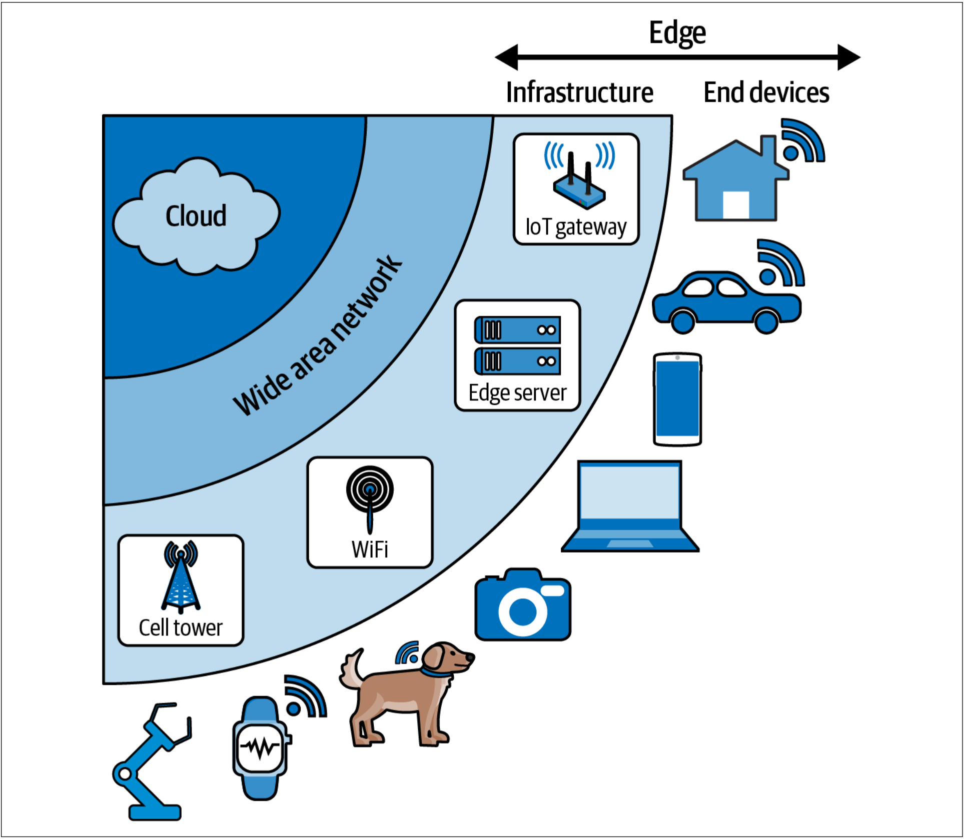


Figure 1-3. Devices at the edge of the network can communicate with the cloud, with edge infrastructure, and with each other; edge applications generally span multiple locations within this map (for example, data might be sent from a sensor-equipped IoT device to a local edge server for processing)

There are some major benefits to being at the edge of the network. For one, it's where all the data comes from! Edge devices are our link between the internet and the physical world. They can use sensors to collect data based on what is going on around them, be that the heart rate of a runner or the temperature of a cold drink. They can make decisions on that data locally and send it to other locations. Edge devices have access to data that nobody else does.

Are Mobile Phones and Tablets Edge Devices?

As portable computers that live at the edge of the network, mobile phones, tablets, and even personal computers are all edge devices. Mobile phones were one of the first platforms to feature edge AI: modern mobile phones use it for many purposes, from voice activation to smart photography.[3]

We'll come back to edge devices later (since they're the focus of this book). Until then, let's continue to define some terms.

Artificial Intelligence

Phew! This is a big one. Artificial intelligence (AI) is a very big idea, and it's terribly hard to define. Since the dawn of time, humans have dreamed of creating intelligent entities that can help us in our struggle to survive. In the modern world we dream of robot sidekicks who assist with our adventures: hyperintelligent, synthetic minds that will solve all of our problems, and miraculous enterprise products that will optimize our business processes and guarantee us rapid promotion.

But to define AI, we have to define intelligence—which turns out to be particularly tough. What does it mean to be intelligent? Does it mean that we can talk, or think? Clearly not—just ask the slime mold (see Figure 1-4), a simple organism with no central nervous system that is capable of solving a maze.

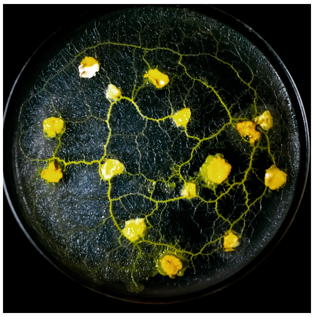


Figure 1-4. Slime molds are single-celled organisms that have been documented as being able to solve mazes in order to locate food, via a process of biological computation—as shown in "Slime Mould Solves Maze in One Pass Assisted by Gradient of Chemo-Attractants" (Andrew Adamatzky, arXiv, 2011)

3 Embedded engineering and mobile development are typically separate disciplines. Even within a mobile device, the embedded firmware and operating system are distinct from mobile applications. This book focuses on embedded engineering, so we won't talk much about building mobile apps—but we will cover techniques that are relevant in both cases.

Since this isn't a philosophy book, we don't have the time to fully explore the topic of intelligence. Instead, we want to suggest a quick-and-dirty definition:

> Intelligence means knowing the right thing to do at the right time.

This probably doesn't stand up to academic debate, but that's fine with us. It gives us a tool to explore the subject. Here are some tasks that require intelligence, according to our definition:

- Taking a photo when an animal is in the frame
- Applying the brakes when a driver is about to crash
- Informing an operator when a machine sounds broken
- Answering a question with relevant information
- Creating an accompaniment to a musical performance
- Turning on a faucet when someone wants to wash their hands

Each of these problems involves both an action (turning on a faucet) and a precondition (when someone wants to wash their hands). Within their own context, most of these problems sound relatively simple—but, as anyone who has used an airport restroom knows, they are not always straightforward to solve.

It's pretty easy for most humans to perform most of these tasks. We're highly capable creatures with *general* intelligence. But it's possible for smaller systems with more *narrow* intelligence to perform the tasks, too. Take our slime mold—it may not understand why it is solving a maze, but it's certainly able to do it.

That said, the slime mold is unlikely to also know the right moment to turn on a faucet. Generally speaking, it's a lot easier to perform a single, tightly scoped task (like turning on a faucet) than to be able to perform a diverse set of entirely different tasks.

Creating an artificial *general* intelligence, equivalent to a human being, would be super difficult—as decades of unsuccessful attempts have shown. But creating something that operates at slime mold level can be much easier. For example, preventing a driver from crashing is, in theory, quite a simple task. If you have access to both their current speed and their distance from a wall, you can do it with simple conditional logic:

```
current_speed = 10 # In meters per second
distance_from_wall = 50 # In meters
seconds_to_stop = 3 # The minimum time in seconds required to stop the car
safety_buffer = 1 # The safety margin in seconds before hitting the brakes

# Calculate how long we've got before we hit the wall
seconds_until_crash = distance_from_wall / current_speed

# Make sure we apply the brakes if we're likely to crash soon
```

```
if seconds_until_crash < seconds_to_stop + safety_buffer:
  applyBrakes()
```

Clearly, this simplified example doesn't account for a lot of factors. But with a little more complexity, a modern car with a driver assistance system based on this conditional logic could arguably be marketed as AI.[4]

There are two points we are trying to make here: the first is that intelligence is quite hard to define, and many rather simple problems require a degree of intelligence to solve. The second is that the programs that implement this intelligence do not necessarily need to be particularly complex. Sometimes, a slime mold will do.

So, what is AI? In simple terms, it's an artificial system that makes intelligent decisions based on some kind of input. And one way to create AI is with machine learning.

Machine Learning

At its heart, machine learning (ML) is a pretty simple concept. It's a way to discover patterns in how the world works—but automatically, by running data through algorithms.

We often hear AI and machine learning used interchangeably, as if they are the same thing—but this isn't the case. AI doesn't always involve machine learning, and machine learning doesn't always involve AI. That said, they pair together very nicely!

The best way to introduce machine learning is through an example. Imagine you're building a fitness tracker—it's a little wristband that an athlete can wear. It contains an accelerometer, which tells you how much acceleration is happening on each axis (x, y, and z) at a given moment in time—as shown in Figure 1-5.

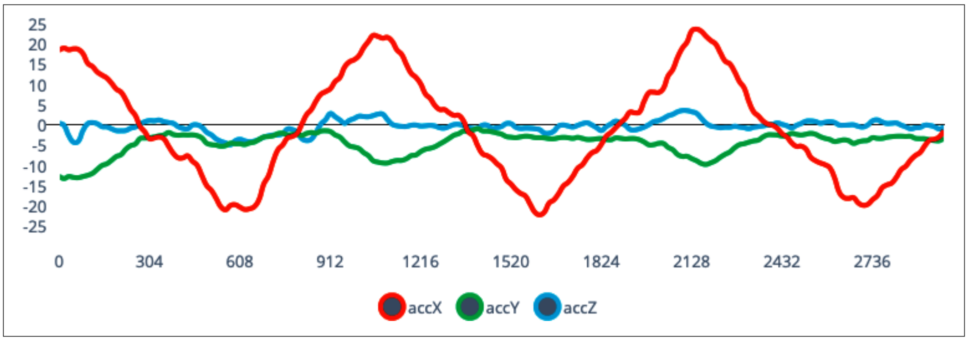


Figure 1-5. The output of a three-axis accelerometer sampled at 6.25 Hz

4 For many years it was hoped that artificial general intelligence could be achieved by complex conditional logic, hand-tuned by engineers. It has turned out to be a lot more complicated than that!

To help your athletes, you want to keep an automatic log of the activities they are doing. For example, an athlete might spend an hour running on Monday and then an hour swimming on Tuesday.

Since our movements while swimming are quite different from our movements while running, you theorize that you might be able to tell these activities apart based on the output of the accelerometer in your wristband. To collect some data, you give prototype wristbands to a dozen athletes and have them perform specific activities—either swimming, running, or doing nothing—while the wristbands log data (see Figure 1-6).

Now that you have a dataset, you want to try to determine some rules that will help you understand whether a particular athlete is swimming, running, or just chilling out. One way to do this is by hand: analyzing and inspecting the data to see if anything stands out to you. Perhaps you notice that running involves more rapid acceleration on a particular axis than swimming. You can use this information to write some conditional logic that determines the activity based on the reading from that axis.

Analyzing data by hand can be tricky, and it generally requires expert knowledge about the domain (such as human movements during sport). An alternative to manual analysis might be to use machine learning.

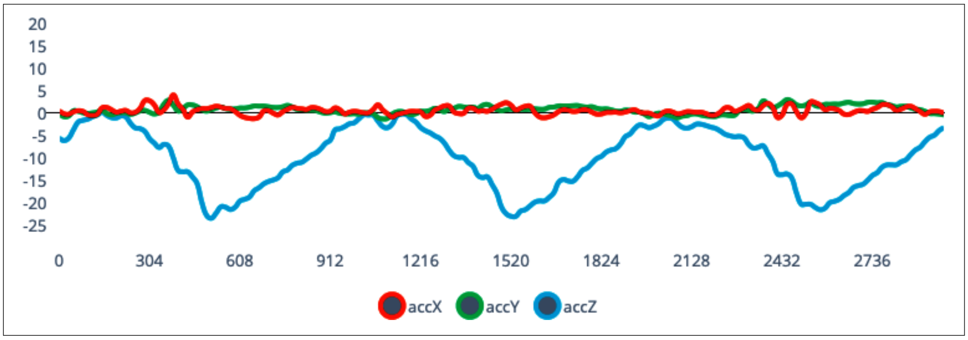


Figure 1-6. The output of a three-axis accelerometer showing a different activity than in Figure 1-5; each activity can be characterized by a pattern of changes in acceleration on each axis over time

With an ML approach, you feed all of your athletes' data into a training algorithm. When provided with both the accelerometer data and information about which activity the athlete is currently performing, the algorithm does its best to learn a mapping between the two. This mapping is called a *model*.

Hopefully, if the training was successful, your new machine learning model can take a brand new, never-seen-before input—a sample of accelerometer data from a particular window in time—and tell you which activity an athlete is performing. This process is known as *inference*.

This ability to understand *new* inputs is called *generalization*. During training, the model has learned the characteristics that distinguish running from swimming. You can then use the model in your fitness tracker to understand fresh data, in the same way that you might use the conditional logic we mentioned earlier.

There are lots of different machine learning algorithms, each with their own strengths and drawbacks—and ML isn't always the best tool for the job. Later in this chapter we'll discuss the scenarios where machine learning is the most helpful. But a nice rule of thumb is that machine learning really shines when our data is really complex.

Edge AI

Congratulations, we've made it to our first compound buzzword! Edge AI is, unsurprisingly, the combination of edge devices and artificial intelligence.

As we discussed earlier, edge devices are the embedded systems that provide the link between our digital and physical worlds. They typically feature sensors that feed them information about the environment they are close to. This gives them access to a metaphorical fire hose of high-frequency data.

We're often told that data is the lifeblood of our modern economy, flowing throughout our infrastructure and enabling organizations to function. That's definitely true—but all data is not created equally. The data obtained from sensors tends to have a very high volume but a relatively low informational content.

Imagine the accelerometer-based wristband sensor we described in the previous section. The accelerometer is capable of taking a reading many hundreds of times per second. Each individual reading tells us very little about the activity currently taking place—it's only in aggregate, over thousands of readings, that we can begin to understand what is going on.

Typically, IoT devices have been viewed as simple nodes that collect data from sensors and then transmit it to a central location for processing. The problem with this approach is that sending such large volumes of low-value information is extraordinarily costly. Not only is connectivity expensive, but transmitting data uses a ton of energy—which is a big problem for battery-powered IoT devices.

Because of this problem, the vast majority of data collected by IoT sensors has usually been discarded. We're collecting a ton of sensor data, but we're unable to do anything with it.

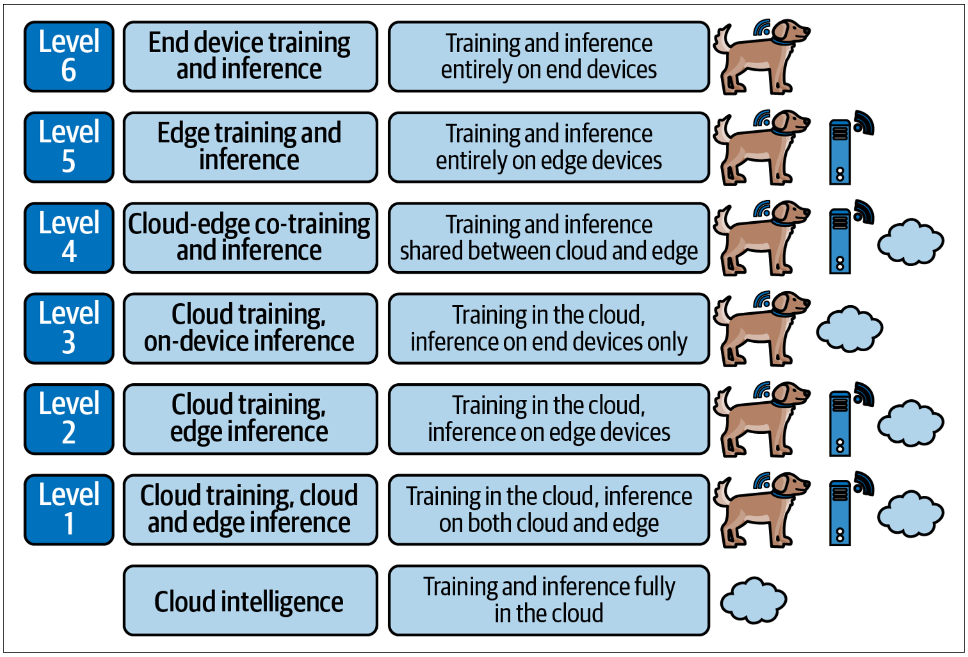
Edge AI is the solution to this problem. Instead of having to send data off to some distant location for processing, what if we do it directly on-device, where the data is being generated? Now, instead of relying on a central server, we can make decisions locally—no connectivity required.

And if we still want to report information back to upstream servers, or the cloud, we can transmit just the important information instead of having to send every single sensor reading. That should save a lot of cost and energy.

There are many different ways to deploy intelligence to the edge. Figure 1-7 shows the continuum from cloud AI to fully on-device intelligence. As we'll see later in this book, edge AI can be spread across entire distributed computing architectures—including some nodes at the very edge, and others in local gateways or the cloud.

As we've seen, artificial intelligence can mean many different things. It can be super simple: a touch of human insight encoded in a little simple conditional logic. It can also be super sophisticated, based on the latest developments in deep learning.

Edge AI is exactly the same. At its most basic, edge AI is about making some decisions on the edge of the network, close to where the data is made. But it can also take advantage of some really cool stuff. And that brings us nicely to the next section!



Level 6
End device training and inference
Training and inference entirely on end devices
Level 5
Edge training and inference
Training and inference entirely on edge devices
Level 4
Cloud-edge co-training and inference
Training and inference shared between cloud and edge
Level 3
Cloud training, on-device inference
Training in the cloud, inference on end devices only
Level 2
Cloud training, edge inference
Training in the cloud, inference on edge devices
Level 1
Cloud training, cloud and edge inference
Training in the cloud, inference on both cloud and edge
Cloud intelligence
Training and inference fully in the cloud

Figure 1-7. The continuum between cloud intelligence and fully on-device intelligence; these six levels were described by "Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing" (Zhou et. al., Proceedings of the IEEE, 2019)

Embedded Machine Learning and Tiny Machine Learning

Embedded ML is the art and science of running machine learning models on embedded systems. Tiny machine learning, or TinyML,[5] is the concept of doing this on the most constrained embedded hardware available—think microcontrollers, digital signal processors, and small field programmable gate arrays (FPGAs).

When we talk about embedded ML, we're usually referring to machine learning inference—the process of taking an input and coming up with a prediction (like guessing a physical activity based on accelerometer data). The training part usually still takes place on a conventional computer.

Embedded systems often have limited memory. This raises a challenge for running many types of machine learning models, which often have high requirements for both read-only memory (ROM) (to store the model) and RAM (to handle the intermediate results generated during inference).

They are often also limited in terms of computation power. Since many types of machine learning models are quite computationally intensive, this can also raise problems.

Luckily, over the past few years there have been many advances in optimization that have made it possible to run quite large and sophisticated machine learning models on some very small, low-power embedded systems. We'll learn about some of those techniques over the next few chapters!

Embedded machine learning is often deployed alongside its trusty companion, *digital signal processing*. Before we move on, let's define that term, too.

Digital Signal Processing

In the embedded world we often work with the digital representations of signals. For example, an accelerometer gives us a stream of digital values that correspond to acceleration on three axes, and a digital microphone gives us a stream of values that correspond to sound levels at a particular moment in time.

Digital signal processing (DSP) is the practice of using algorithms to manipulate these streams of data. When paired with embedded machine learning, we often use DSP to modify signals before feeding them into machine learning models. There are a few reasons why we might want to do this:

- Cleaning up a noisy signal
- Removing spikes or outlying values that might be caused by hardware issues

5 The term *TinyML* is a registered trademark of the TinyML Foundation.

- Extracting the most important information from a signal
- Transforming the data from the time domain to the frequency domain[6]

DSP is so common for embedded systems that often embedded chips have super fast hardware implementations of common DSP algorithms, just in case you need them.

We now share a solid understanding of the most important terms in this book. Figure 1-8 shows how they fit together in context.

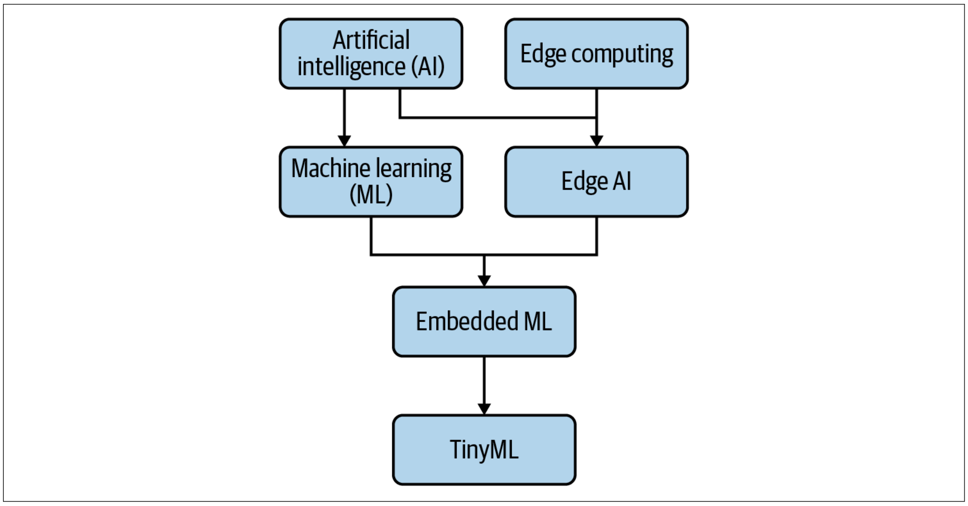


Figure 1-8. This diagram shows some of the most important concepts in edge AI in context with each other—from most general at the top to most specific at the bottom

In the next section, we'll dive deep into the topic of edge AI and start to break down what makes it such an important technology.

Why Do We Need Edge AI?

Imagine that this morning you went on a trail run through Joshua Tree National Park, a vast expanse of wilderness in the Southern California desert. You listened to music the whole time, streamed to your phone via an uninterrupted cellular connection. At a particularly beautiful spot, deep in the mountains, you snapped a photograph and sent it to your partner. A few minutes later you received their reply.

In a world where even the most remote places have some form of data connection, why do we need edge AI? What is the point of tiny devices that can make their own

6 This will be explained in Chapter 4.

decisions if the internet's beefy servers are only a radio burst away? With all of the added complication, aren't we just making life more difficult for ourselves?

As you may have guessed, the answer is no! Edge AI solves some very real problems that otherwise stand in the way of making our technology work better for human beings. Our favorite framework for explaining the benefits of edge AI is a rude-sounding mnemonic: BLERP.

To Understand the Benefits of Edge AI, Just BLERP

BLERP? Jeff Bier, founder of the Edge AI and Vision Alliance, created this excellent tool for expressing the benefits of edge AI. It consists of five words:

- Bandwidth
- Latency
- Economics
- Reliability
- Privacy

Armed with BLERP, anyone can easily remember and explain the benefits of edge AI. It's also useful as a filter to help decide whether edge AI is well suited for a particular application.

Let's go through it, word by word.

Bandwidth

IoT devices often capture more data than they have bandwidth to transmit. This means the vast majority of sensor data they capture is not even used—it's just thrown away! Imagine a smart sensor that monitors the vibration of an industrial machine to determine if it is operating correctly. It might use a simple thresholding algorithm to understand when the machine is vibrating too much, or not enough, and then communicate this information via a low bandwidth radio connection.

This already sounds useful. But what if you could identify patterns in the data that give you a clue that the machine might be about to fail? If we had a lot of bandwidth, we could send the sensor data up to the cloud and do some kind of analysis to understand whether a failure is imminent.

In many cases, though, there isn't enough bandwidth or energy budget available to send a constant stream of data to the cloud. That means that we'll be forced to discard most of our sensor data, even though it contains useful signals.

Bandwidth limitations are very common. It's not just about available connectivity—it's also about power. Networked communication is often the most energy-intensive task an embedded system can perform, meaning that battery life is often the limiting function. Some machine learning models can be quite compute intensive, but they tend to still use less energy than transmitting a signal.

This is where edge AI comes in. What if we could run the data analysis on the IoT device itself, without having to upload the data? In that case, if the analysis showed that the machine was about to fail, we could send a notification using our limited bandwidth. This is much more feasible than trying to stream all of the data.

Of course, it's also quite common for devices to have no network connection at all! In this case, edge AI enables a whole galaxy of use cases that were previously impossible. We'll hear more about this later.

Latency

Transmitting data takes time. Even if you have a lot of available bandwidth it can take tens or hundreds of milliseconds for a round-trip from a device to an internet server. In some cases, latency can be measured in minutes, hours, or days—think satellite communications, or store-and-forward messaging.

Some applications demand a faster response. For example, it might be impractical for a moving vehicle to be controlled by a remote server. Controlling a vehicle as it navigates an environment requires constant feedback between steering adjustments and the vehicle's position. Under significant latency, steering becomes a major challenge!

Edge AI solves this problem by removing the round-trip time altogether. A great example of this is a self-driving car. The car's AI systems run on onboard computers. This allows it to react nearly instantly to changing conditions, like the driver in front slamming on their brakes.

One of the most compelling examples of edge AI as a weapon against latency is in robotic space exploration. Mars is so distant from Earth that it takes *minutes* for a radio transmission to reach it at the speed of light. Even worse, direct communication is often impossible due to the arrangement of the planets. This makes controlling a Mars rover very hard. NASA solved this problem by using edge AI—their rovers use sophisticated artificial intelligence systems to plan their tasks, navigate their environments, and search for life on the surface of another world. If you have some spare time, you can even help future Mars rovers navigate by labeling data to improve their algorithms!

Economics

Connectivity costs a lot of money. Connected products are more expensive to use, and the infrastructure they rely on costs their manufacturers money. The more bandwidth required, the steeper the cost. Things get especially bad for devices deployed on remote locations that require long-range connectivity via satellite.

By processing data on-device, edge AI systems reduce or avoid the costs of transmitting data over a network and processing it in the cloud. This can unlock a lot of use cases that would have previously been out of reach.

In some cases, the only "connectivity" that works is sending out a human being to perform some manual task. For example, it's common for conservation researchers to use camera traps to monitor wildlife in remote locations. These devices take photos when they detect motion and store them to an SD card. It's too expensive to upload every photo via satellite internet, so researchers have to travel out to their camera traps to collect the images and clear the storage.

Because traditional camera traps are motion activated, they take a lot of unnecessary photos—they might be triggered by branches moving in the wind, hikers walking past, and creatures the researchers aren't interested in. But some teams are now using edge AI to identify only the animals they care about, so they can discard the other images. This means they don't have to fly out to the middle of nowhere to change an SD card *quite* so often.

In other cases, the cost of connectivity might not be a concern. However, for products that depend on server-side AI, the cost of maintaining server-side infrastructure can complicate your business model. If you have to support a fleet of devices that need to "phone home" to make decisions, you may be forced into a subscription model. You'll also have to commit to maintaining servers for a long period of time—at the risk of your customers finding themselves with "bricked" devices if you decide to pull the plug.[7]

Don't underestimate the impact of economics. By reducing the cost of long-term support, edge AI enables a vast number of use cases that would otherwise be infeasible.

Reliability

Systems controlled by on-device AI are potentially more reliable than those that depend on a connection to the cloud. When you add wireless connectivity to a device, you're adding a vast, overwhelmingly complex web of dependencies, from link-layer communications technologies to the internet servers that may run your application.

Many pieces of this puzzle are outside of your control, so even if you make all the right decisions you will still be exposed to the reliability risk associated with technologies that make up your distributed computing stack.

7 Not all edge AI applications are immune to this since it's often necessary to monitor devices and push updates to algorithms. That said, there are certainly many cases where edge AI can reduce the burden of maintenance.

For some applications, this might be tolerable. If you're building a smart speaker that responds to voice commands, your users might understand if it stops recognizing their commands when their home internet connection goes down. That said, it can still be a frustrating experience!

But in other cases, safety is paramount. Imagine an AI-based system that monitors an industrial machine to make sure that it is being operated within safe parameters. If it stops working when the internet goes down, it could endanger human lives. It would be much safer if the AI is based entirely on-device, so it still operates in the event of a connectivity problem.

Reliability is often a compromise, and the required level of reliability varies depending on use case. Edge AI can be a powerful tool in improving the reliability of your products. While AI is inherently complex, it represents a different type of complexity than global connectivity, and its risk is easier to manage in many situations.

Privacy

Over the past few years, many people have begrudgingly resigned themselves to a trade-off between convenience and privacy. The theory is that if we want our technology products to be smarter and more helpful, we have to give up our data. Because smart products traditionally make decisions on remote servers, they very often end up sending streams of sensor data to the cloud.

This may be fine for some applications—for example, we might not worry that an IoT thermostat is reporting temperature data to a remote server.[8] But for other applications, privacy is a huge concern. For example, many people would hesitate to install an internet-connected security camera inside their home. It might provide some reassuring security, but the trade-off—that a live video and audio feed of their most private spaces is being broadcast to the internet—does not seem worth it. Even if the camera's manufacturer is entirely trustworthy, there's always a chance of the data being exposed through security vulnerabilities.[9]

Edge AI provides an alternative. Rather than streaming live video and audio to a remote server, a security camera could use some onboard intelligence to identify that an intruder is present when the owners are out at work. It could then alert the owners in an appropriate way. When data is processed on an embedded system and is never transmitted to the cloud, user privacy is protected and there is less chance of abuse.

8 Even in this innocuous example, a malicious person accessing your thermostat data could use it to recognize when you're on vacation so they can break into your house.

9 This exact scenario unfolded in 2022 with the Ring home security system, which was found to be vulnerable to an attack ("Amazon's Ring Quietly Fixed Security Flaw That Put Users' Camera Recordings at Risk of Exposure", TechCrunch, 2022).

The ability of edge AI to enable privacy unlocks a huge number of exciting use cases. It's an especially important factor for applications in security, industry, childcare, education, and healthcare. In fact, since some of these fields involve tight regulations (or customer expectations) around data security, the product with the best privacy is one that *avoids* collecting data altogether.

Using BLERP

As we'll start to see in Chapter 2, BLERP can be a handy tool for understanding whether a particular problem is well suited for edge AI. There doesn't have to be a strong argument for every word of the acronym: even meeting just one or two criteria, if compelling enough, can give merit to a use case.

Edge AI for Good

The unique benefits of edge AI provide a new set of tools that can be applied to some of our world's biggest problems. Technologists in areas like conservation, healthcare, and education are already using edge AI to make a big impact. Here are just a few examples we're personally excited about:

- Smart Parks is using collars running machine learning models to better understand elephant behavior in wildlife parks around the world.
- Izoelektro's RAM-1 helps prevent forest fires caused by power transmission hardware by using embedded machine learning to detect upcoming faults.
- Researchers like Dr. Mohammed Zubair Shamim from King Khalid University in Saudi Arabia are training models that can screen patients for life-threatening medical conditions such as oral cancer using low-cost devices.
- Students across the world are developing solutions for their local industries. João Vitor Yukio Bordin Yamashita, from UNIFEI in Brazil, created a system for identifying diseases that affect coffee plants using embedded hardware.

The properties of edge AI make it especially well-suited for application to global problems. Since reliable connectivity is expensive and not universally available, many current generation smart technologies only benefit people living in industrialized, wealthy, and well-connected regions. By removing the need for a reliable internet connection, edge AI increases access to technologies that can benefit people and the planet.

When machine learning is part of the mix, edge AI generally involves small models—which are often quick and cheap to train. Since there's also no need to maintain expensive backend server infrastructure, edge AI makes it possible for developers with limited resources to build cutting-edge solutions for the local markets that they know better than anyone. To learn more about these opportunities, we recommend watching "TinyML and the Developing World", an excellent talk given by Pete Warden at the TinyML Kenya meetup.

As we saw in "Privacy" on page 17, edge AI also creates an opportunity to improve privacy for users. In our networked world, many companies treat user data as a valuable resource to be extracted and mined. Consumers and business owners are often required to barter away their privacy in order to use AI products, putting their data in the hands of unknown third parties.

With edge AI, data does not need to leave the device. This enables a more trusting relationship between user and product, giving users ownership of their own data. This is especially important for products designed to serve vulnerable people, who may feel skeptical of services that seem to be harvesting their data.

TinyML for Developing Countries

If you're interested in the global benefits of edge AI, the TinyML for Developing Countries (TinyML4D) initiative is building a network of researchers and practitioners who are focused on solving developing world challenges using edge AI.

As we'll see in later sections, there are many potential pitfalls that must be navigated in order to build ethical AI systems. That said, the technology provides a tremendous opportunity to make the world a better place.

If you're thinking about using edge AI to solve problems for your local community, the authors would love to hear from you. We've provided support for a number of impactful projects and would love to identify more. Send an email to the authors at *hello@edgeaibook.com*.

Key Differences Between Edge AI and Regular AI

Edge AI is a subset of regular AI, so a lot of the same principles apply. That said, there are some special things to consider when thinking about artificial intelligence on edge devices. Here are our top points.

Training on the edge is rare

A lot of AI applications are powered by machine learning. Most of the time, machine learning involves *training* a model to make predictions based on a set of labeled data. Once the model has been trained, it can be used for *inference*: making new predictions on data it has not seen before.

When we talk about edge AI and machine learning, we are usually talking about *inference*. Training models requires a lot more computation and memory than inference does, and it often requires a labeled dataset. All of these things are hard to come by on the edge, where devices are resource-constrained and data is raw and unfiltered.

For this reason, the models used in edge AI are often trained before they are deployed to devices, using relatively powerful compute and datasets that have been cleaned and labeled—often by hand. It's technically possible to train machine learning models on the edge devices themselves, but it's quite rare—mostly due to the lack of labeled data, which is required for training and evaluation.

There are two subtypes of on-device training that are more widespread. One of these is used commonly in tasks such as facial or fingerprint verification on mobile phones, to map a set of biometrics to a particular user. The second is used in predictive maintenance, where an on-device algorithm learns a machine's "normal" state so that it can act if the state becomes abnormal. There'll be more detail on the topic of on-device learning in Chapter 4.

The focus of edge AI is on sensor data

The exciting thing about edge devices is that they live close to where the data is made. Often, edge devices are equipped with sensors that give them an immediate connection to their environments. The goal of an edge AI deployment is to make sense of this data, identifying patterns and using them to make decisions.

By its nature, sensor data tends to be big, noisy, and difficult to manage. It arrives at a high frequency—potentially many thousands of times per second. An embedded device running an edge AI application has a limited time frame in which to collect this data, process it, feed it into some kind of AI algorithm, and act on the results. This is a major challenge, especially given that most embedded devices are resource-constrained and don't have the RAM to store large amounts of data.

The need to tame raw sensor data makes digital signal processing a critical part of most edge AI deployments. In any efficient and effective implementation, the signal processing and AI components must be designed together as a single system, balancing trade-offs between performance and accuracy.

A lot of traditional machine learning and data science tools are focused on tabular data—things like company financials or consumer product reviews. In contrast, edge AI tools are built to handle constant streams of sensor data. This means that a whole different set of skills and techniques is required for building edge AI applications.

ML models can get very small

Edge devices are often designed to limit cost and power consumption. This means that, generally, they have much slower processors and smaller amounts of memory than personal computers or web servers.

The constraints of the target devices mean that, when machine learning is used to implement edge AI, the machine learning models must be quite small. On a midrange microcontroller, there may only be a hundred kilobytes or so of ROM available to store a model, and some devices have far smaller amounts. Since larger models take more time to execute, the slow processors of devices can also push developers toward deploying smaller models.

Making models smaller involves some trade-offs. To begin with, larger models have more capacity to learn. When you make a model smaller, it starts to lose some of its ability to represent its training dataset and may not be as accurate. Because of this, developers creating embedded machine learning applications have to balance the size of their model against the accuracy they require.

Various technologies exist for compressing models, reducing their size so that they fit on smaller hardware and take less time to compute. These compression technologies can be very useful, but they also impact models' accuracy—sometimes in subtle but risky ways. Chapter 4 will talk about these techniques in detail.

That said, not all applications require big, complex models. The ones that do tend to be around things like image processing, since interpreting visual information involves a lot of nuance. Often, for simpler data, a few kilobytes (or less) of model is all you need.

Learning from feedback is limited

As we'll see later, AI applications are built through a series of iterative feedback loops. We do some work, measure how it performs, and then figure out what's needed to improve it.

For example, imagine we build a fitness monitor that can estimate your 10K running time based on data collected from onboard sensors. To test whether it's working well, we can wait until you run an actual 10K and see whether the prediction was correct. If it's not, we can add your data to our training dataset and try to train a better model.

If we have a reliable internet connection, this shouldn't be too hard—we can just upload the data to our servers. But part of the magic of edge AI is that we can deploy intelligence to devices that have limited connectivity. In this case, we might not have the bandwidth to upload new training data. In many cases, we might not be able to upload anything at all.

This presents a big challenge for our application development workflow. How do we make sure our system is performing well in the real world when we have limited access to it? And how can we improve our system when it's so difficult to collect more data? This is a core topic of edge AI development and something we'll be covering heavily throughout this book.

Compute is diverse and heterogeneous

The majority of server-side AI applications run on plain old x86 processors, with some graphics processing units (GPUs) thrown in to help with any deep learning inference. There's a small amount of diversity thanks to Arm's recent server CPUs, and exotic deep learning accelerators such as Google's TPUs (tensor processing units), but most workloads run on fairly ordinary hardware.

In contrast, the embedded world includes a dizzying array of device types:

- Microcontrollers, including tiny 8-bit chips and fancy 32-bit processors
- System-on-chip (SoC) devices running embedded Linux
- General-purpose accelerators based on GPU technology
- Field programmable gate arrays (FPGAs)
- Fixed architecture accelerators that run a single model architecture blazing fast

Each category includes countless devices from many different manufacturers, each with a unique set of build tools, programming environments, and interface options. It can be quite overwhelming.

The diversity of hardware means there are likely to be multiple suitable systems for any given use case. The hard part is choosing one! We'll cover this challenge over the course of the book.

"Good enough" is often the goal

With traditional AI, the goal is often to get the best possible performance—no matter the cost. Production deep learning models used in server-side applications can potentially be *gigabytes* in size, and they lean on powerful GPU compute to be able to run in a timely manner. When compute is not an obstacle, the most accurate model is often the best choice.

The benefits of edge AI come with some serious constraints. Edge devices have less capable compute, and there are often tricky choices involved with trading off between on-device performance and accuracy.

This is certainly a challenge—but it's not a barrier. There are huge benefits to running AI at the edge, and for a vast number of use cases they easily outweigh the penalty of a little reduced accuracy. Even a small amount of on-device intelligence can be infinitely better than none at all.

The goal is to build applications that make the most of this “good enough” performance—an approach described elegantly by Alasdair Allan as Capable Computing. The key to doing this successfully is using tools that help us understand the performance of our applications in the real world, once any performance penalties have been factored in. We’ll be covering this topic at length.

Tools and best practices are still evolving

As a brand-new technology that has only begun to reach mass adoption, edge AI still depends on tools and approaches that were developed for large-scale, server-side AI. In fact, the majority of AI research is still focused on building large models on giant datasets. This has a couple of implications.

First, as we’ll see in Chapter 5, we’ll often find ourselves using existing development tools from the fields of data science and machine learning. On the positive side, this means we can draw from a rich ecosystem of libraries and frameworks that is proven to work well. However, few of the existing tools prioritize things that are important on the edge—like small model sizes, computational efficiency, and the ability to train on small amounts of data. We often have to do some extra work to make these the focus.

Second, since edge AI research is fairly new we’re likely to see extremely rapid evolution. As the field grows, and more researchers and engineers turn to focus on it, new approaches for improving efficiency are emerging—along with best practices and techniques for building effective applications. This promise of rapid change makes edge AI a very exciting field to work in.

Summary

In this chapter, we’ve explored the terminology that defines edge AI, learned a handy tool for reasoning about its benefits, explored how moving compute to the edge can increase access to technology, and outlined the factors that make edge AI different from traditional AI.

From the next chapter onward, we’ll be dealing with specifics. Get ready to learn about the use cases, devices, and algorithms that power edge AI today.

CHAPTER 5

Tools and Expertise

The edge AI development workflow includes many highly technical tasks, and most projects will require skills and expertise pooled by a team of experts.

The first section of this chapter is a guide to building the team that will turn your ideas into reality. Even if you're still at an early stage, it's helpful to understand the types of skills that will be important and the challenges you can expect to encounter. AI is all about automating human insights, so it's vital that you have the right insights on your team.

The second part of the chapter, starting with "Tools of the Trade" on page 38, is designed to help get you to grips with the key technical tools for working with edge AI. If you're still early in your product development journey, you may want to skim over some of the details—and then use this chapter as a reference once you've come up with some concrete ideas and are ready to start.

Building a Team for AI at the Edge

Edge AI is a truly complete technology. As a topic, it makes use of knowledge from everything from the physical properties of semiconductor electronics all the way up to the engineering of high-level architectures that span devices and the cloud. It demands expertise in the most cutting-edge approaches to artificial intelligence and machine learning along with the most venerable skills of bare-metal embedded software engineering. It makes use of the entire history of computer science and electrical engineering, laid out end to end.

Nobody in the world holds deep expertise in every subfield of edge AI. Instead, the people working at the heart of the field rely on assembling networks of experts who they can look to for insight into other pieces of the puzzle. If you're building an edge AI product, you may have to do the same for yourself.

The best team for edge AI is one that has broad, cross-disciplinary knowledge, direct experience working on the problem domain, and comfort working in an iterative development process. The best executed products so far have come from teams with direct experience of the issue they are trying to solve: they've taken their existing knowledge and used it to inform their edge AI product.

It isn't necessary for a single team to have experts in every subfield of edge AI. The absolute bare minimum is probably two roles:

- A domain expert, who has deep insight into the problem to be solved
- An embedded engineer with experience developing for devices similar to the target

There's no reason why these two roles can't be filled by the same person. However, without experience working with machine learning or other AI algorithms, they'll have to rely very heavily on end-to-end platforms designed to guide non-ML experts through the process of algorithm creation.

If you're a solo developer without embedded development experience, you can level up your skills by building some non-AI projects on your target hardware. To make your life easier, you might consider sticking to SoC-level hardware, since embedded Linux development is much easier than bare metal. If you're using an end-to-end edge AI platform it should be relatively simple to deploy your model.

Determination and some scrappy improvisational skills can go a long way: we've seen plenty of scientific researchers build their own AI-powered hardware with relatively simple embedded skills.

While many problems can be solved by a minimal team, the most complex problems will require more heavy lifting. The remainder of this chapter lays out the roles and responsibilities that can potentially be important, which will hopefully give you a sense of what you need for your own team. It also talks through the challenges of hiring for edge AI.

Domain Expertise

Domain expertise, as we'll learn about in detail in Chapter 7, is by far the most essential component of your team. If you have nothing but domain expertise and a budget, you can still hire developers and get a product built. But if nobody on your team has a deep understanding of the problem you're trying to solve, it's very unlikely you'll be able to solve it. In fact, there is a fair chance you may end up trying to solve the wrong problem or creating a solution that nobody needs.

It would be difficult to build any kind of quality product without domain expertise, but building an AI product without domain expertise is almost impossible. The goal of edge AI is to distill expert knowledge into a piece of software and use it to automate a process. As we learned earlier in the book, intelligence means knowing the right thing to do at the right time. But how can we build a system that does that if we do not know it ourselves?

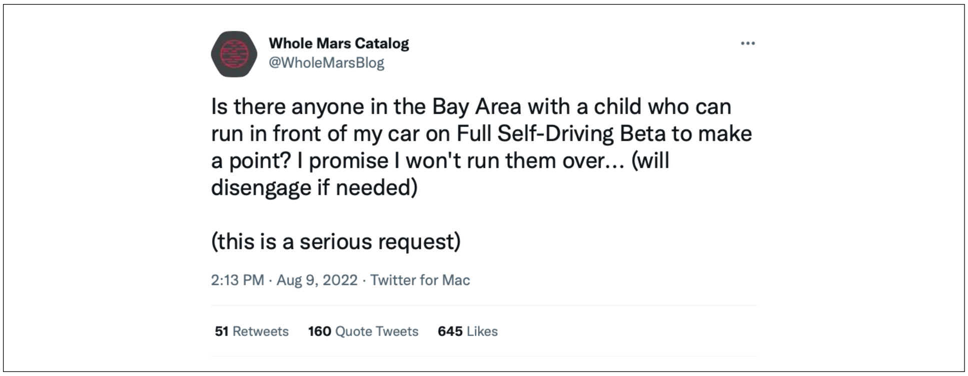
If you aren't a domain expert yourself, your first job is to find someone who is. Your second job is to have them validate the solution that you are planning to build. Here are some of the questions to ask them:

- Does the problem you wish to tackle really exist?
- If it exists, is it a useful problem to solve?
- Are there solutions to the problem that already exist?
- Would your proposed solution actually help solve the problem?
- Does your proposed solution sound feasible to build?
- If you build your solution, would anybody in the field want to buy it?

You should hopefully be able to ask someone these questions without having to pay too much money: they're the basic questions that any genuine domain expert would be thinking about if you were to offer them a job. You should make sure you pay attention to their answers, even if you disagree. If a genuine expert is telling you something is a bad idea, there's likely some truth to it.

Domain expertise should be at the heart of your organization and part of your core team. Your experts will be involved with so many aspects of the project that it isn't feasible for them to be peripheral members. That said, the ideal situation is that you have domain expertise at every level of your organization. For example, in addition to your core expertise you may have engineers, board members, and advisors who all have experience in the relevant area. Their combined insight will help your team anticipate and mitigate risk.

If you are unable to find anyone with the required expertise, you should abort your project before it gets started. There's simply no way to work ethically if you don't have the appropriate knowledge. Your project may be violating some golden rule of the field—and you wouldn't have any way to know about it. It's not acceptable to test unqualified functionality on your customers, as Figure 5-1 makes clear. It's so difficult to establish a feedback loop with performance in the field that you likely won't know what's going wrong.


Whole Mars Catalog
@WholeMarsBlog

Is there anyone in the Bay Area with a child who can run in front of my car on Full Self-Driving Beta to make a point? I promise I won't run them over... (will disengage if needed)

(this is a serious request)

2:13 PM · Aug 9, 2022 · Twitter for Mac

51 Retweets 160 Quote Tweets 645 Likes

Figure 5-1. Using customers to validate your solution is a horrible idea (Twitter, 2022)

If you are truly convinced that you have a good idea, you may have to spend some time developing the required expertise yourself.

Diversity

In addition to domain expertise, the other essential property your team should aim for is diversity. As we discussed in Chapter 2, one of the best defenses against societal issues is to build a team with diverse perspectives.

It can be helpful to think of workplace diversity in terms of four core areas:[1]

Internal
: Internal diversity reflects the things that a person is born with and didn't choose for themselves. Some of these areas include age, nation of origin, race, ethnicity, sexual orientation, gender identity, physical ability, and personality types.

External
: External diversity includes the things we pick up along the way, whether due to influence by external factors or due to conscious choices. Some examples are socioeconomic status, life experiences, education, personal interests, family status, location, and religious beliefs.

Organizational
: Organizational diversity relates to a person's role within an organization. This might include their place of work, job function, level within a hierarchy, pay level, seniority, or employment status.

1 See "What Are the 4 Types of Diversity?" for more information on the four core areas.

Worldview

Diversity in worldview relates to how a person sees the world. It can include things like ethical frameworks, political beliefs, religious beliefs, personal philosophy, and general outlook on life.

As a result of differences in these four areas, every person has a different set of experiences that make their perspective unique. This unique viewpoint means that they will see the same situation in different ways. As a team building technology products, a diversity of perspectives is incredibly valuable because it allows the organization to view the problem and the proposed solutions from a multitude of different angles.

This provides a significant advantage over organizations that lack diversity. You'll be more capable of identifying all the nuance in a given situation, which has huge benefits when mapping out the space of possible solutions. Perhaps someone's personal experiences will translate into an amazing idea that nobody else would have thought of.

Even more importantly, diverse perspectives will help you identify issues with your own product. For example, you may find that different people naturally come up with different axes on which to evaluate your product's performance. A person with kids is more likely to consider the need for a product to cope well with family life, and someone with a physical disability may be more likely to think about accessibility.

This isn't to say that the members of your team should stand in for domain experts in these areas: just because a person has a disability does not mean they are automatically your official accessibility expert, a role they may neither want nor be qualified for. However, the fact that your team has diverse perspectives hopefully means that they are more likely to consider the *need* to bring on an accessibility expert.

It's not enough to just have a diverse team: individuals have to be comfortable sharing their input, and the rest of the organization has to actually listen to them. The work of building that type of environment is beyond the scope of this book, but there's plenty of literature on the subject. A good place to start is this introduction to psychological safety from Google, who have found that teams where individuals can confidently speak up are far more effective.

Another key idea is that you should make use of perspectives from your entire organization. Beyond the people working directly on the product, you should draw feedback from everyone you can—from executives to entry-level workers. This will help you avoid blind spots in your insight. At many large tech companies, employees are encouraged to sign up to test new products that are still in development,[2] allowing development teams to access insight from across the entire company.

2 It's part of a strategy known as "dogfooding," covered in Chapter 10.

As with everything in iterative development, this process is all about building feedback loops that will help make your product over time. You should create systems to gather the perspectives of your diverse team from the very earliest stages when you're still sketching out ideas.

It's not always feasible for a single team to include all of the necessary diversity of perspectives. For example, you may need input on a product from young children, who are unlikely to be paid employees of your organization! One way to ensure these perspectives are included is to create a budget for having focus groups with these types of people throughout the course of your project.

Another way of broadening perspectives is to find a diverse group of advisors who can help inform your decisions. Assembling an advisory board that combines expertise in key areas with diverse representation is a powerful tool for helping you make the right decisions. They can act as a review board who can help you understand whether you are meeting your goals or veering off course.

Regardless of whether you have a large team, you should be relentless in seeking feedback from the people affected by your product—the most diverse group of all.

The Costs of Diversity

It's worth noting that diversity comes with some costs. Beyond the practical expense of compensating people for their time, diverse teams may find it harder to reach agreement on things like values and goals.

The leadership of a project needs to be prepared for this and may have to make some decisions that do not have full agreement. Documenting the reasoning behind any decisions, along with any dissenting views, is essential in ensuring a team can track its decision making and improve it over time.

That said, if there's fundamental disagreement on a particular issue, it can be a sign of significant risk.

Stakeholders

The stakeholders of your project are all of the people and communities who are potentially affected. This includes people within your organization, your customers, the end users of your system, and anybody who may be impacted—both directly and indirectly.

For a system to be effective, and for it to avoid causing harm, the needs and values of stakeholders must be considered. For example, if your system will come into contact with members of the public then it is important that they are considered as stakeholders, and that the project is designed with them in mind.

The best way to understand the needs and values of stakeholders is to ask them directly. They should be represented throughout your development workflow, from ideation to end of life.

Stakeholders can be identified using a well-established tool known as stakeholder mapping. You should make sure your team includes someone who is familiar with the process.

Roles and Responsibilities

Building a product takes a village full of people, and the next section of this chapter outlines some of the roles that are required. Your project may require roles that are not included here; these are just the most common ones that directly participate in the edge AI workflow.

You don't need to hire an individual person for each role. It's perfectly possible for the same person to play multiple roles in a project, and early on it may be the case that all of your prototyping is done by a single person.

For ease of digestion, we'll divide up the roles by type.

Knowledge and understanding

The roles in this category are critical to understanding the problem and solving it in the right way:

Domain expert
: The starring role in the play, the domain expert brings a deep understanding of the project area. While a product manager's job is to understand how a project fits into the surrounding context (such as the market), a domain expert is the person who understands the science of the situation. For example, an industrial automation project might require a domain expert in the relevant industrial processes, and a healthcare project might need an expert in the related areas of medicine and biology.

Ethics and fairness expert
: The ethics and fairness role is required in order to avoid making the types of mistakes that often result in harmful or ineffective products. They need a strong understanding of the technologies that will be used to solve a problem, the types of pitfalls that can emerge, and the processes that need to be followed. Domain expertise is important, too, since ethical issues can be specific to a domain.

Planning and execution

These high-level roles are important in guiding the project down the right path as it travels from ideation to launch and long-term support:

Product manager
: A product manager is responsible for making decisions about the product: what it should be, what it should do, and who it should serve. Their job is to deeply understand the problem and the market, and work with those in technical roles to design and implement an effective solution. They lead through influence, pulling together different threads to weave a product that fits the right requirements.

Project manager
: The project management role involves coordinating the execution of complex tasks across groups of people. For example, a project manager may organize the collection of a dataset that will be used to build and create a product.

Program manager
: Program managers coordinate high-level strategies that are made up of multiple projects. For instance, a company planning to incorporate edge AI into multiple parts of its business to make cost savings may use a program manager to coordinate the process.

Algorithm development

These roles are involved in the exploration of datasets and the design of algorithms—along with mechanisms for evaluation of the system that is being built. This work can increasingly be done by nonexpert users who are using end-to-end platforms, but it's always good to have some solid experience to draw on to avoid making rookie mistakes:

Data scientist
: The data science role is responsible for gathering, maintaining, and understanding the data that underlies an edge AI project. They have skills in data cleaning, analysis, and feature engineering. This role may often encompass machine learning work, but it could just as easily be distinct.

DSP engineer
: A DSP engineer develops and implements DSP algorithms. They typically have strong skills in both algorithm development and low-level programming. DSP is extremely important in most edge AI projects—with the exception of those that combine deep learning with image data, since images are typically input without much processing.

ML practitioner

Machine learning practitioners spend their time trying to solve problems with ML. An ML practitioner will try to frame a problem in terms of different types of learning algorithms. They will then work with a dataset, attempting to develop algorithms that solve the problem. A key part of their role is determining how to evaluate algorithms and their performance, both in the lab and in the field.

In an edge AI project, DSP engineers and ML practitioners work very closely together, since DSP is a sophisticated form of feature engineering—which is a key part of the ML workflow.

Product engineering

This set of roles leads development of the product itself. They create the hardware and application code, and they implement the algorithm in a form that works efficiently on-device:

Hardware engineer

A hardware engineer designs the hardware that powers a product. This design includes both the sensors that capture raw data and the processors that attempt to make sense of it, along with the design and layout of printed circuit boards.

It's critical for hardware engineers to work closely with those in algorithm development roles so that the hardware and algorithms support each other. This is a two-way street: algorithm design must be informed by hardware constraints, and hardware design must be informed by algorithm design.

Embedded software engineer

Embedded software engineers write the low-level code that brings a piece of hardware to life. Their code has to interface with sensors, run algorithms, and interpret their output in order to make decisions. They implement the embedded application itself.

Embedded ML engineer

Some embedded software engineers focus specifically on machine learning. Their job is to make sure that ML algorithms run as efficiently as possible on a particular piece of hardware. They may have deep knowledge of the mathematics behind machine learning, along with experience with low-level software optimization. They aren't necessarily an expert in data science, although they can likely train simple ML models.

This is a very new role, but it's growing in step with the edge AI space.

Industrial designer
: An industrial designer creates the physical design of the product. This is relevant to edge AI in that the physical design dictates many of the realities of sensor data collection: moving a sensor to another location on a product can completely change its typical output and make a dataset instantly obsolete. This means there needs to be significant communication between industrial design, electronics engineering, and algorithm development.

Software engineer
: Many projects involve software engineering outside of the embedded space. For example, a lot of edge AI projects involve a server-side component. Writing this backend code requires different skills to developing embedded applications, so a different type of engineer is needed.

Technical services

These supporting roles help keep the technical side of the development process running smoothly and manage the tools that keep the team productive and safe:

MLOps engineer
: An MLOps engineer is responsible for building and maintaining the MLOps solutions that are used by the rest of the team. It's essentially a DevOps role, but it requires strong understanding of the processes and demands of the edge AI workflow.

Security practitioner
: This role attends to the security needs of the team, its data, and the products that are produced. It is both a consulting role—helping other roles understand how to be secure in what they do—and a proactive role, putting measures in place that help reduce security risk.

Quality assurance engineer
: This role helps design and implement testing plans that put a product through its paces, allowing a team to understand whether the product is meeting its design goals. There's more about quality assurance in Chapter 10.

Hiring for Edge AI

A significant challenge of edge AI development is that as a very new field, there are not many people out there with experience working on it. At the time of writing, it is almost impossible to hire an engineer who has existing experience with edge AI: there are likely only a few hundred in the world, and most are still working on their first exciting edge AI projects and haven't had long enough to get itchy feet.

Fortunately, the fact that this is a new field means that even the most experienced engineers only have a couple of years' advantage. Recent advances in edge AI tooling, particularly in the form of end-to-end platforms, have massively reduced the barriers to entry. Hiring for edge AI has two main fronts where very specific knowledge is required: algorithm development and embedded engineering.

In the case of algorithm development, you'll likely be in the market for data scientists and ML practitioners. Some practitioners have backgrounds in applied engineering, solving practical problems in industry. Others may have a more academic background, investigating the principles underlying machine learning and coming up with new techniques.

Applied practitioners will have more experience with problem framing, which is very important in edge AI. This makes them a desirable choice, especially as an initial or solo hire. That said, academic researchers can still be a good fit for edge AI projects. They are less likely to have experience working within a typical software development environment and may take longer to ramp up. On the other hand, they are easier to hire than applied practitioners: there are simply more of them.

> ML research is very different from applied ML, and some ML researchers may feel bored routinely applying existing techniques rather than attempting to come up with new ones. Make sure that it's clear to candidates what the expectations are around a role to avoid disappointment on both sides.

One difficulty is that not many people in data science and machine learning have much experience with sensor data. While vision is a common modality, audio is less so, and time series sensor data is likely to be a mystery to most practitioners: while time series analysis is common in data science, it's not typically the type of high-frequency time series that are produced by electronic sensors.

Fortunately, DSP engineers have a similar workflow and toolchain to ML practitioners, and they are already experts in feature engineering for sensor data. The skills and experience of DSP engineers makes them well suited to learning embedded ML, so one potential avenue is to recruit DSP engineers and have them learn the basics of machine learning. A team composed of both DSP engineers and ML practitioners will have a much easier time than either role alone.

In terms of embedded engineering, the challenges vary. While working with deep learning interpreters (or code generated by a deep learning compiler) is often simply a matter of library integration, embedded engineers may sometimes have to dig into the internals to figure out when something is going wrong. In these cases, some knowledge and understanding of deep learning is definitely helpful. Embedded engineers may also end up being responsible for the onerous task of converting a model into the appropriate form to use on-device, which is definitely easier with some ML insight.

Another common task for embedded engineers is to implement classical ML models in software. There isn't yet a great embedded-specific C++ library for this but porting them is usually easy: there are reference implementations in higher-level languages that are simple to understand.

Unfortunately, finding an embedded engineer with existing ML knowledge is going to be a challenge for a while. That said, end-to-end platforms make things a lot easier, and eventually the number of experienced embedded ML engineers will grow. For now, it shouldn't be a blocker: a competent embedded engineer should be able to learn today's tools without too much trouble.

Learning Edge AI Skills

Over the last few years, some great resources have emerged for learning about AI at the edge. Like with most fields, there are two sides: theory and practice. Theory content will be most interesting to those who wish to contribute to advancing the field, while practical content is more helpful for those who wish to build products.

A word of caution: don't get lost in the weeds. Many people who wish to build AI products end up getting paralyzed by learning, exploring every rabbit hole they can rather than actually getting started on their projects. The reality is that this is a massive field, and you're never going to be able to learn it all. Be oriented toward action, learn enough to take your next step, and then reevaluate. Successful hardware products require teams, so figure out the minimum you need to know and then bring some experts on board.

Here are our top recommendations for both practical and theoretical content.

Practice

The final three chapters of this book, starting with Chapter 11, will walk through the edge AI workflow end to end with three real-world use cases: wildlife monitoring, food quality assurance, and consumer products.

Once you're done with that, here's some further content:

Introduction to Embedded Machine Learning (Coursera course)
: A highly rated online course intended as a practical introduction to the subject.

Computer Vision with Embedded Machine Learning (Coursera course)
: A follow-up to the first course, focused specifically on vision.

Applied Machine Learning (TinyML) for Scale (HarvardX course)
: This brilliant collection of courses focuses on the applied skills and big-picture expertise required for working with embedded ML.

TinyML Cookbook, a book by Gian M. Iodice (Packt, 2022)
: A practical book based around useful "recipes" that demonstrate various concepts within embedded ML.

TinyML, a book by Pete Warden and Daniel Situnayake (O'Reilly, 2020)
: A working introduction to embedded ML on microcontrollers, with examples focused on TensorFlow Lite for Microcontrollers.

Designing Machine Learning Systems, a book by Chip Huyen (O'Reilly, 2022)
: A fantastic book about the machine learning development workflow, geared toward server-side applications but still very relevant.

Making Embedded Systems, a book by Elecia White (O'Reilly, 2011)
: The best available practical introduction to developing embedded systems.

Theory

This content is for people who want to dig deeper into the theory of embedded machine learning. Remember, it's not a prerequisite for successful product development—so don't feel intimidated or get lost down the rabbit hole of studying.[3]

Tiny Machine Learning (TinyML) (HarvardX course)
: This set of courses overlaps with Applied Tiny Machine Learning (TinyML) for Scale, referenced earlier, but starts with the absolute fundamentals—which may not be necessary if you want to get building as quickly as possible.

The Scientist and Engineer's Guide to Digital Signal Processing, a book by Steven W. Smith (California Technical, 1997)
: A truly comprehensive guide to digital signal processing, available for free and as a hardcover book. A good resource for any non-DSP engineer who will be working seriously with DSP algorithms.

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, a book by Aurélien Géron (O'Reilly, 2022)
: A wonderful introduction to practical machine learning concepts and skills. A good resource for any non-ML engineer who will be working with ML algorithms.

3 Remember, the best way to learn is to build! Don't fall into the common trap of thinking you need to memorize all of the theory first. This field develops so rapidly that you can never hope to learn it all.

Deep Learning with Python, a book by François Chollet (Manning, 2021)
: Another fantastic introduction to ML, specifically focused on deep learning algorithms.

TinyML Foundation (YouTube channel)
: The TinyML Foundation hosts regular presentations on embedded ML. Typically highly technical, this content reflects the cutting edge of research and engineering.

TinyML papers and projects (GitHub repository)
: This repo is a goldmine of papers and resources related to the field.

Tools of the Trade

The story of edge AI is a story of tooling. In terms of raw technology, most of the basic ingredients required for putting artificial intelligence on edge devices have existed for a decade or more. However, these technologies—from capable embedded processors to deep learning models—tend to have a steep learning curve when they first become available.

Over time, however, our global technology ecosystem evolves tooling designed to manage the complexity and improve the usability of even the most challenging technologies. A rich combination of open source and commercial libraries, frameworks, and products have brought edge AI into the toolbox of the average embedded engineer.

A lot of this work has happened in the past two or three years, with libraries such as TensorFlow Lite for Microcontrollers[4] and end-to-end development platforms like Edge Impulse[5] bringing the technology over the threshold to enable mass adoption.

The following sections will walk through the tools that we think are most essential to AI at the edge. Successful teams will be at least passingly familiar with all of them.

End-to-End Platforms

End-to-end development platforms for edge AI incorporate many of the tools described in the following section, providing automated integration between them—along with a conscious, holistic design intended specifically for edge AI projects. They can massively reduce the complexity burden, making development much faster and less risky—and helping you avoid drowning in an ocean of unfamiliar tooling.

4 Founded by Pete Warden, then at Google, who kindly wrote the foreword for this book.

5 Which impressed this book's authors enough that we left jobs at Google and Arm to come and work on it.

End-to-end platforms are explained in their own section, "End-to-End Platforms for Edge AI" on page 64. While it's helpful to have a basic understanding of low-level tools, it's advisable to start with an end-to-end platform and only attempt to "roll your own" if the platform doesn't fully meet your needs. In that case, the best end-to-end platforms will integrate with other industry standard tools so that you can extend their functionality without losing the benefits.

Software Engineering

A large portion of edge AI involves software development, so modern software engineering tools are incredibly important. Here are some of the key contributors.

Operating systems

It's important to consider operating systems during both development and deployment. In development, your OS of choice will determine how easy it is to work with the extremely diverse set of software tools that make up the edge AI ecosystem. There's a bit of minor conflict between two different engineering traditions.

In embedded engineering it's historically common to use Windows as an operating system, and some embedded tools are written with this assumption. In contrast, the tools of data science and machine learning are typically best suited to a Unix-compatible environment such as Linux or macOS.

That said, this isn't a huge problem in practice. It's not strictly necessary for every member of a team to be able to run all of the tooling: for example, a machine learning engineer might train and optimize models with Linux and then hand them over to an embedded engineer who uses Windows. There are also plenty of tools for mixing environments, such as Windows Subsystem for Linux. In addition, it's common for more recent embedded toolchains to work fine in Unix environments—although embedded engineers may still prefer Windows as a familiar environment. The entire team at Edge Impulse, including both embedded and ML engineers, uses a combination of macOS and Linux virtual machines.

In deployment, operating systems are sometimes used on edge devices themselves. These are typically either embedded Linux (stripped-down distributions of Linux compiled to run on SoCs) or real-time operating systems (RTOS), which are special embedded-specific operating systems designed to run with minimal overhead. Both of these options, plus the option of no OS at all[6] (which is the most common case for microcontrollers), are fully compatible with edge AI.

6 Known as "bare metal."

Programming and scripting languages

The two most important programming languages for edge AI are Python and C++. Python is overwhelmingly the current language of choice for machine learning, thanks to a vast array of open source mathematical and scientific computing libraries and nearly 100% adoption by the machine learning research community. Since Python is also a first-class language for general software engineering, it beats domain-specific languages such as R.[7] The two most important deep learning frameworks, TensorFlow and PyTorch, are both written in Python, as are the incredible tools we'll encounter in "Mathematical and scientific computing libraries" on page 45. Python has its quirks, but it's the right language to use for developing edge AI algorithms—from machine learning to DSP.

C++ (pronounced *C-plus-plus*) is a ubiquitous language in modern embedded software engineering. While some embedded platforms only support C (a simpler language than C++ that shares some characteristics), the high-end embedded devices that are typically used for edge AI are generally programmed with C++. The ecosystem around C++ features numerous tools and libraries that can make development easier, which is lucky—since it's the only game in town for most microcontroller-based systems.

C++ is a low-level language that provides a huge amount of control over the underlying hardware. It takes a skilled engineer to write good C++ code, but it can be much faster than the equivalent written in a higher-level language such as Python.

It's interesting to note that most of the mathematical heavy lifting done by Python libraries is actually implemented in C++ under the hood: the Python code is just used as a convenient wrapper. This gives developers the best of both worlds.

You're also likely to use scripting languages, such as Bash, during the development process. They are used to chain together and automate the complex tools and scripts that help build applications and deploy them to devices.

In terms of targets, you can expect to almost always use C++ when working with microcontrollers. SoCs, which run full operating systems, are often a lot more flexible—you may be able to run high-level languages such as Python. The trade-off is that they are far more expensive and consume a lot more energy than smaller devices.

Since most targets require C++, you'll need to port any algorithms developed in higher-level languages (like Python) in order to deploy your work. There are some tools explained later that make this easier, but it's not always a simple process.

7 A popular language for statistical computing that is not typically used for purposes outside data analysis.

Dependency management

Modern software typically has a lot of dependencies, and AI development takes this to the next level. Data science and machine learning tools often require absurd numbers of additional third-party libraries; installing a major deep learning framework such as TensorFlow brings everything from web servers to databases along for the ride.

Things can get complex on the embedded side, too, since signal processing and machine learning algorithms commonly require sophisticated, highly optimized mathematical computing libraries. In addition, the compilation and deployment of embedded C++ code often requires a rat's nest of dependencies to be present on a machine.

All of these dependencies can be an absolute nightmare and managing them is truly one of the most challenging parts of edge AI development. Various techniques exist to make it easier, from containerization (see the next section, "Containerization" on page 41) to language-specific environment management.

For Python, one of the most helpful tools is called Poetry. It aims to simplify the process of specifying, installing, and isolating dependencies in multiple environments on a single machine.[8] Other essential tools include OS-specific package management systems like aptitude (Debian GNU/Linux) and Homebrew (macOS).

One of the worst parts of dependency management comes when attempting to integrate different parts of a system together. For example, a model trained with one version of a deep learning framework may not be compatible with an inferencing framework released slightly later. This makes it extremely important to test systems end to end very early in the development process, to avoid nasty surprises later on.

Containerization

Containerization is the use of OS-level techniques to run software inside of sandboxed environments called *containers*. From inside, a container appears entirely distinct from the machine that is running it. It can have a different operating system and dependencies, and limited access to system resources.

Edge AI involves many different toolchains, used for everything from machine learning to embedded development. These toolchains often have mutually incompatible dependencies. For example, two toolchains might require entirely different versions of a language interpreter. Containerization is a powerful tool for enabling these incompatible toolchains to happily live side by side on a single machine.

8 The most common Python dependency management tools are pip and Conda; Poetry is a relative newcomer but is highly recommended.

Containers are typically state-free and highly portable. This means that you can treat an entire painstakingly configured machine—described in a special syntax—as a command-line program that does a specific task. You can chain these together in order to perform useful work, and you can easily run them on different machines for a distributed computational environment.

It's also possible to run containers on embedded devices, typically within embedded Linux on an SoC. This can be an interesting way to package your software and its dependencies for distribution, although there is some overhead involved.

The most popular tools for containerization are Docker and Kubernetes. Docker is typically used locally on a development workstation, while Kubernetes is used to run clusters of containers within distributed computing infrastructure.

Distributed computing

Distributed computing is the idea of running different processes on different machines, potentially located anywhere in the world and connected via the internet. It's a more flexible way to approach computation than the use of single, high-powered mainframes and supercomputers, and it's the architectural style underlying the majority of modern computing.

Distributed compute is important to edge AI for many reasons. First, edge AI is an example of distributed computing! Computation is performed at the edge, where the data is created, and the results are either used locally or sent across the network.

Second, managing datasets, developing algorithms, and training machine learning models can be highly compute and storage intensive. This makes distributed computing a good fit for these parts of the process. For example, it's common to rent a highly capable remote server in order to train deep learning models—as opposed to having to buy and maintain a powerful machine for your office.

The task of organizing and controlling distributed computing infrastructure is called *orchestration*. There are many open source orchestration tools available, designed for different tasks. Kubeflow is an orchestration framework designed for running machine learning workloads across multiple machines.

Cloud providers

Businesses like Amazon Web Services, Google Cloud, and Microsoft Azure provide on-demand distributed computing resources that are available to anyone willing to pay for them. This type of distributed compute is known as "cloud compute," since diagrams of computer networks typically use a cloud symbol to signify resources that are located outside of the local network.

Cloud providers host most of the world's websites. They take care of the physical hardware and the network configuration, allowing developers to focus on building applications rather than managing equipment. They make heavy use of containerization to allow many different workloads to live side by side on the same infrastructure.

It's common for edge AI projects to use cloud compute for storing datasets, training machine learning models, and providing a backend from which edge devices can send and receive data. In some cases, such as Chapter 8, AI algorithms running on cloud servers work in unison with those on edge devices in order to provide a service.

Working with Data

Data is a key ingredient of edge AI applications, and many tools exist for collecting, storing, and processing data.

Data capture

Obtaining data from the field can be difficult since there's often limited connectivity available at remote locations. Two useful tools are data loggers and mobile broadband modems.

Data loggers are small devices designed to capture and log data collected by sensors in the field. They typically have a large amount of persistent storage for collecting sensor readings and can either be battery powered or connected to a permanent power source. The benefit of using a data logger is that you can begin collecting data immediately, before designing and building any of your own hardware. The downside is that data needs to be collected manually, by physically connecting to the logger.

Mobile broadband modems provide a wireless internet connection, typically via cellular networks—although satellite connections are also available. They can potentially transmit data from almost anywhere in the world, although connectivity depends on local availability and conditions. They offer the convenience of immediate data availability. However, data rates can be quite expensive, and wireless communication consumes a lot of energy, so they are not feasible for use in all situations.

IoT device management

Many platforms exist for communicating with IoT devices, managing their operation, and collecting data from them. Using them typically involves integrating either libraries or APIs into your embedded software. The software then connects with a cloud server that you can use to control the device.

These platforms can be convenient for collecting sensor data, especially in brownfield deployments where device management software may already be in use.

Data storage and management

As you collect your dataset, you'll need somewhere to store it. This can be as simple as comma-separated files on a hard disk—or as complex as a time series database designed specifically for storing and querying time series data. We will cover some of these options in Chapter 7.

Data storage solutions are designed for various purposes. Some are intended to be extremely fast at real-time querying of data, while others are designed to be as robust as possible against data loss. For edge AI applications, you're typically dealing with data in a "batch" mode, so performance isn't usually the most important factor. Instead, you should aim for a simple solution that fits the type of data you are collecting.

It's pretty common for AI datasets to be stored in the filesystem, without any type of database at all. Filesystems are designed for this type of data, and filesystem tools such as those available for the Unix command line can be helpful in manipulating it efficiently. Python's scientific computing ecosystem includes a lot of tools that are great at reading data from disk and helping you explore and visualize it.

While a fancy database isn't necessary, storing data in the right format is still important. As we will learn in Chapter 7, sensor readings themselves should be stored in an efficient, compact binary representation such as CBOR, NPY, or perhaps TFRecord —which is specifically designed for high performance during machine learning training. Metadata about readings should be stored in separate files (known as *manifest files*) or in a simple database. Separating data from metadata in this way allows you to efficiently explore and manipulate datasets without reading massive files into memory.

Data pipelines

A data pipeline is a process that takes raw data and transforms it for use in a task, such as training a machine learning model. It's the way that data engineers automate things like data cleaning and wrangling. A typical data pipeline might take raw sensor data, filter it, combine it with other data, and write it into the correct format for training a machine learning model.

Many tools exist for defining data pipelines, some more complex than others. Edge AI data pipelines tend to involve very large amounts of relatively simple data, so avoid tools that are designed for working with structured data (such as data stored in relational databases). Instead of querying capabilities, look for high throughput and enough flexibility to run arbitrary signal processing algorithms.

Many cloud providers have features for running data pipelines in their distributed infrastructure. Some end-to-end platforms for edge AI make data pipelines a core feature and are designed specifically for the characteristics of sensor data.

Algorithm Development

Algorithm development is where most of the tooling complexity lives; there's a real galaxy of software available to help with the process. Some software is better suited to edge AI than others.

Mathematical and scientific computing libraries

The Python community has created some legitimate marvels of software engineering in the form of various open source libraries for performing mathematics and analysis of numbers. Some of the most important ones are:

NumPy
: NumPy describes itself as "the fundamental package for scientific computing with Python," and it's absolutely true. It provides the high-performance backbone for most Python-based numerical computing, and it has a wonderful API that lets you do complex things to large arrays of numbers with minimal effort. Its file format, NPY, is a convenient way to store sensor data.

pandas
: What NumPy is to arrays, pandas is to tables of data. It provides an almost magically intuitive syntax for querying and transforming any information that can be organized into rows and columns. Pandas works with NumPy, so you can use it to help explore your sensor data; it's super-fast.

SciPy
: SciPy provides a collection of fast implementations of algorithms that are essential to scientific computing. It's used heavily in developing DSP algorithms, and it's the magic that powers many other tools.

scikit-learn
: The library scikit-learn, built using NumPy and SciPy, provides a huge library of implementations of machine learning algorithms, along with the tools needed to feed them with processed data and evaluate their performance. Its API is designed so that you can plug its components together interchangeably, meaning you can easily compare and combine different algorithms. It's the gold standard for classical machine learning in Python, and its data processing and evaluation tools are often used even when training deep learning models with other frameworks.

Data visualization

When working with data, visualization is an essential tool—especially when the data concerned is digital signals. Graphs and charts allow us to represent and interpret numeric information that would otherwise be incomprehensible. The Python ecosystem has some fantastic libraries for visualizing data. They can be quite complex to get to grips with—especially if you want to customize visualizations beyond the provided defaults—but once you get the hang of them they can quickly turn rows of numbers into clear insight.

The two most common libraries are Matplotlib and seaborn. Matplotlib provides a million different ways to create data visualizations; it's commonly used to create the figures in scientific publications. Its syntax can be a little challenging, but it's so popular that a quick web search will usually help you figure out what you're trying to do.

Seaborn is built on top of Matplotlib and is designed to tame some of the complexity, making it easier to build attractive visualizations like the one in Figure 5-2 without getting tangled in difficult APIs. It's made specifically to pair well with pandas.

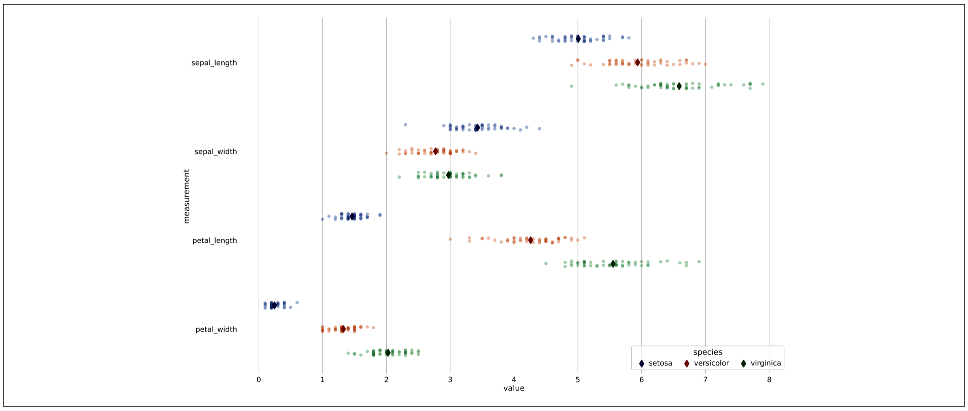


Figure 5-2. This plot shows the ranges and means for various columns in a dataset of plant measurements; it's one of the visualizations in seaborn's example gallery

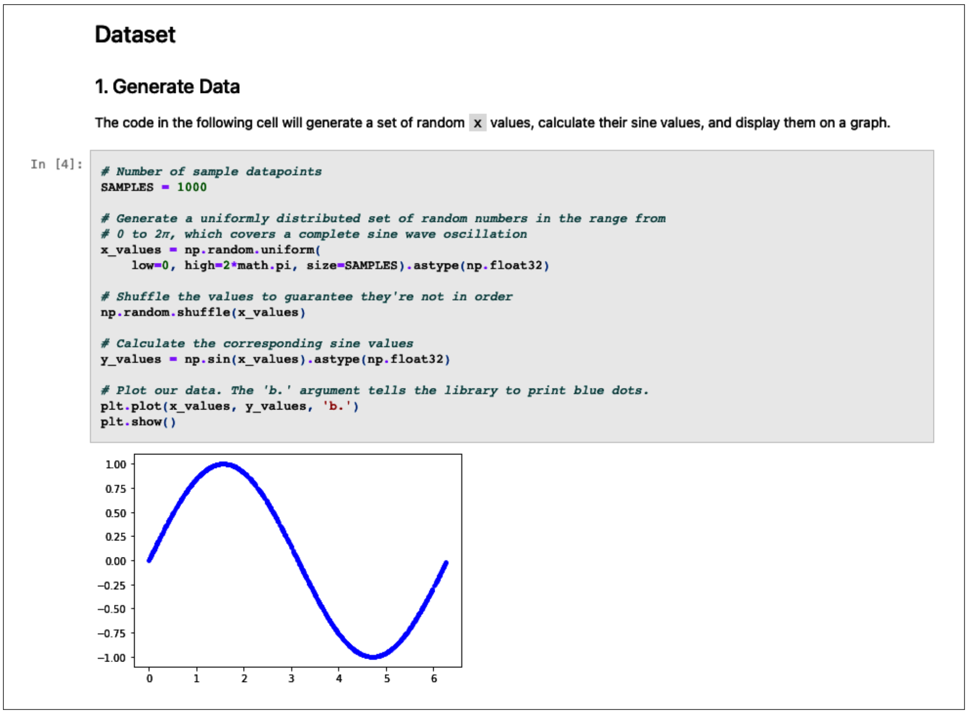
Seaborn and Matplotlib output image files—but some visualization libraries, such as Plotly, produce interactive visualizations that can be explored dynamically.

Interactive computing environments

Edge AI development involves a lot of exploration that lives outside of the context of routine software engineering. Exploratory data analysis, digital signal processing, and machine learning all have a workflow that involves trying different ideas and quickly visualizing the results.

Various interactive environments exist for this purpose. Rather than just running a script and writing the results to a file or having to build an entire web application just to express information visually, interactive computing environments allow code and visualizations to exist side by side in the same editor.

The most important interactive environment for Python code is called Jupyter Notebook. Inside a notebook you can write and run Python code, and the output of the code is displayed alongside. This includes any visualizations you generate using libraries such as Matplotlib—as seen in Figure 5-3.



Dataset

1. Generate Data

The code in the following cell will generate a set of random `x` values, calculate their sine values, and display them on a graph.

In [4]:

```
# Number of sample datapoints
SAMPLES = 1000

# Generate a uniformly distributed set of random numbers in the range from
# 0 to 2π, which covers a complete sine wave oscillation
x_values = np.random.uniform(
    low=0, high=2*math.pi, size=SAMPLES).astype(np.float32)

# Shuffle the values to guarantee they're not in order
np.random.shuffle(x_values)

# Calculate the corresponding sine values
y_values = np.sin(x_values).astype(np.float32)

# Plot our data. The 'b.' argument tells the library to print blue dots.
plt.plot(x_values, y_values, 'b.')
plt.show()
```

Figure 5-3. A screenshot from a Jupyter Notebook, showing a mixture of rich text, code, and the output of the code; the featured notebook is from the TensorFlow Lite for Microcontrollers Hello World example

This allows you to build interactive living documents that contain both the implementation of algorithms and the results of running them. They're valuable as both an interactive tool for experimentation and as documentary evidence of the work that you've done. A common workflow is to experiment with algorithms in a notebook until you find the best candidate, then port the code into regular Python scripts once you know it works well.

Jupyter can be run locally, but there are also Jupyter-based hosted environments. One of these is Google Colab, and another is Amazon SageMaker. Both can be used without cost but will provide additional compute for a fee.

Another common environment for interactive computing is MATLAB, which combines a similar interactive environment with its own programming language. It's common in academia and engineering, but as a closed-source commercial product that costs money to license, it's less popular with software engineers. It's quite likely that those with a background adjacent to electrical engineering are familiar with MATLAB, including DSP engineers.

There's even an interactive environment designed specifically for edge AI. The OpenMV IDE is an open source product created by the OpenMV team to support development of machine vision applications. It makes it easy to test and implement algorithms that interpret visual information, which can subsequently be deployed to both OpenMV's hardware, and to other targets. The OpenMV IDE is unique in that it can be connected to a camera-equipped hardware device and display the results of algorithms running in real time.

Digital signal processing

DSP algorithm development is typically performed in Python or MATLAB. Either environment can be used, with individual DSP engineers typically preferring one over the other.

In Python, SciPy's `scipy.signal` module provides implementations of a lot of important DSP algorithms. In MATLAB, the signal processing and image processing toolboxes are very helpful.

MATLAB has some nice GUI-based tools that reduce the amount of programming required for algorithm development, but Python has the advantage of being directly compatible with the toolchains used for training machine learning models—as well as being free.

An increasingly popular third choice is GNU Octave, designed to be a free, open source alternative MATLAB.

Deep learning frameworks

The ecosystem of deep learning tools is dominated by two wildly popular open source frameworks, written for Python: TensorFlow, created by Google, and PyTorch, created by Meta.[9] Each framework originated as an in-house system for training deep learning models, and they both reflect the priorities of their respective sponsors.

Deep learning frameworks are different from typical software libraries (like NumPy or scikit-learn) in that they attempt to provide entire suites of tools under a single banner. TensorFlow and PyTorch both include systems for defining and training machine learning models, handling data, coordinating distributed systems, deploying to different types of compute, and much more.

Example 5-1. A simple deep learning model architecture being defined and trained using Keras, the high level API of TensorFlow

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Defining the model architecture
model = Sequential()
model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=10, activation='softmax'))

# Setting up the training process
model.compile(loss='categorical_crossentropy',
              optimizer='sgd',
              metrics=['accuracy'])

# Training the model
model.fit(x_train, y_train, epochs=5, batch_size=32)

# Evaluating the model
loss_and_metrics = model.evaluate(x_test, y_test, batch_size=128)
```

The history of both tools has led to TensorFlow being the primary framework for industry, while PyTorch is the preferred tool of deep learning researchers.[10] A big part of this is due to the TensorFlow ecosystem having more options available for model deployment, and this is especially relevant for edge AI.

At the time of writing, the majority of tools for optimizing models for efficiency and deploying them to edge devices have been written to integrate with the TensorFlow ecosystem. TensorFlow and PyTorch have different formats for storing models, and

9 Back when they were known as Facebook.

10 The history and comparison between the two frameworks is quite interesting, and this fantastic blog post from AssemblyAI does a great job of summarizing it.

while there are ways to convert between them, it isn't a straightforward process.[11] This means that the majority of ML engineers working in edge AI currently use TensorFlow.

Since PyTorch is the framework of choice for researchers, many of the newest model architectures are available first in a PyTorch format. This can be frustrating for industry developers who are using TensorFlow for its deployment capabilities. Fortunately, most of the researchers focusing on producing smaller, more efficient models that are suited for edge deployment are doing so within the TensorFlow ecosystem. The area where model incompatibility is most frustrating is in visual object detection since the training code for object detection models tends to be complex and difficult to port from one framework to another.

At the time of writing, TensorFlow is the best choice of framework for edge AI development. Developers using PyTorch will struggle with a complex and unreliable conversion process when they attempt to deploy their models. It will be interesting to see how this evolves over time as the PyTorch ecosystem matures.

Model compression and optimization

Edge devices typically require small, efficient models—especially in a deep learning context, where parameter count and computational requirements can scale up fast. In Chapter 4, we learned about the various techniques available for improving the performance of models. Some of them are applied during training, while others happen afterwards.

Compression and optimization tools are generally available either as part of deep learning frameworks or by hardware vendors whose hardware supports particular optimizations. The TensorFlow Lite converter has become the de facto standard for operator fusion and basic quantization, with the TensorFlow Lite model file format becoming close to a standard in the industry.[12] Still in the TensorFlow ecosystem, the TensorFlow Model Optimization Toolkit provides a collection of open source tools that cover other types of optimization and compression.

It's worth remembering that most optimization approaches also require special tooling at inference time, covered later in the sidebar "Inference and Model Optimization" on page 58. At the time of writing the best supported optimization approach is quantization, with 8-bit quantized operator implementations widely available. Other techniques are less well supported, with sparsity being the biggest red herring: it sounds impressive, but there's currently very little hardware that supports it.

11 In fact, it can be an absolute nightmare even for the most experienced developers.

12 Other formats are in use, such as ONNX, but the TensorFlow Lite format is by far the most popular.

Experiment tracking

Algorithm development is an iterative, exploratory process, and over the course of a project you're likely to make hundreds or thousands of different attempts at getting something that works acceptably well. It's important to keep things scientific, testing ideas systematically rather than just making random changes and hoping for the best. To achieve this, you'll need some kind of system for tracking experiments.

A typical experiment might involve taking a specific set of data samples, applying a particular DSP algorithm, using the features to train a machine learning model with a unique set of hyperparameters, and then testing the model on a standard test dataset. This situation has a lot of variables: the choice of samples, the DSP algorithm, the model, and its parameters.

Experiment tracking tools are designed to keep a log of which experiments are run, how their variables are set up, and what the results are. They try to organize what would otherwise be an unreliable, informal process of taking notes in a notebook and trying not to forget any details. Experiment trackers can also store the artifacts that result from experiments: training scripts, datasets, and trained models. This is helpful in understanding and reproducing your work at a later stage.

Experiment trackers are available as both open source packages and hosted commercial products. One of the simplest options is TensorBoard, an official part of the TensorFlow ecosystem.[13] TensorBoard provides a simple web interface for visualizing and comparing the logs collected during training runs, along with some very powerful tools for optimizing and debugging training code. It's useful for keeping track of basic experiments, although it isn't designed as a persistent datastore that will last the lifetime of a project, and it doesn't work well if you are running very large numbers of trials.

A more sophisticated open source option is MLflow. It's a complex web application, backed by a database that can track experiments, store trained models, and package data science code so that experiments can be reproduced easily. It's better suited to long-term use than TensorBoard, and it can scale to track many thousands of experiments. It doesn't have the same optimization and debugging features as TensorBoard, which remains the tool of choice for improving the computational performance of training.

Many commercial products exist to help with experiment tracking. A notable option is Weights & Biases, which has a simple API and a well-designed web interface (along with many features that fit into the MLOps category, which we'll explore in "Machine learning operations (MLOps)" on page 53). A nice benefit of commercial tools is that you don't have to host them on your own infrastructure; you just pay a monthly

13 TensorBoard works with both TensorFlow and PyTorch.

fee and someone else performs the setup and maintenance and makes sure they are secure.

Automated machine learning (AutoML)

Once you begin tracking experiments using software, it's a simple step to start running them from software, too. AutoML tools are designed to automate the process of iteratively exploring a design space. Given a dataset and some constraints, they'll design experiments to test different combinations of variables in order to try and find the best model or algorithm.

This process is called *hyperparameter optimization*,[14] and it's a highly effective way to find the best model for a particular dataset. There are many different algorithms that guide hyperparameter optimization, from a simple grid search (where every possible combination of variable is tried in turn) to named algorithms such as Hyperband that aim to intelligently control the process for maximum efficiency.

AutoML isn't a magic wand that will solve problems for you. It still takes domain expertise to frame a problem and set up the design space in the correct way. What AutoML *can* do is take the guesswork and tedium out of the ML workflow: it's a way to automate the trial and error while you focus on more productive things.

Some AutoML systems just take a design space as input and output a list of experiments to run, while others take it a step further toward the MLOps world (see the next section, "Machine learning operations (MLOps)" on page 53) by orchestrating the process of running the experiments using distributed computing techniques. A particularly complex flavor of AutoML is neural architecture search (NAS), which incorporates machine learning into the process of exploring the design space.

In terms of specifics, we recommend Ray Tune as a popular open source framework for hyperparameter tuning that is able to orchestrate the task of running hyperparameter optimization within your distributed infrastructure. Sweeps by Weights & Biases is a commercial, hosted product that helps orchestrate experiments on your own hardware.

AutoML is especially powerful for edge AI. This is because models designed for edge devices tend to be small and quick to train, which makes it easy to try a lot of different options. It's also especially important, since in edge AI we are optimizing for more than just model accuracy: we also need to find the smallest, fastest, and lowest power-consuming models that we can.

Typical AutoML tools don't account for these things, but some end-to-end edge AI platforms do.[15]

14 Or *hyperparameter tuning*.

Machine learning operations (MLOps)

The machine learning workflow has a lot of moving parts, and MLOps is the art and science of keeping track of them all. It encompasses many of the types of tools that we've covered in this chapter, from data storage systems to experiment tracking and AutoML capabilities.

As an engineer on an ML project, you're doing MLOps whether you're conscious of it or not. Even in the simplest projects, keeping track of your dataset, training scripts, and your current best model can be a challenge. In more complex projects, where every part of the workflow is constantly evolving as the result of feedback loops, keeping a handle on what is going on can be nearly impossible without effective tools.

ML Pipelines

One of the key pieces of functionality that makes up an MLOps solution is the ability to define and run ML pipelines. An ML pipeline is a scripted process that takes data, applies transformations (including signal processing or any other feature engineering), uses it to train a machine learning model, and evaluates the results. It's an extension of a data pipeline that includes the ML parts, too.

While initial experimentation often takes place in a notebook or in local scripts, it's common to start defining a formal pipeline once you want to begin automating the process of training a model. For example, pipelines make it easier to run repeated experiments to try different hyperparameters, and they are convenient if you are continually adding fresh data and want to automatically train and compare new models.

The simplest ML pipelines are implemented using scripting languages, either Python or Bash, and run on a single machine. More complex pipelines may be designed to run in distributed infrastructure, potentially with steps running in parallel to improve performance. It's common for sophisticated ML pipelines to make use of containerization (see "Containerization" on page 41): each step of the pipeline may be defined in a separate container that contains all of its required dependencies, and the containers are invoked one after the other.

An MLOps system can be built from individual components: you may choose one tool for dataset management, another tool for experiment tracking, and a different tool to store your best models. It is equally common to use comprehensive frameworks that take care of every stage in the process. It's also possible to use a mixture of comprehensive frameworks and whichever individual tools fit your specific needs.

15 Introduced in "End-to-End Platforms for Edge AI" on page 64.

MLOps is a big area that encompasses many categories of tools, including some that we've seen earlier in this chapter. The website *ml-ops.org*, a great resource for understanding MLOps, says that MLOps includes the following tasks:[16]

- Data engineering
- Version control of data, ML models, and code
- Continuous integration and continuous delivery pipelines
- Automating deployments and experiments
- Model performance assessment
- Model monitoring in production

Since edge AI is a new field, most MLOps systems are designed with the assumption that models will be "served" by web services, not deployed to edge devices. The unique nature of edge AI development involves some additional tasks, including:

- Capture of data from devices and sensors
- Digital signal processing and rule-based algorithms
- Estimation of on-device performance[17]
- Model compression and optimization
- Conversion and compilation for edge device support
- Tracking which model versions are currently in the field

A great way to think about MLOps is as a "stack": a set of software tools that work together to enable development, deployment, and maintenance of an edge AI system. The company Valohai created the idea of an MLOps stack template: a diagram that shows how all of the components of the MLOps stack fit together. Their original stack template is based on a server-side context, but Figure 5-4 shows the idea adapted to suit edge ML.

16 Listed in State of MLOps at *ml-ops.org*.

17 Including both model quality and computational performance.

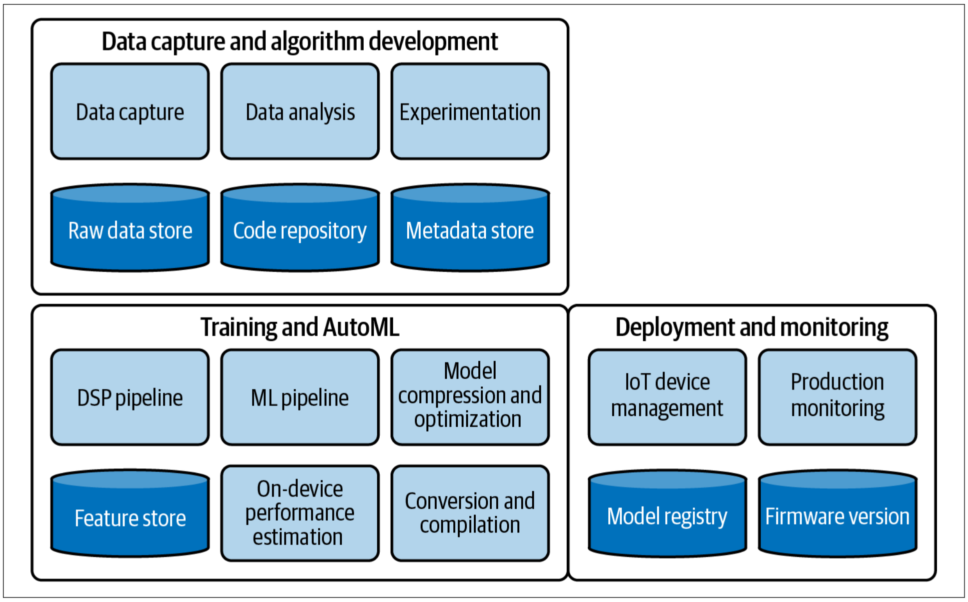


Figure 5-4. A stack template for ML at the edge; you'll need a solution for each of the boxes, and probably some others depending on your particular use case

Over the course of development, you might choose to incrementally assemble your stack from various software components. On the other hand, you may benefit from working with a comprehensive MLOps platform that is designed specifically for edge AI, as we'll encounter in "End-to-End Platforms for Edge AI" on page 64.

MLOps is a massive topic, too extensive to cover fully in a book about edge AI. If you're looking to dig deeper, we recommend the following resources—with the caveat that most MLOps content is written with server-side models, not edge AI, in mind:

- The website *ml-ops.org*.
- *Introducing MLOps*, a book by Mark Treveil et al. (O'Reilly, 2020).
- Google Cloud's introduction to MLOps, an excellent technical article.

Running Algorithms On-Device

Designing algorithms and training models requires one set of tools, while another is needed to run them efficiently on-device. These include both general-purpose C++ libraries and highly efficient implementations that are optimized for specific hardware architectures.

Math and DSP libraries

Various implementations of common mathematical operations are available, providing functionality for both DSP algorithms and deep learning ops—it would be time consuming to have to implement these fundamental algorithms from scratch. Some notable examples are:

- Fast Fourier transforms, used heavily in DSP, such as KISS FFT and FFTW.
- Matrix multiplication libraries such as gemmlowp and ruy.

Hardware devices often have features designed to improve the performance of common algorithms. These are available in hardware-specific libraries, such as the CMSIS DSP Software Library that provides optimized implementations of many popular DSP algorithms for Arm's Cortex-M and Cortex-A hardware.

There are similarly optimized implementations available for deep learning kernels, such as the CMSIS NN Software Library. Equivalents exist for many modern processor architectures, including microcontrollers and SoCs. When choosing hardware, you should investigate the availability of optimized kernels, since they can make a huge (10–100x) difference in latency.

Machine learning inference

One way to run inference on an edge device is to write a custom program that implements a specific deep learning model in code that is hand-optimized for the target architecture. However, this would be time consuming and inflexible: you wouldn't be able to reuse your code for new applications or with different hardware, and if you made any changes to your model you would have to change your entire program.

Developers have come up with various solutions to avoid this problem.[18] The most common approaches are as follows:

Interpreters
: An interpreter (or *runtime*) is a program that reads a file describing a model, including both its operations and its parameters, and then uses a set of prewritten operators to execute the model's operations one after the other. Interpreters are very flexible: using an interpreter, an identical few lines of code can be used to run any model interchangeably. The trade-off is that the process of reading and interpreting a model introduces some operational overhead beyond what is required for the model's operations. Interpreters consume additional RAM, ROM, and CPU cycles.

18 Pete Warden has an excellent blog post that outlines the technical challenges in this area.

The most widely used interpreters are both from the TensorFlow ecosystem. TensorFlow Lite was originally designed for cellphones but works on many popular SoCs, and TensorFlow Lite for Microcontrollers works well on microcontrollers and DSPs. Both of them are implemented in C++, but TensorFlow Lite provides Python and Java APIs for convenience. They both benefit from operator fusion and quantization provided by the TensorFlow Lite converter.

The kernels used by interpreters can be switched out depending on the device being targeted, so highly efficient optimized kernels can be used where available. These are readily available for several common devices and architectures.

Code generation compilers

With a code generation approach, a code-generating compiler takes a model file as input and transforms it into a program that implements it. For operator support, the program relies on a library of prewritten operators, calling them in the correct order and passing the appropriate parameters.

Code generation provides many of the same benefits as an interpreter-based approach but eliminates the majority of the overhead associated with the interpreter itself. Code generation may even make use of the wide array of prewritten operators available for interpreters: for example, Edge Impulse's EON Compiler is compatible with TensorFlow Lite for Microcontrollers kernels.

Bytecode compilers

It's possible for a compiler with knowledge of a target to directly generate the bytecode that implements a model, applying target-specific optimizations along the way. This results in a highly efficient implementation that makes use of whatever performancing-enhancing features are available on the silicon. For example, Synaptics' TENSAI Flow neural network compiler is designed to compile models for deployment to Synaptics Katana Edge AI processors.

Virtual machines

The big downside of the bytecode compiler approach is that a compiler has to be written for each device that is going to be targeted, and writing a compiler is a difficult task. To get around this problem, some compilers target a so-called *virtual machine*: an abstraction layer that sits directly above the hardware and provides instructions that map to various low-level processor capabilities.

The abstraction layer slightly reduces the efficiency, but the benefits can outweigh the drawbacks—although the virtual machine still has to be ported to new processors. This approach is used by Apache TVM, which also uses an on-device runtime that can iteratively test different implementations to find the most efficient.

Hardware description language

A newly emerging trend is the use of special compilers to generate hardware description language (HDL), the code that describes processor architectures and is used to program FPGAs and ASICs. Using these techniques, it is possible to implement a model directly in hardware, which can be extremely efficient.

CFU Playground and Tensil are both open source tools that aim to make it easier to design custom accelerators using this approach.

Alternative methods

Some accelerator chips are programmed using systems that fall outside of the normal workflow of code and compilation. For example, some chips with hardware implementations of neural network kernels provide an interface via which a model's weights are written directly to a special memory buffer, separately from any application code.

Inference and Model Optimization

The optimization of kernels for high performance on specific devices is distinct from the optimization of *models* through compression and other techniques. Model optimizations tend to require their own kernel—and sometimes hardware—support.

For example, to run a quantized model, kernels compatible with the specific level of quantization must be available. A model quantized to 8-bit integer precision requires kernels designed to support it, and the same is true of other quantization levels. In fact, specific kernels are required depending on the data type used, whether that might be `int8`, `uint8`, `int16`, or so on.

The same is true of other optimization techniques. For example, pruning results in models that have a large amount of sparsity: they have lots of zeros. By itself, this doesn't make any difference to execution time—the model has to be run using special kernels or hardware that can make use of the sparsity to reduce computation time. These kernels and hardware have yet to enter wide availability, so pruning remains of limited utility in the field.

On-device learning

As we learned in Chapter 4, the data and computational requirements of deep learning training means that on-device training remains of limited utility. Most of the time, "on-device training" means a simple approach that involves calculating the distance between embedding vectors, for example if determining whether the embeddings of two fingerprints are a match.

It's very rare for actual deep learning training to happen on an edge device. If you do have a device with the required amount of storage and compute—typically an SoC or mobile telephone— TensorFlow Lite provides some functionality.

The problem remains that it is incredibly difficult to understand whether a model trained on-device is actually performing well. On-device deep learning is best avoided unless you have an extremely good reason to require it.[19]

Federated learning remains a topic of fascination for many people, but as we learned earlier in the book it is not a particularly good fit for the vast majority of problems. In addition, the tooling around federated learning is still primitive and experimental.[20] Many people feel drawn to follow the federated learning rabbit hole and end up wasting time: the chance that a project really needs it is very slim. However, if you really feel compelled to dig deeper, TensorFlow Federated is a good resource.

Embedded Software Engineering and Electronics

Edge AI is a subfield of embedded software engineering, which is closely tied to the practical disciplines of electrical engineering and electronics. Each of these areas involves multitudes of tools and techniques—there's no way we'd have space in this book to cover them all.

Instead, we'll step through the parts that specifically matter for developing AI at the edge.

19 Pete Warden's blog post, "Why Isn't There More Training on the Edge?" does a great job of illuminating this topic.

20 Although it will invariably improve over time.

Just Getting Started

If you're prototyping your own edge AI project but don't have much embedded experience, Arduino and Arduino Pro products are a great place to start. Arduino has created an embedded development environment that is easy for beginners to use but still powerful enough for building real applications—perfect if you're an ML engineer beginning to work with edge devices, or a newcomer to both fields. The Arduino team have understood the potential of the edge AI movement since the very beginning and have contributed a lot to its growth.

Embedded hardware tools

Developing embedded software is challenging due to the nature of embedded devices. Software is harder to debug when it's running on a separate device, especially one with limited ways to display its internal state. Embedded programs must take care of everything from basic hardware integration—it's common to have to write your own drivers for hardware such as sensors—to the complex handshakes of low-level communications protocols.

As such, embedded development requires some tools that would appear unusual to other software engineers. Some of these items include:

- Device programmers, which are pieces of hardware that allow a developer to upload new programs to an embedded device. They are often device specific.
- Debug probes, hardware devices that connect to embedded processors and allow analysis of a program at runtime. They are also device specific.
- USB to UART adapters, which send and receive arbitrary data between the developer's workstation and the embedded device. They are generic.
- Multimeters, which measure voltage, current, and resistance and can be used to understand the state of an embedded circuit as it is being controlled by a program.
- Oscilloscopes, which measure signals on the device or PCB, expressed as voltage over time.

These tools are necessary in order to reach into, manipulate, and understand the states of embedded devices. For example, to test a program is running correctly you might have it toggle a specific pin on the processor when it gets to a certain point. You would then use a multimeter to measure whether the pin has been toggled. Another common way to communicate with an embedded device is via a serial (UART) cable, which can send and receive data at a relatively low frequency—but still high enough to transfer sensor data in a reasonable timeframe.

Development boards

An embedded processor on its own is just a little piece of sand, wrapped in plastic. In order to actually do anything, it requires a small constellation of other electronic components to be wired up to it. As we saw in Chapter 3, development boards (or dev boards) provide a convenient ready-to-go platform that includes an embedded processor and various inputs and outputs, often including some sensors.

The goal of a dev board is to allow embedded engineers to evaluate a particular chip for suitability for a project, and allow software development to proceed without being blocked by the hardware development process. Once a working iteration of the product's own hardware is ready, development can move there. The exception is with rapid prototyping platforms, such as Arduino Pro, which are designed for use in small-batch production designs.

Dev boards are available for most families of embedded processors. When deciding on hardware, it's a good idea to obtain a few different dev boards to experiment with. For example, you might try to run an early version of your deep learning model on a few different dev boards to understand their relative performance.

Some end-to-end platforms (see "End-to-End Platforms for Edge AI" on page 64) provide deep integration with dev boards, allowing you to capture data from their sensors or deploy and evaluate models without writing a single line of code. This can be extremely useful in development and testing.

Embedded software tools

For the purposes of edge AI, embedded software engineering generally means C++ development. This can be done in your text editor of choice, but it's also common for embedded processor vendors to provide their own integrated development environments (IDEs) that integrate neatly with their hardware and make it easier to upload and debug code.

Vendors will often provide SDKs, drivers, and libraries that can be used on their hardware to help you access various processor features—but they are not always great quality, often provided more as a proof of concept than as production-quality code.

To reduce the amount of boilerplate code you need to write, you may choose to use a real-time operating system (RTOS). An RTOS provides the functionality of a simple operating system, but it arrives as a bunch of library code that you compile alongside your own program. You can then call the RTOS APIs to do things such as controlling peripherals or performing network communication.

Embedded development frequently involves complex toolchains: programs and scripts that are supplied by the hardware vendor and are used to take code from a text file, turn it into a program, and "flash" it onto a hardware device.

The workflow generally looks like this:

1. Make changes to your source code.
2. Run a compiler (supplied by the processor vendor) and linker to transform your code into a binary.
3. Run a script to flash your code onto the embedded device.
4. Use a serial connection to communicate with the device and test your code.

When your code is running on-device, you can often use a hardware tool called a *debug probe* to inspect it from your development machine. This allows you to debug as if you were running the code locally, setting break points, examining variables, and stepping through code.

Some parts of your code will be generic C++, and you'll be able to run it on your development machine with no problem, perhaps in the form of unit tests. However, you'll also end up with plenty of code that integrates with the specific hardware APIs of your processor. It's not possible to run that on your development machine—so you can either shrug your shoulders and test it on-device only, or you can attempt to use an emulator.

Emulators and simulators

An *emulator* is a piece of software that aims to reproduce a processor virtually, running on your development machine, so that you can execute your embedded code without having to flash it to the device. It will never be a perfect representation of the real hardware—for example, it won't necessarily run at the exact same speed as the program on real hardware—but it can be close enough to be a valuable tool.

If you need to determine how fast a program will run, for example in order to estimate the latency of an AI algorithm, a cycle-accurate simulator will allow you to determine the exact number of clock cycles that will run on the real hardware. You can divide this number by the clock rate to give you a precise estimate of latency. The emulator won't actually *run* at that speed, but it will give you the information you need to create an estimate.

Simulation is the use of software to simulate an entire device, including an emulated processor plus all of the other devices it may be attached to—including sensors and communications hardware. Some simulators can even represent multiprocessor boards, or entire networks of interconnected devices.

Emulators aren't available for all processors, but Renode is a powerful emulation and simulation environment for many common processor architectures, and Arm Virtual Hardware allows you to emulate Arm processors in the cloud.

Embedded Linux

Most of the specialized embedded tools we've mentioned so far are intended for working with microcontrollers and other bare-metal devices. SoCs and edge servers are another story: with enough computing power and memory to host a full-blown operating system, SoC development is much more similar to development for personal computers and web servers. This is one of their major benefits: developers don't need quite so many specialized skills.

A typical SoC will run a distribution of Linux, with all the helpful tools and libraries that that implies. Programming can be done in nearly any language, with the same trade-offs as on any other platform: low-level languages like C++ are fast and efficient, while high-level languages like Python are flexible and easy to use.

Google provides a TensorFlow Lite runtime that is prebuilt for some popular platforms, and you'll have the benefit of being able to use Python computing libraries directly: for example, you can use SciPy's digital signal processing functions within your application.

Embedded Linux devices can even make use of containerization for deployment: embedded applications can be packaged as Linux containers, making them easy to install and use.

With SoCs, it's relatively common to use an off-the-shelf board in a production installation. Many vendors exist who design and sell SoC-based platforms designed for specific applications. For example, you can buy devices in ruggedized housing designed for industrial deployments. To deploy, you just connect whatever sensors are required and install your application.

One challenge working with SoCs is that despite the familiar Linux environment, prebuilt packages are not always available. You may have to get used to building libraries from source in order to make your applications work, which can get a little involved at times.

It's important to think about security when working with devices that have fully fledged operating systems. The embedded Linux running on an SoC needs to be locked down as tightly as any other machine on your network to avoid it becoming a vector for attacks. Insecure IoT devices are notorious for being compromised by hackers and used to attack other systems.

Automated hardware testing

Modern software engineering best practices encourage the use of continuous integration tests: every code change is put through its paces by a suite of automated tests. Creating automated tests for embedded applications can be difficult, since code that interacts with hardware can't be tested on a development machine; it can only be tested on the target device itself.

However, it's easy for an embedded device to get into a state where it is unable to run tests. For example, if the program crashes it may not be possible to restart the device without physically interacting with it. Similarly, uploading new firmware may require physical intervention.

To get around this problem, developers build automated hardware testing systems that can interact with embedded devices to facilitate easier testing. These systems are a combination of software and hardware that can do things like flash new code, power cycle devices between tests, and even provide input to I/O ports or sensors.

Automated hardware testing systems are usually custom built. They are based around a host system (perhaps an embedded device itself) that is connected to whatever continuous integration tools the team is using, as well as being connected to the devices that are intended to run the code.

If integration with a sensor needs to be tested—for example, a microphone that is supposed to be detecting a keyword—the host system might even feature a speaker that can issue keywords on demand.

End-to-End Platforms for Edge AI

In an ideal world, any team with expertise in a certain domain would be able to capture its knowledge and deploy as edge AI. People with deep insight into diverse fields like healthcare, agriculture, manufacturing, and consumer technology should be able to take what they know and use it to build amazing AI-powered products.

Unfortunately, with so many moving parts and so much to learn, it's easy to feel overwhelmed by the edge AI development process. A huge amount of the workflow is focused not on domain knowledge but on the arcane engineering skills required to build a complex product across multiple fronts, including machine learning, digital signal processing, and low-level software engineering on embedded hardware.

In the early days, only a small number of technologists—who happened, by accident, to have all of the required skills—were able to work with edge AI technology. However, over the past few years a vibrant ecosystem of tools has sprung up that is designed to reduce the barriers to entry and make it possible for people without backgrounds in machine learning or embedded systems to build fantastic new products.

End-to-end edge AI platforms are designed to assist developers with the entire process of developing an application: collecting, managing, and exploring datasets; performing feature engineering and digital signal processing; training machine learning models; optimizing algorithms for embedded hardware; generating efficient low-level code; deploying to embedded systems; and evaluating systems' performance on real-world data. This flow is summarized in Figure 5-5.

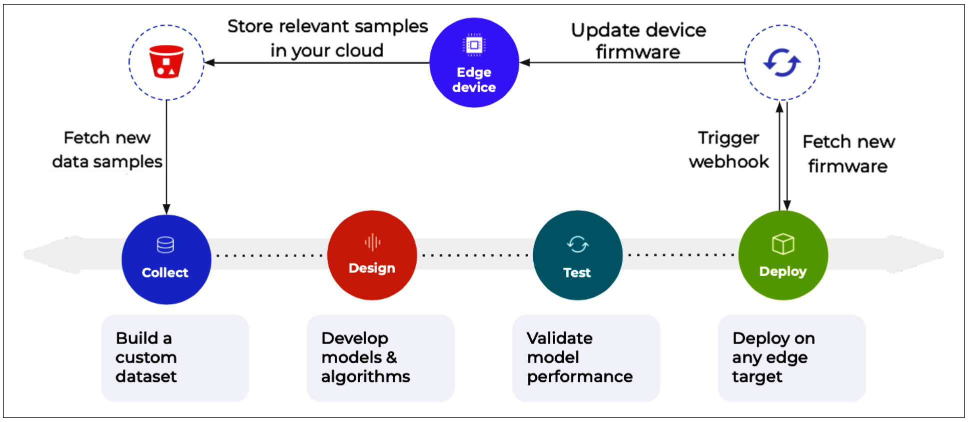


Figure 5-5. A major advantage of using an end-to-end platform is that it includes all of the components required for an iterative, data-driven feedback loop; that said, the most flexible platforms provide points for integration with external tools (Image courtesy of Edge Impulse Inc.)

End-to-end platforms are designed to apply the principles of MLOps to the specific process of creating algorithms that will run on embedded devices. As highly integrated tools, they are able to take most of the friction out of the development process: far less time is wasted in getting different parts of a toolchain to work together, and a holistic view of the entire process allows for helpful guidance that massively reduces exposure to risk.

For example, an end-to-end platform can analyze a dataset in order to help a user select the type of ML model best suited to it, or it might provide estimates of on-device performance that can help the developer choose an algorithm or embedded processor. A platform may perform AutoML with the goal of finding the best possible combination of signal processing and model that will fit on a specific device, within a maximum specified latency, or within a limited power budget.[21] A wide range of ready-to-deploy algorithms or architectures are typically available, preoptimized for various processors.

Platforms can also help teams collaborate. For instance, a cloud-based edge AI platform can act as a central repository for a team's datasets and workflow artifacts. APIs and configurable ML pipelines allow teams to automate routine tasks: so, for example, a new version of a model might be trained, tested, and deployed whenever new data is available. And visualizations and low-code user interfaces make it possible for

21 The paper by Kanav Anand et al., "Black Magic in Deep Learning: How Human Skill Impacts Network Training" (arXiv, 2020), shows that prior experience can have a strong impact on performance when tuning ML models by hand, which suggests the value of AutoML tools.

anyone on a team to contribute insight, not just those with existing data science or embedded engineering skills.

Cloud-based platforms also allow developers to benefit from distributed compute without having to administer their own systems. For example, data processing and model training might occur on powerful cloud servers that are managed by the platform, not by the user. This simplifies the process of running AutoML, where experiments may be run in parallel—as shown in Figure 5-6.

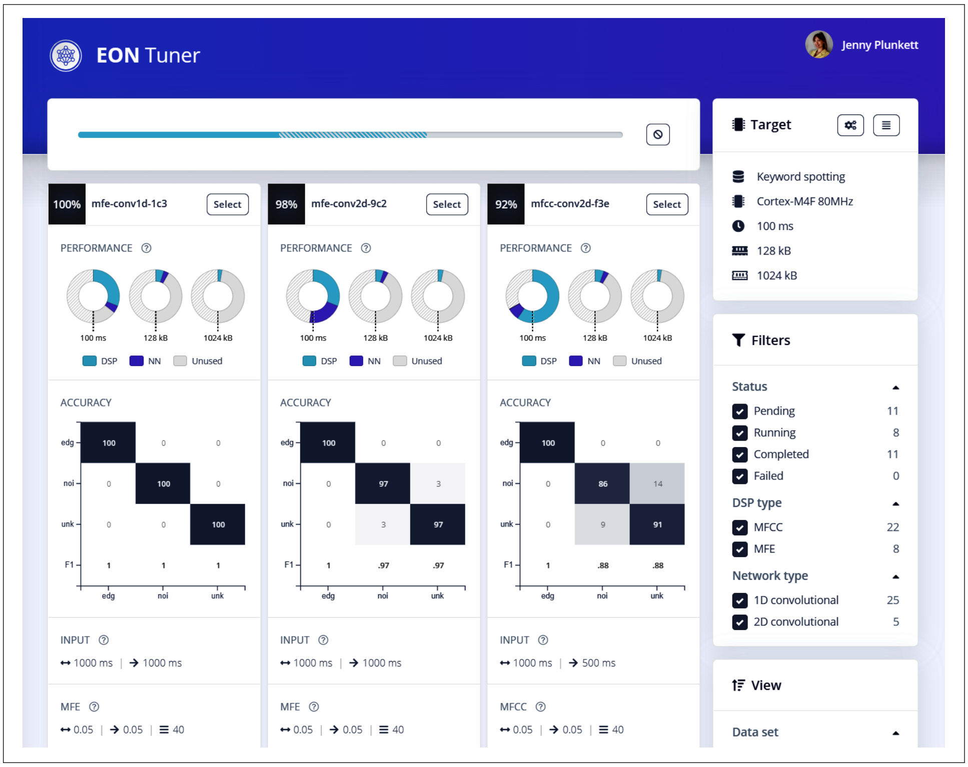


Figure 5-6. An AutoML sweep conducted using Edge Impulse's EON Tuner; with end-to-end platforms, optimization of signal processing and machine learning algorithms can occur hand in hand, incorporating estimates of on-device latency and memory use

The best end-to-end platforms have a focus on tightening all the feedback loops in the edge AI workflow. They make it possible to iterate rapidly, moving back and forth between development and testing with minimal overhead. This makes it far easier to build a successful product, since you're able to immediately detect and course-correct on any issues.

Traditionally, getting an algorithm to run on-device for the first time—using real sensor data—has been a tricky process. Some end-to-end platforms provide prebuilt firmwares for popular development boards, allowing you to capture sensor data and deploy and test models without writing any code. This enables you to close the loop between model development and real-world testing.

Another big benefit provided by end-to-end platforms is the ability to conveniently try out a variety of hardware to find the best fit. The same model can potentially be deployed in an optimized form to a multitude of microcontrollers, SoCs, and ML accelerators with a couple of clicks, allowing a development team to compare performance and determine the right choice for their application. Done by hand, this process could take weeks.

The AI ecosystem is built on top of open source tools, and good end-to-end platforms will allow you to continue using them however you want; they will integrate with industry standard technologies throughout the workflow and won't rely on vendor lock-in to keep you as a customer. You should be able to easily export your data, models, and training code, and it should be simple to create a mixed MLOps stack that incorporates parts of multiple solutions.

End-to-End or Roll Your Own?

You may be wondering which is a better choice: using an end-to-end platform or assembling your own custom set of tools from different sources. At the time of writing, it's pretty clear that the vast majority of projects will benefit from the productivity, structure, and cross-workflow integration provided by an end-to-end platform.

Even if you have strong existing skills in data science, signal processing, or embedded engineering, the process of setting up your own toolchain from scratch can be extremely tough. Depending on your target, you may find that you aren't even able to install the required tools side by side without resorting to containerization in order to isolate dependencies.

Beyond startup costs, individual tools on their own won't provide the type of immediate feedback and seamless iterative workflow required to build a successful project. You'll have to create your own automation between tools, which will result in countless scripts—and further dependencies—that need to be tracked, maintained, and scaled to serve your whole team.

In addition, end-to-end platforms are designed to provide the appropriate guidance to fill in your blind spots. Almost nobody has all of the required skills to build an edge AI application on their own. For example, someone with deep domain expertise is unlikely to also have a strong intuition about which deep learning model architectures are most efficient on a specific model of embedded processor.

The teams developing end-to-end platforms for edge AI have spent years building and improving them—so while it's certainly possible for a sufficiently large enterprise organization to build their own internal platform, it would be a multimillion dollar investment in time and resources. It's very unlikely that the cost-benefit analysis would make sense, which is why some of the world's biggest, most sophisticated organizations—from government organizations like NASA to industrial giants like Bosch—are users of end-to-end platforms.

A major, valid concern is around flexibility and openness. If a company decides to use an end-to-end platform, what happens if they need to use techniques—for example, specific algorithms—that aren't available in the platform?

Fortunately, the best platforms already account for this and provide easy integration points for interoperability. New algorithms, data stores, deployment targets, and evaluation methods can be connected seamlessly, and APIs allow end-to-end platforms to be interwoven with other tools, including existing internal systems and alternative open source AI tooling.

Another concern is cost. End-to-end platforms are typically supported by an enterprise subscription fee that includes technical support and compute time, with many products also offering a free tier for individual projects. One platform, Edge Impulse, has a large, active community of free users who provide support to one another, including sharing example projects for inspiration and technical guidance.

If you don't have a budget, it's absolutely possible to use a product's free tier to build a successful project. If you do have a budget, platforms are typically quite affordable—especially when compared to the time cost of setting up and managing your own environment. The subscription cost typically buys you the type of heavy duty functionality that is required when dealing with big enterprise datasets and large teams.

Given the absurd complexity of the edge AI toolchain, it's very easy to recommend end-to-end edge AI platforms as the best starting point for the vast majority of projects. In the uncommon case that you require a feature that isn't supported, high-quality platforms make it simple to integrate with external tools—so you can use the platform as a foundation and extend it however you need.

At this point, it's worth addressing the fact that the authors of this book, Daniel and Jenny, are part of the team that has designed and developed Edge Impulse, an extremely popular end-to-end edge AI development platform. It's always important to take recommendations with a pinch of salt when the people doing the recommendation have a vested interest! Since we work on an end-to-end tool, how can we be expected to recommend anything else?

Hopefully, the history of this book provides some reassurance. One of the authors, Dan, was coauthor of *TinyML*—a book that helped introduce the field of embedded machine learning to a wider audience. *TinyML* introduces the process of building edge AI software using open source tools. At about 500 pages, it's not a short guide—

but it only covers the absolute basics, and it relies on its readers learning both Python and C++. Working directly with low-level tools is not a productive way to go.

Writing *TinyML* inspired both of its authors to try and make life easier for developers. Dan went on to join Edge Impulse as its founding engineer, inspired by a demo where the company's CEO built and deployed a deep learning model for activity classification live in under ten minutes. The other coauthor of *TinyML*, Pete Warden, is working to simplify machine learning deployment by integrating sensors and ML as closely as possible.

Machine Learning Sensors

Building an effective edge AI product requires a lot of difficult work and specialized knowledge. Another concept that could help make the task easier is the idea of *machine learning sensors*. Proposed in a 2022 paper[22] by a team led by Pete Warden, ML sensors are designed to be as simple to work with as ordinary sensors—but to include a dash of intelligence.

For example, a "person detector" ML sensor might be provided as a single chip that includes an image sensor, a processor, and a deep learning model that can identify human beings from images. As an interface, the ML sensor could expose a single digital pin that toggles high when a person is detected or low when there is nobody there.

Integrating with an ML sensor would be far easier than training and incorporating a machine learning model (along with all of the required dependencies) into an embedded application, making it trivial to add intelligence to devices. The trade-off is reduced flexibility—although if required, models could be customized via integration with end-to-end platforms.

At the time of writing, Pete's company Useful Sensors is selling the Person Sensor, a small, low-power device that can detect and locate human faces. You can find more general information at ML Sensors.

Summary

We've now encountered the people, skills, and tools that are prerequisites to successful edge AI projects. From the next chapter onward, we'll be taking a journey through the iterative development workflow that real-world teams use to build applications.

22 Pete Warden et al., "Machine Learning Sensors", arXiv, 2022.

About the Authors

Daniel Situnayake is head of machine learning at Edge Impulse, where he leads embedded machine learning research and development. He is coauthor of the O'Reilly book *TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-Low Power Microcontrollers*, the standard textbook on embedded machine learning, and has delivered guest lectures at Harvard, UC Berkeley, and UNIFEI. Dan previously worked on TensorFlow Lite at Google, and cofounded Tiny Farms, the first US company using automation to produce insect protein at industrial scale. He began his career lecturing in automatic identification and data capture at Birmingham City University.

Jenny Plunkett is a senior developer relations engineer at Edge Impulse, where she is a technical speaker, developer evangelist, and technical content creator. In addition to maintaining the Edge Impulse documentation, she has also created developer-facing resources for Arm Mbed OS and Pelion IoT. She has presented workshops and tech talks for major tech conferences such as the Grace Hopper Celebration, Edge AI Summit, Embedded Vision Summit, and more. Jenny previously worked as a software engineer and IoT consultant for Arm Mbed and Pelion. She graduated with a BS in electrical engineering from The University of Texas at Austin.

Colophon

The animal on the cover of *AI at the Edge* is a Siberian ibex (*Capra sibirica*). They can be found across Asia in places like China, Mongolia, Pakistan, and Kazakhstan. Siberian ibexes are essentially a large species of wild goat. The color of their fur ranges from dark brown to light tan with an occasional reddish tint. Males have large, black, ringed horns while females have smaller gray horns. Both sexes have beards. Their coat lightens in color during the winter and darkens during the summer. They tend to travel in single-sex herds of 5 to 30 animals.

The ideal habitat for Siberian ibexes is above the tree line on steep slopes and rocky scree. They can be found as low as 2,300 feet in semiarid deserts. Their diet consists mainly of grasses and herbs found in scrublands and grasslands.

Because Siberian ibexes are found in abundance in their natural habitat, they are considered a species of Least Concern even though their population is decreasing. Their biggest threat is hunting for food and poaching for sport. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover illustration is by Karen Montgomery, based on a black-and-white engraving from *The Natural History of Animals*. The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.