O'REILLY®

AI at the Edge

Solving Real World Problems with Embedded Machine Learning

Early Release

RAW & UNEDITED

Daniel Situnayake
& Jenny Plunkett

AI at the Edge

Solving Real World Problems with Embedded Machine Learning

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Daniel Situnayake and Jenny Plunkett

Beijing • Boston • Farnham • Sebastopol • Tokyo

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.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://oreilly.com). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Rebecca Novack

Development Editor: Angela Rufino

Production Editor: Elizabeth Faerm

Copyeditor: TO COME

Proofreader: TO COME

Indexer: TO COME

Interior Designer: David Futato

Cover Designer: Karen Montgomery

Illustrator: Kate Dullea

December 2022: First Edition

Revision History for the Early Release

- 2022-01-21: First Release
- 2022-05-10: Second Release
- 2022-08-17: Third Release

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

978-1-098-12014-6

[TO COME]

Chapter 1. A Brief Introduction to Edge AI

> **A NOTE FOR EARLY RELEASE READERS**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 1st chapter of the final book. Please note that the GitHub repo will be made active later on.
>
> If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at dan@situnayake.com and jplunkett@utexas.edu.

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 any 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?

Embedded ML, edge AI, and tiny machine learning

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. 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.

< Images of a range of embedded systems >

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—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.

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 work. 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.

< photo of a mainframe >

Eventually, however, computers were connected to terminals 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.

< diagram of mainframe with terminals >

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, by the end of 2021 it is estimated that there will be 12.3 billion assorted items connected to the Internet, creating and consuming data. This vast network of objects is called the Internet of Things, 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.

< picture of the first ever IoT device, a coke machine >

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**.

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 they can send it up to a central server. Edge devices have access to data that nobody else does.

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 (A.I.)

Phew! This is a big one. Artificial intelligence 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, hyper-intelligent 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, a simple organism with no central nervous system that is capable of solving a maze.

< photo of slime mold solving maze >

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

> *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 limited 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 attempts have shown. But creating something that operates at slime mold level can be much easier. For example, preventing a driver from crashing is 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 = 10 # 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*.[1]

There are two points we are trying to make here. The first is that intelligence is quite hard to define, and some rather simple problems require 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.

Machine learning (ML)

At its heart, machine learning is a pretty simple concept. It's a way to discover rules about 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 wrist band 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.

< diagram of accelerometer readings>

To help your athletes, you want to keep an automatic log of the activities that 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 wrist band. To collect some data, you give prototype wrist bands to a dozen athletes and have them perform specific activities—either swimming, running, or doing nothing—while the wrist bands log data.

< diagram of readings for different activities>

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.

With an ML approach, you feed all of your athletes' data into a special 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 new input—a sample of accelerometer data from a particular window in time—and tell you which activity the athlete was performing. During training, the model has learned the characteristics that distinguish running from swimming. You can then use the model in your fitness tracker, 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 wrist band 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 dumb 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 remote location for processing, what if to 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 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.

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!

Embedded machine learning (Embedded ML) and tiny machine learning (TinyML)

Embedded ML is the art and science of running machine learning models on embedded systems. TinyML is the concept of doing this on the most constrained embedded hardware available—think microcontrollers, digital signals processors, and 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 (to store the model) and RAM (to handle the intermediate results generated during inference).

They are often also limited in terms of clock cycles. Since many types of machine learning models are quite processor 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 (DSP)

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 is the idea of using algorithms to manipulate these streams of data. When paired with embedded machine learning, we often use digital signal processing 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
- Extracting the most useful information from a signal, reducing its bit rate

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. 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?

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 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 this topic 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.

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.

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.

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 that 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 on-board 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 solve 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 labelling 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 which would previously be 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 photos 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.

Don’t underestimate the impact of economics. By reducing the cost of long term support, edge AI enables a whole galaxy of use cases.

Reliability

Simplicity and reliability go hand-in-hand. Adding connectivity to a product makes it vastly more complex—not only due to the extra hardware

and firmware required, but due to the back-end infrastructure required to support a fleet of devices.

Systems controlled by on-device AI are inherently more reliable than those which 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 risk associated with technologies that make up your distributed computing stack.

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.

Privacy

Over the past few years our society has become used to the idea that there's a tradeoff between convenience and privacy. 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: [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 in to your house.]. 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 tradeoff—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.

Edge AI provides an alternative. Rather than streaming live video and audio to a remote server, a security camera could use some on-board intelligence to identify that an intruder is present when the owners are out at work. It could then send a simple notification and a still image directly to the owners. 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.

The ability of edge AI to enable true 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, it's often much easier to build a product that *avoids* collecting data.

The moral case for edge AI

One of the most exciting things about edge AI is the opportunity it presents to help us build a fairer, healthier, and more equitable world. Technologists in areas like conservation, healthcare, and education are already using edge AI tools to make a big impact. Here are some examples we're personally excited about:

- Smart Parks are 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, 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 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.

Since reliable connectivity is expensive and not universally available, many current generation smart technologies only benefit people living in wealthier countries. By removing the need for a reliable Internet connection, edge AI reduces the cost of life saving technologies and allows the entire planet to benefit from advances in everything from medical screening to precision agriculture.

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 back-end 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 for the Developing World", an excellent talk given by Pete Warden at the TinyML Kenya meetup.

In addition to making an impact in communities around the world, edge AI 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.

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 tremendous opportunity to make the world a better place.

> **NOTE**
>
> 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 Daniel Situnayake at dan@edgeimpulse.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 labelled 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 of computation and memory, and it often requires a labelled dataset. Both 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 labelled—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 labelled data.

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 take action if the state becomes abnormal. We’ll discuss both of these ideas later!

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 is very messy. It arrives at a high frequency—potentially many thousands of times per second. An embedded device running an edge AI application has a limited duty cycle 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.

For this reason, digital signal processing is 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. It’s a whole different world.

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 mid-range microcontroller there may only be a hundred kilobytes or so available to store a model's weights, and some devices have far smaller amounts of ROM. Since larger models take more time to execute, the slow clock speeds of devices can also push developers towards 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. The next chapter will talk about these challenges 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.

The feedback loops are 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 on-board 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 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's probably an ideal system for any particular use case. The hard part is choosing one! We'll cover this challenge in a later chapter.

"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 "https://aallan.medium.com/capable-computing-50867847a8d8[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.

Summary

In this chapter we've explored the terminology that defines edge AI, learned a handy tool for reasoning about its benefits, explored the moral case for moving compute to the edge, and outlined the factors that make edge AI different.

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

---

1 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!

Chapter 2. Edge AI in the Real World

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at dan@situnayake.com and jplunkett@utexas.edu.

We now have a basic understanding of what edge AI means and what makes it—in theory—a useful set of technologies. In this next chapter we'll see what that theory looks like when it makes contact with the real world. We'll start by examining some actual products that are out in the field today. After that, we'll explore the top application areas for edge AI products. Finally, we'll learn more about the ethical considerations required to make any product a success.

Common use cases for edge AI

As we learned in the previous chapter, edge AI is especially valuable for devices with an abundance of sensor data but a lack of compute or connectivity. Luckily for us, these conditions can be found nearly everywhere.

In modern cities it can feel like we're never very far from a power socket or a wireless access point. But even when high bandwidth network connections and reliable power are available, there are huge advantages to

limiting the communications and power consumption of devices. As we saw in "To understand the benefits of edge AI, just BLERP", the pursuit of desirable features such as portability, reliability, privacy, and cost can drive product development towards devices that are designed to minimize the use amount of connectivity and energy usage.

Despite our seemingly global Internet, there are plenty of places on the planet that are limited in connectivity or power. At the time of writing, 50% of the Earth's land is relatively untouched by human development. Only a small percentage of the planet's surface has cellular or wireless coverage, and billions of people do not have reliable access to power. But beyond the obviously remote regions, there are plenty of hidden corners in our most built up regions that fall into this category. In our modern industrial supply chains, there are places where it's impractical to provide hard-wired DC power for embedded devices—making efficient, battery powered devices the perfect fit.

<image of peel and stick sensor on a cardboard box, or something similar>

At the same time, sensors are becoming cheaper, more sophisticated, and less power hungry. Often, even simple embedded devices ship with highly capable sensors that remain underutilized due to the challenges in getting the data off the system for remote processing. For example, imagine a basic fitness wearable that uses an accelerometer to count steps. Even this simple device might be equipped with a sensitive multi-axis accelerometer that has a very high sample rate, capable of recording the most subtle movements. Unless the device's software is equipped to interpret this data, most of it will be thrown away.

Green and brown field projects

Conditions such as those discussed above produce nearly endless opportunities for deploying edge AI. In practical terms, it can be helpful to group these opportunities into two categories: *green field* and *brown field*. These terms are borrowed from urban planning. A green field project is one that takes place on a site that has yet to be developed and is still a grassy,

green field. A brown field project takes place on a site that has already been developed and may have some existing legacy infrastructure.

In the edge AI world, green field projects are ones where the hardware and software are designed together from the ground up. Since there's no existing hardware, green field projects can make use of the latest and greatest innovations in compute and sensing—which we'll learn more about later in this chapter. The developers have more freedom to design the ideal solution for the use case they are trying to target. For instance, modern cellphones are designed to include dedicated low-power digital signal processing hardware so that they can continually listen out for a wakeword (such as "OK, Google", or "Hey, Siri") without draining the battery. The hardware is chosen with the specific wakeword detection algorithm in mind.

In contrast, brown field edge AI projects begin with existing hardware that was originally designed for a different purpose. Developers must work within the constraints of the existing hardware to bring AI capabilities to a product. This reduces developers' freedom, but it avoids the major costs and risks associated with designing new hardware. For example, a developer could add wakeword detection to a bluetooth audio headset that is already on the market by making use of spare cycles in the device's existing embedded processor. This new functionality could even be added to existing devices with a firmware update.

Green field projects are exciting because they allow us to push the limits of what is possible by pairing the latest edge AI hardware and algorithms. On the other hand, brown field projects allow us to bring new capabilities to existing hardware, delighting customers and making the most of existing designs.

Real world products

The best way to understand a technology is to see how it's applied in the real world. We're still in the early days of edge AI, but it's already being used across a huge range of applications and industries. Here are three brief

overviews of real world systems that have been developed using edge AI. Perhaps your own work will be featured in a future edition of this book!

Preventing forest fires using power line fault detection

Power lines transmit electricity across vast swathes of wilderness, including Europe's ancient forests. Equipment failure can potentially ignite vegetation and cause wildfires. With thousands of miles of towers and power lines, often in very remote areas, electrical equipment can be difficult to monitor.

Izoelektro's RAM-1 device uses edge AI to help solve this problem (Figure 2-1). A package of sensors monitor conditions at each electrical pylon, including temperature, inclination, and voltage, and use a deep learning classification model ("Algorithms for edge AI") to identify when a fault may be developing. Technicians can visit the pylon and make a repair before there is any danger of fire. The device has a rugged construction designed to withstand extreme weather conditions over many years of service.

Figure 2-1. Izoelektro's RAM-1 device

There are two main factors that make this a perfect application for edge AI. The first is the lack of connectivity in wilderness locations. It would be prohibitively expensive to transmit raw sensor data from thousands of remotely located pylons in real time. Instead, the elegant solution is to interpret the sensor data at the source and transmit *only when a fault is*

predicted. The device is able to understand which data is crucial enough to require immediate attention, sending less important information in periodic batch transmissions.

This selective communication helps with the second, slightly unintuitive factor. Although the RAM-1 is mounted on an electricity pylon, it actually makes use of battery power. This ensures it keeps working even if there's a fault in the power lines, and it reduces the cost and complexity of installation. Since radio transmission uses a lot of energy, the RAM-1's ability to avoid unnecessary transmission helps it preserve battery life. In fact, with the help of edge AI, its battery can last for twenty years.

Here's how the RAM-1 fits the BLERP model:

- Bandwidth: Connectivity is limited in remote locations where RAM-1 is deployed.
- Latency: It's critical to identify failures as soon as they happen.
- Economics: Avoiding unnecessary communication saves money and means the device can run on battery power, which reduces the cost of installation.
- Reliability: The ability to run on battery power improves reliability.
- Privacy: Remote monitoring of pylons reduces the need to send workers on to private land.

Protecting first responders with intelligent wearables

The nature of their work means that firefighters are often exposed to high temperatures, and the extreme heat conditions can have a major impact on their long term health. In fact, according to FEMA the leading cause of firefighter line-of-duty deaths is sudden cardiac events.

SlateSafety's BioTrac Band is a wearable device designed for workers, like firefighers, who are exposed to extreme conditions (Figure 2-2). It provides an early warning system that can help alert individuals and teams to conditions that may result in heat strain and overexertion. The BioTrac Band uses an embedded machine learning model alongside heuristic

algorithms to analyze data from multiple sensors—including signals from the wearer's body—and predict when an injury is about to happen. This intelligence made the device one of Time Magazine's 100 best inventions of 2021.


FIRE DEPARTMENT

Figure 2-2. SlateSafety's BioTrac Band

The extreme environments that the BioTrac Band is deployed in make it a fantastic use case for edge AI. By analyzing data on-device, the band can continue to function and warn its wearer even when connectivity becomes limited or unavailable during the course of an emergency. In addition, the ability to interpret data on-device means that unnecessary transmission of data can be avoided—which saves energy and improves battery life, while allowing the size and weight of the device to be kept to a minimum.

Here's how the BioTrac Band fits the BLERP model:

- Bandwidth: Connectivity is limited in extreme environments where firefighters work.
- Latency: Health issues are time-critical and must be identified immediately.
- Economics: Streaming raw data from sensors would require expensive high bandwidth connections.
- Reliability: The device can continue to warn firefighters of potential risks even if connectivity drops, and it can function for a long time on a small battery.
- Privacy: Raw biosignal data can be kept on-device, with only critical information being transmitted.

Understanding elephant behavior with smart collars

With increased pressure on their natural habitat, wild elephants are increasingly coming into contact with human beings. These interactions often end badly for the animals, with poaching or conflict with farmers and settlers frequently leading to injury and death. To reduce the likelihood of these events, conservation workers and scientists are trying to learn more about elephant behavior and the types of conditions that lead to dangerous encounters.

ElephantEdge is an open source project to create a tracking collar designed to help researchers understand elephant behavior (Figure 2-3). The collar,

fit around the neck of an elephant, can provide insight into the animal's location, health, and activities using a combination of embedded sensors and machine learning models. This data can be used for scientific study—and it can also be used to alert humans to the presence of animals so that conflict can be avoided.

Figure 2-3. The OpenCollar Edge tracking collar being fitted to a sedated elephant

Since the device is attached to a wild elephant, replacing the battery is a difficult task! Edge AI technology helps by minimizing the amount of energy consumed by the device. Instead of transmitting large amounts of raw sensor data, the machine learning-equipped collar is able to transmit high-level information about the animal's activities: for example, whether it is walking, eating, drinking, or engaging in other behaviors. The models that allow it to do this were prototyped by a community of citizen scientists working with public datasets.

These low bandwidth requirements mean the collar can take advantage of an extremely low power wireless communication technology named LoRa.

The collar is able to communicate with LoRa-equipped satellites that pass over once per day, sending a summary of the animal's activities since the last transmission. This means that the system can work reliably even in places with no traditional connectivity, but the battery can last for an estimated five years.

Here's how the OpenCollar Edge fits the BLERP model:

- Bandwidth: Connectivity is limited in elephant habitat; on-device analysis enables use of low energy wireless technologies.
- Latency: Even though the device only transmits once per day, this is very frequent compared to traditional tracking collars that require manual downloading.
- Economics: The device saves money by replacing traditional methods for monitoring elephants, which are labor intensive.
- Reliability: Infrequent transmission means the battery can last for years, and makes satellite technology economically viable, increasing range.
- Privacy: Tracking of elephants directly is less intrusive to local people than setting up cameras to monitor animal activity.

These three use cases represent a tiny sample of what is possible. In the next section, we'll talk through some general high level categories of applications.

Types of applications

There are opportunities to deploy edge AI across every part of our modern world, from heavy industry to healthcare, agriculture to art. The possibilities are nearly endless! To make things easier to discuss, the roles that edge AI technologies play within these applications can be grouped into a few high level categories:

- Keeping track of objects

- Understanding and controlling systems
- Understanding people and living things
- Generating and transforming signals

Let’s walk through each of these categories and understand where edge AI fits.

Keeping track of objects

From vast container ships to individual grains of rice, our civilization depends on the movement of objects from one place to another. This might occur in the controlled conditions of a warehouse, where items are moved carefully from storage to shipment. It may also occur under the most extreme conditions, like the motion of weather systems across the face of the planet.

Tracking and interpreting the state of objects, both man-made and natural, is a key application area for edge AI. Intelligent sensors can help encode the state of the physical world in a form computers can understand, allowing us to do a better job of coordinating our activity.

Table 2-1 discusses edge AI use cases that involve keeping track of objects:

Table 2-1. Edge AI use cases for keeping track of objects

Use case	Key sensors
Monitoring shipments using smart packaging to detect damage during transit.	Accelerometer, vibration, GPS, temperature, humidity
Counting products on store shelves using embedded cameras, so items can be restocked before they run out.	Vision
Analyzing the movement of plastic waste in the ocean so it can be cleaned up.	Vision
Identifying and tracking obstacles at sea to help ships avoid collisions.	Radar

DEEP DIVE: MONITORING SHIPMENTS USING SMART PACKAGING

It’s common for manufactured products to travel thousands of miles on their way to a customer—and they don’t always make it in one piece. Damage during shipping costs businesses money, but when a shipment arrives damaged after a long voyage it isn’t always easy to figure out what happened.

With edge AI, a logistics company could attach a device to high-value shipments that can recognize when an item is at risk of damage. For example, if equipped with an accelerometer, the device could use a machine learning model to distinguish between normal bumps and jolts and specific types of rough handling that might lead to damage. Any rough handling event could be logged, along with a timestamp and a GPS location.

The logs could be uploaded periodically, whenever the device is able to get a wireless connection. Upon arrival, if there is any damage, the company could analyze the logs to discover the time and place where the damage happened—allowing them to find and fix the cause of the issue.

What makes this a good use case for edge AI? Let’s think about it in terms of BLERP:

- Bandwidth: To detect sudden bumps, the accelerometer data would have to be quite high frequency. This makes it difficult to transmit from a low-power wireless radio, which are generally low bandwidth. By processing data on-device we can massively lower the bandwidth requirements.
- Latency: Not a major consideration for this use case.
- Economics: It’s expensive to transmit data wirelessly, especially since the device could be anywhere in the world. Using edge AI helps conserve data and lower costs.

- Reliability: Shipments in transit are unlikely to have reliable connectivity, so it’s important that the device can keep logging even when out of range. If we don’t have to store raw data, we can log all the interesting events in a small amount of memory.
- Privacy: Not a major consideration for this use case.

Key benefits for object tracking

Object tracking tends to make use of the connectivity and cost related benefits of edge AI. The world has many objects, and they’re not always in convenient places. Cheap edge AI sensors making use of low-cost, opportunistic connectivity can provide high resolution visibility into gaps in the supply chain that would otherwise be too expensive to monitor.

Of course, the exact benefits of deploying edge AI vary from project to project. For example, a system using cameras to monitor stock on store shelves might use edge AI for privacy. If Internet-connected cameras were used to monitor store shelves, employees might feel like they are under constant scrutiny by HQ. But a stock tracking system that works offline, solely for the benefit of the store’s team, could be a welcome aid.

Understanding and controlling systems

Our modern world is built on millions of complex, interconnected systems—everything from production lines to transportation networks, climate control to smart home appliances. The wellbeing of our economies is tied intimately to these systems. A breakdown in production can cost vast amounts of time and money, and improvements in efficiency can lead to huge savings in costs, labor, and emissions.

The monitoring, control, and maintenance of complex systems is a vast opportunity for edge AI. The ability to make rapid, reliable decisions at the edge can improve the responsiveness and resilience of our systems, and fine-grained insights into system state can help us better plan for the future.

Some edge AI use cases that involve understanding and controlling systems can be found in Table 2-2:

Table 2-2. Edge AI use cases for understanding and controlling systems

Use case	Key sensors
Monitoring an oil rig for signs that it needs maintenance, avoiding downtime and reducing leaks and spillage.	Accelerometer, vibration, load, temperature, audio, vision, and more
Autonomously driving a combine harvester, helping a farmer quickly harvest their crops.	Vision, GPS
Understanding and shaping traffic flow on a busy highway, using variable speed limits to keep cars moving.	Vision, magnetometer
Identifying faulty items on a production line using computer vision, improving quality control and quickly recognizing issues.	Vision
Cleaning a carpet using a robot vacuum, saving time for the owner of the home.	Vision, proximity, touch, voltage
Fetching items in a warehouse using robots, reducing labor costs and workplace health risks.	Vision, proximity, touch, light
Detecting intrusions in computer networks using traffic analysis, automatically responding to security threats.	Network logs

This is a truly enormous category of applications, including many of the things we associate with our vision of "the future": self-driving vehicles, industrial robots, and smart factories. What they have in common is the use of edge AI to monitor the state of a complex system and to provide feedback and control when change is required.

Key benefits for understanding and controlling systems

A broad category, the automated monitoring and control of systems makes use of most of the benefits of edge AI. Economics and reliability are particularly important to many business use cases, and the benefits of low-

bandwidth, low-latency solutions provide further justification where otherwise a server-side system might be used.

DEEP DIVE: PREDICTIVE MAINTENANCE AT AN OIL RIG

If a piece of industrial equipment suddenly fails, the resulting downtime and disruption to processes can cost tremendous amounts of money. In some cases, it may also pose a threat to the health of human beings and the environment. Predictive maintenance is the art of identifying when a system is starting fail—so that steps can be taken before it does.

An oil well is an incredibly complicated piece of machinery that operates under extreme conditions. Due to its precarious position in the middle of the ocean, faults can result in more than just costly downtime—the lives of the rig's crew are at stake, and oil spills can contaminate the ocean environment.

Using edge AI, sensor-equipped devices can be deployed to monitor key components of an oil rig, measuring factors such as vibration, temperature, and noise. They can learn the "normal" state of each part of the system, building a model of what nominal operation looks like. If conditions start to deviate, they can alert a maintenance team to investigate further. Particularly sophisticated predictive maintenance systems might even have some control over the equipment, automatically halting operation if a dangerous situation is detected.

To understand why this is a good fit for edge AI we can use the BLERP model:

- Bandwidth: Most oil rigs rely on satellite for connectivity, making it challenging to stream large amounts of sensor data from thousands of rig components into the cloud. Further, there are places within a drilling operation that have **very** limited connectivity—for example, a drilling bit might be miles beneath the ocean floor! On-device predictive maintenance can turn a vast stream of noisy data into a lightweight sequence of events that are easy to transmit.
- Latency: It's expensive to pay expert human beings to travel to an oil rig and inspect equipment. This means that inspection happens

periodically, limiting how quickly a problem can be identified. With constant monitoring from an edge AI system, issues can be identified and addressed as soon as they present.

- Economics: Predictive maintenance can save vast amounts of money that might otherwise be lost to downtime. In addition, monitoring via AI-equipped smart sensors is a lot cheaper than paying humans to perform the dangerous work of inspecting heavy machinery.
- Reliability: In the extreme off-shore environment, you can't always depend on reliable transportation or communications. Using edge AI, insight into equipment health can continue even when usual operations are disrupted. Privacy: Not a major factor in this application.

Understanding people and living things

The biological world is complex, messy, and can change quickly. There's huge value in being able to understand and react to it in real-time. This category includes human-oriented technologies, like fitness tracking watches and educational toys, as well as systems for monitoring nature, agriculture, and the microscopic world.

These applications help bridge the gap between biology and technology, allowing our rigid computer systems to interface with the dynamic and flexible world of life on planet Earth. As our understanding of biology improves, this field will continue to grow.

Table 2-3 and Table 2-4 show examples of edge AI use cases that connect computers with people and living things:

Table 2-3. Edge AI use cases involving people

Use case	Key sensors
Alerting workers in a dangerous environment when they are missing protective equipment.	Vision
Understanding human gestures to control a video game.	Vision, accelerometer, radar
Identifying when an intensive care patient's health is deteriorating and notifying a medical attendant.	Biosignals, medical equipment
Recognizing when a thief has broken into a home and alerting the authorities.	Vision, audio, accelerometer, magnetic sensors
Categorizing physical activities using sensors in a smart watch.	Accelerometer, GPS, heart rate
Recognizing a user's voice commands and controlling an appliance.	Audio
Counting the number of people who are waiting at a bus stop.	Vision
Warning a driver when they are falling asleep at the wheel of a car.	Vision

Table 2-4. Edge AI use cases involving living things

Use case	Key sensors
Informing researchers when wildlife of interest is spotted by a remote trail camera.	Vision, audio
Cooking food to perfection by monitoring and controlling a sensor-equipped kitchen appliance.	Vision, temperature, volatile organic compounds
Diagnosing crop diseases using camera in a remote rural location with no cellphone coverage.	Vision, volatile organic compounds
Recognizing sounds made by marine mammals to track their movements and understand their behavior.	Acoustic
Warning villagers of an approaching elephant so they can avoid human-animal conflict.	Thermal imaging, vision

Key benefits for understanding people and living things

Another large area, applications involving people and living things make use of every aspect of the BLERP model. That said, this is a category where privacy can be especially important. There are many applications that are technically feasible using server-side AI, but only become socially acceptable when done on-device.

The most widespread example of this is digital personal assistants, such as Apple's Siri or Google's Google Assistant. As discussed earlier, personal assistants work by using on-device models to constantly listen for wakewords. Only after the wakeword is detected is any audio streamed to the cloud. Without the on-device component, the assistant would have to constantly stream audio to the service provider. This would be incompatible with most people's expectations around privacy.

By moving functionality onto devices, and avoiding the transmission of data, we unlock massive possibilities—especially in vision, which until recently required large models that could only be run in the cloud.

DEEP DIVE: SPOTTING RARE WILDLIFE WITH TRAIL CAMERAS

A trail camera, or camera trap, is a special type of camera designed for monitoring wildlife. It has a tough, weatherproof housing, a high capacity battery, and a motion sensor. Installed with a view of a trail, it automatically snaps photos whenever it detects motion.

Researchers who are monitoring specific species install camera traps in remote locations and leave them for months at a time. When they return, they download the photos from the camera and use them to better understand their target species—for example, they may try to estimate how many individuals exist.

There are a some significant problems with camera traps that cost a lot of time and money:

- Most of the photos captured do not feature the target species—instead, the capture was triggered by non-target species or by random motion in the field of view.
- Due to the high number of false positives, it would not be helpful to send notifications of captures via a network connection.
- Instead, researchers must travel out to the remote location to collect the saved photos.
- This is extremely expensive, and can result in missing data if the memory card gets full, or unnecessary trips if nothing interesting has been photographed.
- Researchers must then trawl through thousands of useless photographs to find the few that matter.

Using edge AI, camera traps can be equipped with deep learning vision models trained to identify target species and reject any images that do not contain them. This means that researchers no longer have to worry about filling up memory cards with useless images. Even better, it means that cameras can potentially be equipped with low-power or

cellular radio transmitters that allow them to report back on animal sightings without anyone having to visit the field. This can massively reduce the cost of a study and increase the amount of scientific work that can be done.

The BLERP model can tell us exactly why this is a great application for edge AI:

- Bandwidth: Camera traps are often deployed in remote areas with low connectivity—perhaps with expensive, low-bandwidth satellite as the only option. With edge AI, the number of photos taken can be reduced enough to make it possible to transmit them all.
- Latency: Without edge AI, the latency involved with sending a researcher to collect photos from camera traps could be measured in months! With edge AI and a low-power radio connection, it's possible to analyze photos immediately and obtain useful information without having to wait.
- Economics: Avoiding trips out into the field saves large amounts of money; so does avoiding unnecessary use of expensive satellite radio.
- Reliability: If useless photos can be discarded, the memory card will take longer to fill up.
- Privacy: An edge AI camera can discard photos of humans on the trail, preserving the privacy of other trail users (such a local people, or hikers).

Transforming signals

To a computer, our world is made up of signals: time series of sensor readings that each describe a small fragment of a situation or environment. Our previous categories of applications are mostly focused on interpreting these signals and responding to them accordingly. Data from one or more

sensors is assimilated, and a simple output is constructed that either facilitates human interpretation or can be used as a control signal for an automated system.

This final category is a little different. Sometimes, rather than converting a raw signal into an instantaneous decision, we simply want to transform one signal into another (Table 2-5). As discussed in "Digital signal processing (DSP)", digital signal processing is an important part of embedded applications. In these use cases, which go much further than the traditional DSP pipeline, it is the end goal rather than a side effect:

Table 2-5. Edge AI use cases for transforming signals

Use case	Signal type
Filtering background noise to improve call quality on a cellphone.	Audio
Removing noise from photographs captured with a smartphone camera.	Vision
Generating music to accompany a musician during practice.	Audio
Blurring the background of a video stream during a remote work meeting.	Vision
Generating realistic human speech from text.	Audio
Translating one written language into another using a smartphone camera.	Vision, text
Up-sampling low resolution audio so that it sounds better to the human ear.	Audio
Compressing video using deep learning so that it can be transmitted via a low bandwidth connection.	Video
Creating a spoken representation of a visual scene for visually impaired people.	Audio
Transcribing a spoken conversation into text for convenience of note-taking.	Audio

Key benefits for transforming signals

Since digital signals are expressed over time, applications in this area tend to benefit from the latency benefits of edge AI. Bandwidth is also particularly important, since access to the original signal is required—and

transmitting the transformed signal often requires the same amount of bandwidth, if not more.

We've now explored the four high level categories that most edge AI applications can be grouped into. As edge AI technologies continue to evolve we'll see many more potential use cases open up. But technological feasibility does not automatically make something a good idea. In the next section, we'll talk about the importance of ethics and learn some of the pitfalls that can result in edge AI applications that cause more harm than good.

DEEP DIVE: BLURRING THE BACKGROUND DURING A REMOTE WORK MEETING

With the growth of remote work and videoconferencing, employees have had to get to used to their previously private home spaces being broadcast to their coworkers. To help maintain some privacy, many videoconferencing tools now support blurring the background of a video stream while leaving the subject of the video intact.

These tools depend on a technique named segmentation, which uses deep learning models to identify the pixels in a stream of video that belong to one category or another. In this case, the model is trained to distinguish between a person and their background scenery. The input is the raw stream of video from a camera. The output is a stream of video with the same resolution but with the background pixels blurred together, making it hard to see what is there.

To preserve privacy, it's important that this technique uses edge AI—otherwise, the unblurred video would be transmitted outside of the user's home. Instead, the segmentation and blurring is performed on-device before the data is transmitted.

Here's how this use case maps onto our BLERP model:

- Bandwidth: The transformation works best if it happens on the high-resolution original video stream rather than a compressed, low resolution version that may contain visual artifacts. It's often not feasible to transmit high-resolution video, so the transformation must be done on-device.
- Latency: Performing the transformation on a remote server may add additional latency vs. directly sending the video stream to a peer. Performing it on-device removes this potential extra step.
- Economics: It's cheaper to perform the required computation on the device sending video as opposed to in the cloud, where the service provider would have to pay for it.

- Reliability: With a cloud server as a middle-man, the video streaming pipeline is more complex and has a higher probability of outages. By processing on-device, the pipeline is simpler and may be less likely to fail.
- Privacy: When the data is transformed on-device, the user can be guaranteed that nobody will ever see the original video.

Building ethical applications

The first part of this chapter has covered some of the most interesting potential applications for edge AI, and the next chapter will provide a framework for breaking down problems and deciding whether they are a good fit for edge AI to solve.

But as we heard in [Link to Come], it's vital that any project is analyzed at every step along the way to make sure that it meets ethical standards. This isn't some warm-and-fuzzy process where we pat ourselves on the back for ticking some boxes and then continue with our work. Ethical problems with technology products can be life-destroying, career-ending disasters—for the end users of the products, the businesses selling them, and the developers creating them.

A solid example of this is Uber's self-driving car division. The ride-share company launched an aggressive drive towards developing a self-driving car, hiring industry luminaries and investing billions of dollars. In its rush to test a system on real streets, the company's flawed safety procedures and ineffective software led to the tragic death of a pedestrian. This disaster resulted in the shutdown of Uber's self-driving program, layoffs of hundreds of employees, and the fire sale of the self-driving car division to another business.[1]

Building a self-driving car is not an inherently unethical task: in fact, done well it could result in safer roads and reduced emissions. It seems like a noble mission. But the unique environment of edge AI lead to potential

pitfalls that are challenging to navigate. When these risks are factored in, a seemingly ethical project can become a deadly minefield.

In Uber's case, their self-driving car was subject to an incredibly common failure mode of machine learning systems: it was incapable of understanding situations that had not appeared in its training dataset. According to the National Traffic Safety Board, Uber's self-driving car lacked "the capability to classify an object as a pedestrian unless that object was near a crosswalk".

A lack of proper ethical consideration—is it right to drive a self-driving car on public roads when we do not have evidence of its ability to handle common conditions?—led directly to the death of a human being and the failure of a company division. We can only assume that the team behind Uber's self-driving software are intelligent, capable people—they were recruited as the best in the business. If *they* could end up making these mistakes, what hope is there for any of us?

The unfortunate truth is that building ethical technology is hard. Beyond the fundamental technical challenges, it's your responsibility as a professional to scrutinize your processes, ruthlessly evaluate your own work, and be willing to shut down a project if it doesn't seem to be going the right way. In a business setting, you may be fighting against organizational inertia that is more concerned with shipping something than making sure it is safe.

But you should always remember that, at the end of the day, your livelihood, reputation, and freedom are on the line if you neglect your ethical responsibilities. Even worse, you could build a product that ruins the lives of others and regret it for the rest of your days.

ETHICAL AI, RESPONSIBLE AI, AND FAIRNESS

When discussing AI and ethics it's common to hear the following related terms:

- Ethical AI: Ethical AI applications are designed to meet moral principles, typically those that are broadly shared by society.
- Responsible AI: Similar to ethical AI, responsible AI applications are designed with consideration of their potential negative impact to individuals, society, and our shared world.
- Fairness: Fairness is a difficult idea to define. According to researchers at University of California Berkeley's School of Information, fairness has different descriptions in fields such as law, social science, mathematics and engineering, and philosophy. To summarize a complex concept, AI applications can be described as "fair" if they avoid creating unequal positive or negative impact on the groups and individuals they affect.

Each of these terms represents an entire field of study and practice, and this book can't hope to capture the full depth of any of them. Instead, we will inform you of key moments within your development workflow for considering ethical issues—and encourage you to go deep on understanding how these fields apply to your specific project.

Black boxes and bias

There are two aspects of edge AI that make it especially prone to ethical issues in practice: *black boxes*, and *bias*.

The term *black box* is a metaphor for a system which is opaque to analysis. Data goes in, decisions come out, but the processes within that lead to those decisions are inscrutable. This is a common criticism of modern AI; especially deep learning models, which are famously difficult to dissect. Some algorithms, like random forests, are quite easy to interpret—if you

have access to the model, you can read its internals to understand why it makes certain decisions. But things are very different on-device.

Edge devices are often, by design, invisible. They are intended to merge into the background of our built environments; they're embedded in our buildings, products, vehicles, and toys. They are literal black boxes; their contents are invisible, often protected by layers of security to avoid any detailed inspection.

Once an AI algorithm—no matter how simple—is deployed to an edge device, it becomes a black box to anyone who is using it. And since the device is likely deployed in different real-world conditions to the laboratory conditions it was created under, even its original developers may have little insight into why it is behaving the way it does.

This is dangerous in several ways, depending on who you are. The device's users—the people who bought and installed it—are now reliant on a system they do not fully understand. They may trust the device to make the right decisions, but they don't have any real insight into the processes behind them.

This isn't a terminal problem as long as the users understand the limitations of the information they are receiving. But it can be deadly. In the example of Uber's self-driving car, the test driver was overly trusting of the self-driving algorithm and failed to react quickly enough when they saw a pedestrian was in the road. Since the test driver didn't have any way of knowing that the black box model hadn't been trained to spot pedestrians outside of crosswalks, they didn't pay sufficient attention during a critical moment.

The pedestrian, an innocent bystander who just happened to be crossing the road, was also impacted by the black box nature of the model. If they had somehow had known that the car approaching was an unreliable self-driving prototype they may not have chosen to cross. But the edge AI system, hidden inside an ordinary-looking car, provided no inherent warning. There was no way for an pedestrian to predict how it might behave.

Finally, the black box nature of edge AI creates a risk for its developers. Unlike server-side AI, which can be deployed and monitored alongside the raw data it processes, edge AI is often deployed specifically in situations where raw data cannot be captured. In practice, this means that there's sometimes no way for developers to validate whether an edge AI application is performing correctly.

For example, imagine an edge AI camera trap being used to monitor the population of an invasive species. It might turn out that the camera suffers from false negatives—it fails to recognize one in every three invasive animals that come past. The researchers will get an underestimate of the animals' population—but they will have no way of knowing, since the raw data may no longer exist.[2]

The hazards of *black boxes* are compounded by the dangers of *bias*. Bias, in edge AI systems, refers to the system being equipped with a model of its application area that does not represent the real world. This isn't usually something that happens on purpose. Instead, it's a reflection of four main things:

- Human bias: Developers have an innate bias towards a certain worldview based on their experience (i.e. thinking people only cross the street at crosswalks)
- Data bias: Datasets tend to reflect the most common events, not the outliers (i.e. a dataset may only have examples of people crossing the street at crosswalks)
- Algorithmic bias: Every AI algorithm has a different built-in bias (i.e. many object detection models don't perform well on small objects, like distant pedestrians)
- Testing bias: Real world testing is difficult, expensive, and only covers common cases (i.e. doing exhaustive testing on an artificial test course is not possible, so real streets must be used)

Bias is inevitable in any AI system. While we tend to associate the term with deliberate, intentional forms of bias (such as deliberate sexism in hiring processes), in technology projects it most commonly occurs as a

natural side-effect of the limitations of our resources. Ideally, a development team would have access to a diverse set of domain experts, an infinitely sized dataset that reflects the exact conditions of the real world, a perfect algorithm with no inherent bias, and an inexhaustible budget for real-world testing. In reality, most of these things are impossible to achieve.

When coupled with black box edge AI products, bias creates a risky situation. As described above, users, developers, and those impacted by a product are likely to assume that it *works*. They will trust in the correct, safe, and reasonable operation of a product. With no way to inspect its mechanism of operation, they are unable to test this assumption. And, since all AI systems are biased, they are guaranteed to run into problems.

A successful AI project must be aware of its own limitations and provide some structure to protect users and the public from its potential failure. It's critical for the team behind a product to define the parameters within which the product will function—and to make sure that its users are aware of these facts.

Later in the book we'll learn a framework for ensuring this awareness, and for putting the brakes on projects that are not safe to deploy. It's an ongoing process, and one that must run for the length of a project. Many projects will run into an ethical quagmire as their true effectiveness is revealed—but some projects are just wrong from the start.

Clear ethical violations

In November 2019 it was discovered that a major supplier of video surveillance cameras, Hikvision, was marketing a surveillance camera designed to classify the race of individuals, including that of Uyghurs, a Chinese minority who have been subject to vicious repression by their government. The New York Times reported that Chinese government authorities are attempting to use edge AI technology to identify Uyghur people by their appearance and "keep records of their comings and goings for search and review".

While Uber's self-driving travails resulted in an unethical situation due to poor performance, Hikvision's racial profiling technology is fundamentally

wrong. Even if functioning perfectly, the system is designed to use advanced techniques to enforce a societal bias against a subgroup of people. There is no way to limit the system's bias; in fact, the bias is present as part of the design. While it may be argued that morality is subjective, and that different societies have different moral systems, the fact is that the millions of Uyghur people being tracked by this system have had no choice in the matter—and would likely reject it if asked.

Such clear violations of moral standards may seem obvious, but human psychology makes it easy for a group of intelligent people to cross moral boundaries without realizing what they may be doing. One example of this is the service HireVue. Designed to reduce the cost of interviewing job candidates, companies use HireVue's product to analyze recorded videos where candidates answer specific questions. The company claims to use AI algorithms to rate the likelihood of a candidate being successful in a given role.

Unfortunately, the developers of HireVue did not consider the impact of human, data, algorithmic, and testing biases on their work. Their product, built to use audio-visual information in hiring decisions, inevitably incorporated the voice, accent, and appearance of candidates when making hiring decisions. The clear risk of discrimination this created led to a lawsuit and a backlash from the public, resulting in HireVue having to scrap features of their product and conduct a third party audit of their algorithms.

A further aspect to consider is that an edge AI technology may be used by customers for purposes other than those for which they were designed—and these purposes may be unethical. For example, consider an edge AI camera trap designed for spotting an endangered species. While intended for scientific research, the camera trap might easily be repurposed by poachers as a tool for locating animals that they wish to capture and sell on the black market. It's important to consider these potential "off-label" uses when designing an application, since the risk may be so high that it outweighs the potential benefits of the product.

The costs of unethical AI

Technologies that use artificial intelligence are often designed to integrate deeply into our world, shaping the day to day interactions we have with our homes, places of work, businesses and governments, and each other. This means that the failure of these systems can have a profound impact on people.

There isn’t space in this book for a full discussion of the myriad ways this can happen, but here are a few examples:

Unintended ethical violations

- Medical hardware could misdiagnose patients, affecting their treatment
- Surveillance equipment could direct enforcement against some groups of people more than others, leading to unequal justice
- Educational toys could perform better for some children than others, reducing access to learning opportunity
- Safety devices could fail due to lack of testing, leading to loss of life
- Insecure devices could be compromised by criminals, facilitating crime

Deliberate ethical violations

- Automated weapons could lead to additional deaths and disrupt the global balance of power
- Surveillance equipment could be used to deliberately target minority groups
- Smart sensors could help poachers target endangered wildlife

Avoiding ethical issues

The framework provided throughout this book will encourage you to take time during the development process to understand the ethical implications of what you are building and to make go/no-go decisions based on your findings. However, ethics is an incredibly nuanced area, and there’s no simple checklist that can ensure you are on the right path.

By far, the best way of avoiding unethical applications of AI is to build a product team with diverse perspectives in both technical expertise and lived experience. Human biases amplify technical biases, and a diverse team is less likely to have blind spots in their collective worldview. If you have a small team, it's important to reach out to the wider community to find people who are willing to help evaluate your ideas and provide feedback, adding their perspectives to the mix.

PSYCHOLOGICAL SAFETY AND ETHICAL AI

Your team's insight is crucial in identifying ethical issues, so it's vital that they feel like they have the ability to speak up and make their voices heard throughout the development process. Even in the best working environments it can feel risky for employees to speak up when they think their feedback may cause problems.

For example, imagine an employee who notices a potential ethical issue but feels unable to mention it because they feel hesitant to derail an important project. In reality, the employee may save the company time and money by pointing out a significant issue. However, if the employee feels afraid of a potential negative impact on their career, reputation, or the team's morale, they may choose not to say anything until it is too late.

Psychological safety is the feeling of being able to speak up and discuss issues without fearing negative consequences. This, along with a culture that reinforces the importance of ethics in AI, is a key factor in building successful AI projects.

Various companies and services exist to help guide teams through the process of developing ethical AI, or to audit existing applications for potential issues. Rather than providing a list of links that may rapidly go out of date once this book is published, we recommend a quick web search for "ethical and responsible AI consulting", or "algorithmic auditing".

If you are concerned about harmful "off-label" use of your work, there are also some legal tools at your disposal. The Responsible AI License (RAIL)

is a technology license designed to help developers restrict the legal use of artificial intelligence products for harmful applications. By attaching a RAIL to their product, developers create legal grounds to prevent its misuse in a specific list of applications, which can be extended to include any categories the developer would like to include. Some of the default prohibited options include surveillance, crime prediction, and generating fake photography. Of course, this only helps prevent ethical usage by entities that consider themselves bound by legal agreements.

Finally, there are many free, high quality online resources that you can use to learn more about ethical and responsible AI and evaluate the work you are doing. To get you started, here's a short list:

- Berkeley Haas' guide to mitigating bias in artificial intelligence
- Google's recommended practices for responsible AI
- Microsoft's responsible AI resources
- PwC's responsible AI toolkit
- Google Brain's People + AI Research

Summary

In this chapter we've developed a solid understanding of how edge AI fits into our world. We know the top use cases, the key benefits, and the critical ethical considerations that need to be applied.

We're now ready to dive into some of the technical details. In the next chapter, we'll learn about the technology that makes edge AI work.

---

1 The head of the division, Anthony Levandowski, was later sentenced to eighteen months in prison for theft of intellectual property—suggesting that ethical issues were a systemic problem.

2 In practice, conservation researchers solve this dilemma by storing all of the photos captured and performing manual reviews when the memory card is collected—but this costs time and money.

Chapter 3. The Technology of Edge AI

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at dan@situnayake.com and jplunkett@utexas.edu.

It's now time to meet the devices, algorithms, and optimization techniques that power edge AI applications. This chapter is designed to provide a broad overview of the most important technical elements of the field. By the end of it, you'll have the building blocks necessary to start the high level planning of an edge AI product.

Sensors, signals, and sources of data

Sensors are electronic components that give devices the power to measure their environments and detect human input. They range from extremely simple (trusty old switches and variable resistors) to mind-blowingly sophisticated (Light detection and ranging [LIDAR] and thermal imaging cameras). Sensors provide our edge AI devices with the streams of data that they use to make decisions.

Beyond sensors, there are other sources of data that our devices can tap into. These include things like digital device logs, network packets, and radio transmissions. Although they have a different origin, these secondary

data streams can be just as exciting as sources of information for AI algorithms.

Different sensors provide data in different formats. There a few data formats that are commonly encountered in edge AI applications. They can be summarized as follows:

Time series

Time series data represents the change in one or more values over time. A time series may contain multiple values from the same physical sensor—for example, a single sensor component may provide readings of both temperature and humidity. Time series data is often collected by polling a sensor at a specific rate, such as a certain number of times per second, to produce a signal. The rate of polling is known as the sampling rate, or frequency. It is common that the individual readings (known as samples) are collected with a constant period, so the time interval between two samples is always the same.

Other time series may be aperiodic, meaning the samples are not collected at a constant rate. This might happen in the case of a sensor that detects specific events—for example, a proximity sensor that sets a pin to high when something comes within a certain distance. In this case, it is common to capture the exact time when an event happened alongside the sensor value itself.

Time series data is the most common form of sensor data for edge AI. It is particularly interesting because, in addition to the sensor values, the signal includes information about the timing of the values. This provides useful information when attempting to understand how a situation is changing. In addition to timing information being useful, time series are valuable because they contain multiple readings from the same sensor, reducing the impact of momentary anomalous readings.

There is no typical frequency for a time series—they can range from a single sample a day to millions of samples per second.

Audio

A special case of time series data, audio signals represent the oscillation of sound waves as they travel through the air. They are generally captured at a very high frequency—thousands of times per second. Since hearing is a human sense, huge amounts of research and development have gone into innovations that make it easier to work with audio data on edge devices.

These technologies include special signal processing algorithms that make it easier to process audio data, which in its raw form is typically captured at an extremely high frequency. As we will see later, audio signal processing is so common that a lot of embedded hardware comes with built-in functionality for performing it efficiently.

One of the most widespread uses of edge AI audio processing is in speech detection and classification. That said, audio doesn't even have to be in the spectrum of human hearing. Sensors used by edge AI devices can potentially capture ultrasound (higher than audible by human hearing) and infrasound (lower than audible by humans) data.

Image

Images are data that represent the measurements taken by a sensor that captures an entire scene, as opposed to a single point. Some sensors, like cameras, use an array of tiny elements to capture data from the entire scene in one go. Other sensors, like LIDAR, build up an image by mechanically sweeping a single sensor element across the scene over a period of time.

Images have two or more dimensions. In their typical form, they can be thought of as a grid of "pixels", where the value of each pixel represents some property of the scene at the corresponding point in space. The size of the grid (for example, 96x96 pixels) is known as the *resolution* of the image. A pixel may have multiple values, or channels—for example, while a black and white image only has one value per pixel (representing how light or dark the pixel is), a color image may have three values per pixel (representing three colors that can be mixed to represent any other color in the visible spectrum).

The typical representation of images, as an n-dimensional grid, means that they contain spatial information about the relative proximity of different aspects of a scene to one another. This information is extremely valuable in understanding what a scene contains. There are entire classes of *image processing* and *computer vision* algorithms that make use of this information.

<figure showing n-dimensional images>

Images don’t have to represent visible light, or even light at all. They can represent infrared light (often used to measure the temperature of parts of a scene), time-of-flight (in the case of LIDAR, which measures how long it takes light to bounce back from each part of a scene), or even radio waves.

Video

Technically another special case of time series data, video deserves its own category due to its distinct utility. A video is a sequence of images, each representing a snapshot of a scene at a point in time. As a time series, video has a sampling rate—although in the case of video it is typically referred to as the frame rate, since each individual image in the sequence is known as a frame.

Video is a very rich format—it contains both spatial information (within each frame) and temporal information (between each frame). This richness means that it tends to occupy a lot of memory, so it tends to require more capable computing devices.

HOW ARE VALUES REPRESENTED?

All of the above categories represent individual sensor readings using single numeric values. For example, a time series is a sequence of individual readings, and an image is a grid composed of individual readings.

Each reading is a number, and can be represented on a computer in a variety of different ways. For example, here are some typical numeric types used to represent sensor data in C++:

- Boolean (1 bit): a number with 2 possible values
- 8-bit integer: a number with 256 possible values
- 16-bit integer: a number with 65536 possible values
- 32-bit floating point: can represent a wide range of numbers with up to 7 decimal places, with a maximum of 3.4028235×10^{38}

By varying the numeric type used to represent a value, developers can trade numerical precision for reduced memory usage and computational complexity.

Types of sensors and signals

There are thousands of different types of sensors on the market. A nice way of grouping them is by their *modality*. According to CMU, modality refers to the way in which something happens or is experienced. From a human perspective, our senses of sight, hearing, or touch all have different modalities.

There's no strictly defined list of sensor modalities, and the best way to describe them may vary between industries and applications. In the following section we'll explore some groupings that make sense from a broad edge AI perspective:

- Acoustic and vibration

- Visual and scene
- Motion and position
- Force and tactile
- Optical, electromagnetic, and radiation
- Environmental and chemical

There are also many non-sensor data sources available to edge devices—we'll go through those, too.

Acoustic and vibration

The ability to "hear" vibrations allows edge AI devices to detect the effects of movement, vibration, and human and animal communication at a distance. This is done with acoustic sensors, which measure the effect of vibrations that are travelling through a medium which might range from air (in the case of microphones) to water (hydrophones) or even the ground (geophones and seismometers). Some vibration sensors are designed specifically for use with heavy industrial machinery.

< Some images of acoustic sensors >

An acoustic sensor typically provides a time series that describes the variation of pressure in its medium. Acoustic signals contain information across various frequencies—for example, the high and low notes of a singing voice. Acoustic sensors generally operate in a certain frequency range, and they may not have a linear response to frequencies even within that range.

In addition to their non-linear frequency response, the ability of acoustic sensors to capture high frequencies depends on their sample rate. To accurately capture a high frequency signal, an acoustic sensor must have a sufficiently high sample rate. When building an edge AI application for acoustics, make sure you understand the properties of the signal you are trying to measure, and choose sensor hardware that is a good fit.

Visual and scene

It is common for edge AI applications to need to understand the scenery around them in a passive manner, without reaching out to touch it. The most common sensors used for this task are image sensors, ranging from tiny, low-power cameras to super high quality multi-megapixel sensors. As described above, the images obtained from image sensors are represented as arrays of pixel values.

<Picture of some image sensors>

Image sensors capture light using a grid of sensor elements. In a camera, light from a scene is focused onto the sensor by a lens. The area that can be imaged by a camera is known as its field of view, and it depends on the size of the lens and the image sensor.

Some common variations in image sensors:

- Color channels—for visual light, sensors can commonly capture data in grayscale or color (red, green, and blue, or RGB).
- Spectral response—which wavelengths of light the image sensor is sensitive to, which may exceed the range of human vision. This can even include infrared radiation, allowing sensors known as thermal cameras to “see” heat.
- Pixel size—larger sensors can capture more light per pixel, increasing their sensitivity.
- Sensor resolution—the more elements on a sensor, the more fine detail it can capture.
- Frame rate—how frequently a sensor can capture an image, typically in frames per second.

Since illumination of a scene is sometimes required, it is common to pair image sensors with light emitters—in both visible and invisible ranges of the spectrum. For example, an infrared LED can be used with an infrared-sensitive camera to illuminate dark scenes without disturbing humans or animals with visible light.

Larger, higher resolution sensors typically require more energy. High resolution sensors produce large amounts of data, which can be difficult to process on smaller edge AI devices.

A relatively new group of image sensors, known as event cameras, work slightly differently. Instead of capturing the entire visual field at a specific frame rate, each pixel in the camera responds individually to changes in brightness but remains silent if nothing is happening. The result is a time series of individual pixel changes that can be easier for edge AI devices to process than a large sequence of full frames.

Another interesting type of image sensors are known as range imaging sensors. These allow devices to image their surroundings in three dimensions—often by emitting light and measuring how long it takes to bounce back, a technique known as “time-of-flight”. A common time of flight sensor technology is known as LIDAR. LIDAR sensors work by scanning their surroundings with a laser beam, measuring how much of the light is reflected back to the sensor. This allows them to visualize an area in three dimensions.

< Picture/diagram of LIDAR >

LIDAR and other time-of-flight sensors are typically much larger, more complex, expensive, and energy intensive than standard image sensors. The large amounts of data they generate can be difficult to process and store on edge devices, which also limits their utility. LIDAR is typically used for mapping environments—including to help self-driving vehicles navigate the world.

RADAR, or radio detection and ranging, is also occasionally used by edge devices to understand the position of surrounding objects in three dimensions, potentially at long range. Like LIDAR, it is complex and has high energy requirements—but is definitely an option if your use case requires it.

Motion and position

It can be useful for edge AI devices to understand both where they are and where they might be headed. Fortunately, there are many different types of

sensors that can help. This is a broad category, ranging from the simplest (mechanical tilt switches) to the most complicated (the satellite-enabled Global Positioning System). As a whole, they allow devices to understand their position and motion within the world.

Here's a list of typical motion and position sensors for edge AI applications:

- Tilt sensor—a mechanical switch that is on or off depending on its orientation. Super cheap and easy to use.
- Accelerometer—measures the acceleration (the change in velocity over time) of an object across one or more axes, often at a high frequency. Accelerometers are the swiss army knives of motion sensing, used for everything from recognizing the characteristic motions of sporting activities (in smart watches) to sensing the vibrations of industrial equipment (in predictive maintenance).
- Gyroscopes—measure the rate of rotation of an object. Often paired with accelerometers to give a picture of the motion of an object in 3D space.
- Rotary or linear encoder—measures the exact position of either a shaft or axle (rotary) or a linear mechanism (like the position of an inkjet printer head). Often used in robotics to capture the positions of robots' wheels, limbs, and other appendages.
- Time of flight—a sensor that uses an electromagnetic emission (light or radio) to measure the distance from a sensor to whatever object is directly in its line of sight.
- Real time locating systems—systems that use multiple transceivers in fixed locations around a building or site to track the position of individual objects, such as pallets in a warehouse.
- Global Positioning System (GPS)—a passive system that uses radio signals from satellites to determine the location of a device, down to a few meters. Requires line of sight from the device to several satellites.
- Inertial measurement unit (IMU)—a system that uses multiple sensors to approximate the current position of a device based on its motion as

measured from an internal frame of reference (as opposed to using external signals such as GPS).

Motion and position are typically represented as a time series of sensor readings. Given the number of sensor types in this category, there are options for every cost and energy budget. Typically, the more confidence in absolute position required, the more cost and complexity involved.

Force and tactile

From switches to load cells, force and tactile sensors help edge AI devices measure the physicality of their environment. They can be helpful in facilitating user interaction, understanding the flow of liquids and gases, or measuring the mechanical strain on an object.

Here are some typical force and tactile sensors:

- Buttons and switches—traditional switches can be used as simple buttons for human interaction, but they can also serve as sensors that provide a binary signal that indicates when a device is colliding with something.
- Capacitative touch sensors—can measure the amount that a surface is being touched by a conductive object, like a human finger. This is how modern touchscreens work.
- Flex sensors—these measure how much an object is being bent, which can be interesting for detecting damage to objects and for building tactile human interface devices.
- Load cells—these measure the precise amount of physical load that is applied to them. They come in a wide range of sizes, from tiny (useful for measuring the weight of small objects) to gigantic (measuring strain in bridges and skyscrapers).
- Flow sensors—designed to measure the rate of flow in liquids and gases, such as water in a pipe.

- Pressure sensors—used to measure pressure of a gas or liquid, either environmental (such as atmospheric pressure) or inside a system (such as inside a car tire).

Force and tactile sensors are typically simple, low-energy, and easy to work with. Their measurements are easy to represent as time series. They are especially useful when building tactile user interfaces or detecting when a robot (or other device that can move around) has hit something.

Optical, electromagnetic, and radiation

This category includes sensors that are designed to measure electromagnetic radiation, magnetic fields, and high energy particles, in addition to basic electical properties such as current and voltage. This may sound exotic, but it includes familiar things like measuring the color of light.

Here are some typical optical, electromagnetic, and radiation sensors:

- Photosensors—a category of sensors that detect light at various wavelengths, both visible and invisible to the human eye. This can be useful for many things, from measuring ambient light levels to detecting when a beam of light has been broken.
- Color sensor—these use photosensors to measure the precise color of a surface, which can be helpful for recognizing different types of objects.
- Spectroscopy sensors—these use photosensors to measure the way that various wavelengths of light are absorbed and reflected by materials, giving an edge AI system insight into their composition.
- Magnetometer—these measure the strength and direction of magnetic fields. A subtype of magnetometer is a digital compass, which can indicate the direction of North.
- Inductive proximity sensors—these use an electromagnetic field to detect nearby metal. They are commonly used to detect vehicles for traffic monitoring.

- Electro-magnetic field (EMF) meters—measure the strength of electromagnetic fields. This includes those emitted incidentally, for example by industrial equipment, or those intentionally emitted by radio transmitters.
- Current sensor—measures the flow of current through a conductor. This can be useful in monitoring industrial equipment, since fluctuations in current can provide information about the functioning of the equipment.
- Voltage sensor—measures the amount of voltage across an object.
- Semiconductor detector—these measure ionizing radition, which is composed of extremely fast moving particles, typically created by the decay of radioactive substances.

As with many other sensors, this category generally provides a time series of measurements. While useful for measuring ambient conditions, the sensors described above can also be useful in arrangements where they detect emissions that are produced deliberately by a device. For example, a photosensor can be paired with a light emitter on the other side of a hallway to detect when someone is moving past.

Environmental, biological, and chemical

A loose category that includes many different types of sensors, environmental and chemical sensing allows edge AI devices to sniff the composition of the world around them. Some common types of sensors include:

- Temperature sensors—these measure temperature, either of the device itself or of a distant source of infrared emissions.
- Gas sensors—many different sensors exist to measure concentrations of different gases. Common gas sensors include humidity sensors (which measure water vapor), volatile organic compound (VOC) sensors, which measure a selection of common organic compounds, and carbon dioxide sensors.

- Particulate matter sensor—these measure the concentration of tiny particles in a sample of air, and are commonly used to monitor pollution levels.
- Biosignals sensors—these cover a vast range of signals that are present in the bodies of living things—for example, measurement of electrical activity in the human heart (electrocardiography) and brain (electroencephalography).
- Chemical sensors—many different sensors are available that are designed to measure the presence or concentration of specific chemicals.

This category of sensors generally provides a time series of readings. Due to their need to interact chemically and physically with the environment, they can sometimes be difficult to work with—for example, calibration against known quantities of chemicals is often required, and sometimes sensors require a warm-up period before they can take a reliable reading. It is common for environmental sensors to degrade over time and require replacement.

Other signals

In addition to gathering signals from the physical world, many edge AI devices have access to a rich feed of virtual data. This can be split roughly into two groups: introspective data, about the state of the device itself, and extrospective data, about the systems and networks that the device is connected to.

Depending on the device, various types of internal state may be available. These could include:

- Device logs, tracking the lifecycle of the device since it was powered up. This could provide information about many different things: configuration changes, duty cycle, interrupts, errors, or anything else you choose to log.

- Internal resource utilization—this might include available memory, power consumption, clock speed, operating system resources, and usage of peripherals.
- Communications—a device can keep track of its physical connections, radio communications, networking configuration and activity, and the resulting energy usage.
- Internal sensors—some devices have internal sensors; for example, many system-on-chip devices include a temperature sensor to monitor their CPU.

One interesting usage of introspective data is in preserving battery life. Lithium rechargeable batteries can lose capacity if they are continually held at 100% charge while plugged in. Apple's iPhone uses an edge AI feature known as Optimized Battery Charging in order to avoid this problem. It uses an on-device machine learning model to learn the user's charging routine, then uses this model to minimize the amount of time the battery spends full—while ensuring the battery is still charged when the user needs it.

Extrospective data streams, which come from outside the device, can be extremely rich in information. Here are some possible sources:

- Data from connected systems—it's common for edge AI devices to be deployed in a network, and data forwarded by adjacent devices can be used as input to AI algorithms. For example, an IoT gateway could use edge AI to process and make decisions based on the data collected by its nodes.
- Remote commands—an edge AI device might receive control instructions from another system or user. For instance, the user of a drone could request that it move to a certain coordinate in 3D space.
- Data from APIs—an edge AI device can request data from remote servers to feed into its algorithms. For example, a home heating system equipped with edge AI might request weather forecast data from an online API and use the information to help decide when to turn the heating on.

- Network data—this might include network structure, routing information, network activity, and even the contents of data packets.

Some of the most interesting edge AI systems make use of all of these data streams together. Imagine an agricultural technology system that helps a farmer take care of crops. It might include remote sensors out in the fields, connections to important online data sources (like the weather forecast, or the price of fertilizer), and a control interface used by the farmer. As an edge AI system, it could potentially operate without an Internet connection—but if it had one, it could make use of valuable information.

In more complex system architectures, edge AI also pairs nicely with server-side AI; we'll learn more about that later in this chapter.

Processors for edge AI

One of the most exciting parts of edge AI is the vast—and growing—array of hardware that applications can make use of. In this section, we'll explore the high-level categories of hardware and learn what makes each one suited to a particular niche.

We're in the midst of a Cambrian explosion of edge AI hardware, so in the time since this book was published it is likely that there are even more options than what is printed here. With a spectrum that runs from cheap, low-power microcontrollers to lightning fast GPU-based accelerators, developers can find hardware that is the perfect fit for almost any application.

Microcontrollers and digital signal processors

It could be argued convincingly that microcontrollers are the foundation of our modern world. They're the tiny, cheap computers that animate everything from car engines to smart appliances. Microcontrollers are manufactured in astonishing volume; it's projected that 26.89 billion of them will be shipped in 2022—that's three and a half for every human being on the planet.

MCUS

Microcontrollers are often referred to as MCUs, an acronym for Microcontroller Units.

Microcontrollers are typically used for single-purpose applications, like controlling a piece of machinery. This means that they can get away with being a lot simpler than other types of computers that need to run multiple programs—for example, they generally do not use an operating system. Instead, their software is run directly on the hardware and incorporates the low-level instructions necessary to drive any peripherals. This can make software engineering for microcontrollers quite challenging, but it gives developers a lot more control over exactly what is going on when their programs run.

One of the distinguishing characteristics of microcontrollers is that the majority of their components are implemented on a single piece of silicon; this is key to their relatively low cost. In addition to a processor, a microcontroller is generally equipped with flash memory (for storing programs and other useful data), RAM (for storing state during program execution), and various technologies for communicating with other devices (such as sensors) using either digital or analog signals.

The microcontroller world is incredibly diverse—part of the reason they are so valuable is that they are available in variants to suit every imaginable situation. For the purposes of this book, we'll divide them into three main categories: low-end, high-end, and digital signal processors.

Low-end MCUs

Many MCUs are designed specifically for low cost, small size, and energy efficiency. The trade-off is that they have limited computational resources and capabilities. Here are some typical specifications:

- 4-bit to 16-bit architecture
- <100 MHz clock speed

- 2-64 KB of flash memory
- 64 bytes to 2KB of RAM
- Digital input and output
- Current draw: low single digits to low tens of milliamps at ~3-5 volts
- Cost: One or two dollars per unit

A NOTE ON POWER

The amount of energy that a microcontroller consumes depends on many factors, most of which are in the developers' hands to control. Amongst other things, you can decrease power consumption by running the processor at a reduced speed, switching off features when not in use, and putting the entire microcontroller into idle mode when it is not currently processing data.

This flexibility, coupled with the general diversity of the microcontroller market, makes it tricky to quote exact numbers for power consumption. If you're designing against tight power constraints, you'll want to evaluate the hardware and measure energy use for yourself.

Many low-end MCUs used today are based on the Intel 8051, a design that has been in use since the 1980s. While technology has continued to improve, there's always a need for simple, low-cost and low-power hardware, so these chips are here to stay. They are extremely common across many industries.

Low-end MCUs have some significant disadvantages when it comes to edge AI. Since they lack memory and compute, they aren't well suited to dealing with large amounts of data or complex signal processing. They generally do not have any hardware implementation of floating point arithmetic, meaning calculations involving rational numbers can be incredibly slow. These attributes limit the types of edge AI algorithms they can run.

The typical applications for low-end MCUs play to their benefits: high-reliability automotive and medical devices, low-cost appliances, gadgets, and infrastructure. While they are an important part of the MCU world, their computational limitations mean that low-end MCUs probably shouldn't be your first choice of target for an edge AI application.

That said, as we mentioned in the first section of the book, edge AI programs don't always have to be computationally challenging. Low-end MCUs are perfectly capable of running complex conditional logic, which may be enough for what you need to do. They can also form part of a network of connected devices that makes use of edge AI—for example, a low-end MCU could capture sensor data and forward it to a more sophisticated device for decision-making.

High-end MCUs

At the other end of the MCU spectrum, today's most powerful microcontrollers have enough compute to give a '90s vintage personal computer a run for its money. In many cases, they still manage to be highly energy efficient. Here are some typical specs:

- 32-bit architecture
- <1000 MHz clock speed
- 16KB to 2MB of flash
- 2KB to 1MB of RAM
- Optional hardware support for faster math
- Floating point unit (FPU)
- Single instruction, multiple data (SIMD)
- Digital and analog input and output
- Current draw: low single digits to high tens of milliamps at ~3-5 volts
- Cost: from low single digits to low tens of dollars per unit

High-end MCUs provide a big jump in performance, thanks to faster processors and a 32-bit architecture. In addition, many models of MCUs have hardware support for some neat tricks that increase computation speed. One of these, SIMD, allows the processor to run several computations in parallel—which can be extremely helpful when running signal processing and machine learning applications, which involve a lot of computation.

Increasingly, high-end MCUs are designed with edge AI applications in mind. It's common for vendors to offer software and libraries that help optimize edge AI code to run efficiently on-device. Another big benefit is a trend towards providing larger amounts of flash and RAM—very helpful for manipulating data and storing large machine learning models.

High-end MCUs are used in a huge range of use cases, from sensing and IoT to digital gadgets, smart appliances, and wearables. At the time of writing, they represent the sweet spot for cost, energy usage, and computational ability for embedded machine learning. They have just enough power to run capable deep learning models—including deep learning models that can process visual information—but they remain simple enough to embed very cheaply into a wide range of applications. Microcontrollers based on Arm's Cortex-M cores are extremely popular.

That said, it's becoming increasingly common to pair general-purpose high-end microcontrollers with purpose-built co-processors designed to accelerate deep learning workloads. We'll cover that more in "Deep learning accelerators".

PERFORMANCE CHARACTERISTICS

An average high-end microcontroller can process audio using deep learning in near real-time, and low-resolution video at a second or so per frame.

Digital signal processors (DSPs)

An interesting subcategory, DSPs are special microcontrollers that are designed to be highly efficient at transforming digital signals. Instead of general purpose computation, their architecture is designed to run specific algorithms and mathematical operations as quickly as possible—including things like multiply-accumulates and fourier transforms, which we'll encounter in "Algorithms for edge AI".

As luck would have it, many of those mathematical operations are very helpful in edge AI, both for processing data and for running machine learning models. This can make DSPs a valuable tool. The downside of DSPs is that they are not designed for general purpose compute, meaning that they may not be suitable for running the non-edge AI parts of your application.

Today's high-end MCUs often have some of the features of DSPs, such as SIMD instructions that can help increase throughput for signal processing tasks—in fact, some are described as "digital signals controllers" in order to highlight these abilities. However, dedicated DSPs can still be useful. For example, many smartphones that include voice assistants (such as the Google Assistant) include a DSP chip in order to run an always-on keyword spotting model without hurting battery life.

HETEROGENEOUS COMPUTE

Hardware designers aren't limited to choosing a single microcontroller for a given application. It's actually quite common to combine multiple microprocessors in a single product. For instance, an edge AI device may include a small, low-power MCU to run its basic operations—alongside a large, powerful MCU that is used for occasional signal processing and machine learning workloads.

This type of set-up is known as heterogeneous compute, and it's increasingly important in edge AI. One of the big challenges with heterogeneous compute is deciding how to split a computational workload between two processors in order to maximize efficiency. If you can do it right, there are major rewards.

Some architectures for edge AI application, such as those that use (XREF for "Cascading Models"), lend themselves particularly well to heterogeneous compute. The rise of "Deep learning accelerators" is making it an increasingly important concept.

System on Chip

After microcontrollers, the next most common type of edge compute comes in the form of system on chip (SoC) devices. While a microcontroller is a stripped-down, optimized version of a computer with all the fat trimmed away, SoC devices attempt to squeeze all of the functionality of an entire traditional computer system into a single chip.

Unlike microcontrollers, whose software interacts directly with the hardware, SoC devices run traditional operating systems that abstract away a lot of the hardware so that developers can focus entirely on their application code. Developers can use the same tools and environments that they use to write server and desktop applications, including high level languages like Python (modern microcontrollers are typically programmed in C or C++).

There are two costs associated with this ease of use: efficiency, and complexity. SoCs are generally a lot less energy efficient than microcontrollers, which limits their fields of application. They are still an order of magnitude more efficient than traditional computer systems with separate peripherals, but they're nowhere near as good as microcontrollers for keeping energy usage to a minimum. This additional energy usage may also introduce heat management issues.

The additional complexity represented by an operating system is another burden on SoC devices. With huge amounts of OS code being run alongside a developer's application, it's more difficult to guarantee reliability in the field.

SoCs tend to be a lot more powerful than microcontrollers, and they have a lot more features. Here are some typical stats:

- 64-bit architecture
- >1 GHz clock speed
- Multiple processor cores
- External RAM and flash (generally multiple gigabytes)
- 2D or 3D graphics processing unit
- Wireless networking
- High performance digital input and output
- Current draw: hundreds of milliamps at ~5 volts
- Cost: tens of dollars per unit

PERFORMANCE CHARACTERISTICS

An average SoC can process audio and high-resolution video using deep learning in near real-time.

Despite being far less efficient than microcontrollers, SoCs have been revolutionary. They allow the power of a formidable general purpose

computer to be deployed in an extremely small form factor. In the modern world, SoCs are pervasive—they power our mobile phones, televisions, car entertainment systems, industrial hardware, security systems, IoT gateways, and pretty much anything else that requires flexible computational power in a small package.

Their power, flexibility, and ease of use makes them especially valuable for edge AI. Developers can use familiar tools to develop applications that run on SoCs, and they have enough memory and processing power to run complex algorithms, such as relatively large deep learning models. There are very few types of edge AI algorithm that will not run on an SoC. Ease of use makes SoCs a great choice for prototyping edge AI applications even if the end goal is to move to cheaper or more efficient hardware.

EMBEDDED LINUX

Linux has become a very common choice of operating system for SoC devices. It's open source, which means it's free to use, and has a lot of community support. Being able to use familiar Unix development tools makes it very easy to work with embedded Linux systems.

Deep learning accelerators

Both microcontrollers and SoCs are typically general purpose computers—they are designed to be as flexible as possible. However, if you're willing to sacrifice some flexibility it's possible to design integrated circuits that run certain operations *extremely* fast.

With the advent of "Deep learning" on embedded devices, semiconductor companies have started to produce accelerators that can be paired with microcontrollers and SoCs to allow deep learning models to be run faster and more efficiently.

There are various types of deep learning accelerators with their own trade-offs between energy use and flexibility. At one end of the spectrum, devices like Syntiant's NDP10x series have hardware implementations of specific deep learning model architectures (we'll learn more about these later) that

can run quickly with incredibly low energy. Since the algorithm itself is baked into the silicon, these devices are not very flexible—but they can be extremely efficient.

At the other end of the spectrum, devices based on graphics processing unit (GPU) technology, like NVidia's Jetson and Google's Coral, offer a huge amount of flexibility and can run basically any type of deep learning model. The trade-off for this flexibility is that they are nowhere near as energy efficient.

Between the two ends of the spectrum are many different types of devices with varying degrees of flexibility and efficiency—like Syntiant's NDP120 or Arm's Ethos-U55 design.

PERFORMANCE CHARACTERISTICS

Deep learning accelerators tend to be extremely fast—you can expect enough computational power to process audio and video in real time. Some devices can even process multiple streams in parallel.

Generally, deep learning accelerators are paired with either microcontrollers or SoCs. The conventional processor runs the application logic, and the accelerator runs the deep learning workload. Many designs combine the microprocessor and accelerator in a single package, and provide special tools to help developers split the processing between them.

Early deep learning accelerators provided very little freedom of choice with regards to network architecture, but as the field matures devices are becoming more flexible. We're still in the very early days, so you can expect big advances and efficiency gains over time. In the long term, expect absurdly capable devices with miniscule power budgets—real time video processing or language transcription that can run for years on a small battery.

FPGAs and ASICs

For the ultimate performance and efficiency benefits, designing your own processor circuit is an option. It's difficult, time consuming, and expensive,

so it isn’t something to be taken lightly, but for certain applications it might make sense.

Field programmable gate arrays, or FPGAs, are silicon integrated circuits that can be reprogrammed on demand to implement custom hardware designs. They allow engineers to create a custom processor design that implements a specific algorithm as efficiently as possible, then load it onto a device for deployment. The designs are created using special programming languages called “hardware description languages”.

Application-specific integrated circuits, or ASICs, are integrated circuits that are customized for particular applications. Unlike ASICs, they can’t be reprogrammed—their logic is permanently written into silicon.

Development with FPGAs is substantially cheaper than with ASICs, but the per-device cost is higher, so it’s common for companies to use FPGAs for prototypes or small production runs and ASICs for high volume. The engineering cost of creating an ASIC puts them out of reach for the majority of companies.

FPGA developer tools are becoming easier to use and more accessible, but they’re still a relatively niche option in edge AI. Researchers are working on tooling that can automatically convert deep learning models into efficient FPGA implementations, so it’s likely that FPGAs will play an increasingly large role in edge AI over time. Here are some interesting projects in the space at the time of publication:

- Tensil.ai, a machine learning model compiler and hardware generator for FPGAs
- Google’s CFU Playground, which helps developers create deep learning accelerators using FPGAs

BOARDS AND DEVICES

A processor isn't much use by itself—it needs to be mounted onto a board along with the other components that make up a full device—power supply, sensors and peripherals, and connectors. Most mass-produced edge AI products use custom printed circuit boards designed for their specific application.

Since these custom boards take time to design and produce, a lot of early engineering work is done with *development boards*, also known as *dev boards*. These are ready-to-use devices, sold by processor manufacturers, that feature a given processor along with everything needed to connect to it and develop software. They allow embedded engineers to evaluate various processors for a particular use case.

<image of dev board>

The computational requirements of edge AI algorithms means there's an interplay between hardware and algorithm choice, which makes development boards extremely valuable. With access to a few development boards, developers can quickly test their algorithms on real processors and find the ideal balance between performance, energy usage, and cost.

While traditionally only used during early prototyping, some manufacturers have realized the potential benefits of development boards for production use. If you're making a small batch of hardware, it may not be worth the extra cost and time to design a custom printed circuit board—which would only be economical if produced at large volume. Instead, you might choose to use a pre-designed platform such as the Arduino Portenta, which features an MCU and a flexible set of inputs and outputs that allow you to easily integrate it into other systems.

These types of devices are available for SoCs and accelerators, too—for example, Raspberry Pi produce a range of fully integrated "single board computers" (SBCs) based on powerful SoCs, and NVidia's Jetson accelerators allow developers to quickly run code on accelerator

hardware. Many of these platforms provide a range of compatible devices, so you can prototype using a single board computer and then deploy to a "system on module" (SOM) designed to integrate into your own hardware without any code changes.

<image of SBC and SOM>

Dev boards are typically just bare circuit boards without any enclosure, so they can't be used in the field without at least a bit of design work. If you'd prefer a fully finished device, industrial IoT gateways place SoCs in rugged enclosures with standard I/O ports, networking hardware, and power supply. They can be fairly expensive, but they may save time and money versus designing and manufacturing a full piece of hardware.

The most common type of pre-built edge AI device—by far—is the smartphone. There are entire books on integrating AI into smartphone apps, such as Laurence Moroney's AI and Machine Learning for On-Device Development, so we won't be covering that topic here. More within our scope is the use of MCUs and DSPs to power specific smartphone features, such as digital assistants that wake up when specific keywords are spotted.

Beyond their integrated edge AI, smartphones can be a handy tool for edge AI developers who are prototyping applications. Since they are battery powered and have great connectivity, they can be useful for collecting initial data or testing out early versions of machine learning models during the initial stages of development, when proving feasibility is the most important thing.

Edge servers

At the other end of the spectrum from custom silicon, it's possible to run conventional server hardware—the same that might be deployed in a data center—at the edge of the network. These powerful computers run a full scale server operating systems (typically Linux or Windows) and can be treated in the same way as any other cloud server. If they have access to AI-specific acceleration, it's likely in the form of GPUs. Some edge servers are

sold in ruggedized form factors that are better suited to industrial settings (like a factory floor) than their data center dwelling equivalents.

The power of edge servers means that they can provide many of the benefits of cloud compute while maintaining the security, privacy, and convenience that comes with keeping data on-site. For some applications, they can provide the best of both worlds—high capability hardware, low latency, reduced risk of data leakage, and economic use of bandwidth.

Another benefit of edge servers is that they can be treated as essentially just another piece of standard IT infrastructure. This means they can fit neatly into the procedures and skillsets of an existing IT department. In fact, it wasn't long ago that *all* commercial compute was done with on-premise servers. Edge compute used to be the norm for every business.

Edge servers have two major downsides: they use huge amounts of energy, and they are very large. If you need vast amounts of compute located conveniently on-site, these trade-offs can be worth it. However, they are typically limited to fixed locations such as buildings and factories where there is spare room and a reliable power supply.

If full-sized edge servers feel like overkill for your application (and they probably are), Linux SoCs offer a great compromise. As standard Linux boxes, an IT department can treat them like any other server—but they are available in tiny, power efficient forms.

Multi-device architectures

Edge AI applications aren't always implemented directly on the devices that host the actual sensors. Sometimes, it makes sense to use a multi-device architecture. For example, sensors on a fleet of shipping pallets might use low power radio to report data back to a gateway device mounted in a truck. The gateway, with less constraints on energy use and insight into the data from multiple pallets, could run the sophisticated edge AI logic that makes decisions with the data.

<diagram showing this architecture>

Things can get even funkier with “Heterogeneous compute” in the mix. A single device might contain multiple types of processor: for example, one for running application code and another for running ML algorithms. A complete system might be composed of many devices, some with multiple processors, that collect and process data at many different points depending on which BLERP benefits are needed. This type of solution can even involve cloud computation.

A great example of this type of architecture is a smart speaker with a voice assistant. Typically, they have at least two processors. The first is a low-power, always-on chip that runs DSP and a machine learning model to listen out for wakewords without using too much energy.

The second is an application processor, which is woken up by the always-on chip when the wakeword is detected. The application processor might run a more sophisticated model to try to catch any false positives that got past the always-on chip. Together, these two processors can identify wakewords without violating user privacy by streaming private conversations to the cloud.

Once the wakeword has been confirmed, the application processor streams the audio to a cloud server, which performs speech recognition and natural language processing in order to come up with an appropriate response.

<diagram showing voice assistant>

When designing a system, don’t be afraid to consider using multiple devices to tackle some of the trade-offs involved with different device types. Some common situations where it can be helpful are:

- Tracking large numbers of individual objects: this can get expensive if high-end AI-capable hardware is used on every object.
- Reducing energy use: sensors are battery powered and need to last a long time.
- Protecting privacy: sending data directly to a large device or cloud server might violate privacy norms.

- Integrating with legacy equipment: existing sensors or gateways might be supplemented with edge AI devices rather than being replaced.

Devices and workloads

It’s important to understand what each type of device is capable of. Table 3-1 provides a quick reference you can use to break down which types of devices are capable of processing which types of data. It shows the level of support for each data type on a given device: *Good*, *Limited*, or *None*.

However, bear in mind that each category is broad and every individual device is unique. Not all high-end MCUs are the same. It’s also worth noting that the state of the art moves *fast* and that this reference may quickly become outdated!

Table 3-1. Data types and devices

Device type	Low frequency time series	High frequency time series	Audio	Low resoluti image
Low-end MCU	Limited	Limited	None	None
High-end MCU	Good	Good	Good	Good
High-end MCU with accelerator	Good	Good	Good	Good
DSP	Good	Good	Good	Good
SoC	Good	Good	Good	Good
SoC with accelerator	Good	Good	Good	Good
FPGA/ASIC	Good	Good	Good	Good
Edge server	Good	Good	Good	Good
Cloud	Good	Good	Good	Good

Algorithms for edge AI

There are two main categories of algorithms that are important in edge AI: feature engineering, and artificial intelligence. Both types have numerous subcategories, and in the next section of this chapter we're going to explore a cross-section of them.

The goal is to provide an overview for each algorithm type from an engineering perspective, highlighting their typical usage, strengths, weaknesses, and suitability for deployment on edge hardware. This should give you a place to start when planning real world projects, which we'll walk through in the coming chapters.

Feature engineering

In data science, feature engineering is the process of turning raw data into inputs usable by the statistical tools we use to describe and model situations and processes. Feature engineering involves using your domain expertise to understand which parts of the raw data contain the relevant information, then extracting that signal from the surrounding noise.

From an edge AI perspective, feature engineering is all about transforming raw sensor data into usable information. The better your feature engineering, the easier life is for the AI algorithms that are attempting to interpret it. When working with sensor data, feature engineering naturally makes use of digital signal processing algorithms. It can also involve chopping the data into manageable chunks.

Working with data streams

As we've seen, the majority of sensors produce time series data. The goal of an edge AI application is to take these streams of time series data and make sense of them.

The most common way to manage streams is to chop a time series into chunks, often called windows, then analyze the chunks one at a time. This results in a time series of results that you can interpret in order to understand what is going on. The following diagram shows a single window being taken from a stream of data:

< diagram >

It takes a certain amount of time to process a single chunk of data—we can call this the *latency* of our system. On a single-threaded device (like most microcontrollers), this limits how often we can take and process a window of data. The rate at which we can capture and process data is known as the *frame rate* of a system, often expressed in the number of windows that can be processed per second. In a running application, a 10 frame per second process might look like this:

< diagram >

The lower the latency, the more windows of data can be analyzed in a given period of time. The more analysis you can do, the more reliable the results. For example, imagine we are using a machine learning model to recognize a

command. If the windows are too far apart, we might miss critical parts of a spoken command and not be able to recognize it:

< diagram >

The choice of window size is very important. The larger the window, the longer it takes to process the data within it. However, larger windows contain more information about the signal—meaning they may make life easier for the signal processing and AI algorithms being used. The tradeoff between window size and frame rate is an important thing to explore when you are developing a system.

As we'll see later, there are many different AI algorithms—and some of them are more sensitive to window size than others. Some algorithms (typically those that maintain an internal memory of what is occurring in a signal) are able to work well with very small window sizes, while others require large window sizes in order to properly parse a signal. Algorithm choice also impacts latency, which also constrains window size. It's a complex system of trade-offs between window size, latency, and algorithm choice.

< diagram with triangle showing trade offs >

Windowing also applies to video streams: in this case, each "window" of the video is a certain number of still images—typically a single one, but some AI algorithms can potentially analyze several images at the same time.

More sophisticated techniques for dealing with streams of data fall into the category of digital signal processing. These techniques can be combined with windowing in order to create data that feeds AI algorithms.

Digital signal processing algorithms

There are hundreds of different signal processing algorithms that can help digest the signals produced by sensors. In this section we'll cover some of the DSP algorithms that are most important for edge AI.

Resampling

All time series signal have a sample rate (also known as a frequency), often described in terms of the number of data samples per second (Hz). It's often necessary to change the sample rate of a signal. For example, you might want to reduce the rate of a signal (known as downsampling) if it is producing data faster than you can process it. On the other hand, you may want to increase the rate of a signal (upsampling) so that it can be conveniently analyzed alongside another signal that has a higher frequency.

Downsampling works by "throwing away" some of the samples in order to achieve the target frequency. For example, if you threw away every other frame of a 10 Hz (10 sample per second) signal it would become a 5 Hz signal. However, due to a phenomenon called aliasing, reducing the frequency in this way can lead to distortion in the output. To help combat this, signals must have some high frequency information removed before they are downsampled. This is achieved using a high pass filter, described in the next section.

Upsampling works in the opposite way—new samples are created and inserted to increase the frequency of a signal. For example, if an extra sample was inserted after every sample in a 10 Hz signal, it would become a 20 Hz signal. The difficult part is knowing what to insert! There's no way to know what would actually have been happening during the time between two samples, but a technique known as *interpolation* can be used to fill in the blanks with an approximation.

In addition to time series, images can also be upsampled and downsampled. In this case, it's the spatial resolution (pixels per image) that is being increased or decreased. Like time series resampling, the resizing of images also requires anti-aliasing or interpolation techniques.

Both upsampling and downsampling are important, but downsampling is more common to encounter in edge AI. It's typical for sensors to produce an output at a set frequency, leaving it to the developer to downsample and obtain the frequency that best suits the rest of their signal processing pipeline.

For edge AI applications, upsampling is mostly useful if you wish two combine two signals with different frequencies into a single time series.

However, this can also be achieved by downsampling the higher frequency signal, which might be computationally cheaper.

CROPPING AND RESIZING IMAGES

Different models of image sensors output images with varying sizes and shapes, and edge AI algorithms (such as deep learning vision models) often require images of very specific sizes. Cropping and resizing is commonly used to make images compatible with models, and can involve both downsampling and upsampling, as well as throwing chunks of an image away.

The following diagram shows some common ways that images can be cropped:

< diagram showing crop styles >

Filtering

A digital filter is a function that, applied to a time series signal, transforms it in certain ways. Many different types of filters exist, and they can be very useful in preparing data for edge AI algorithms.

Low-pass filters are designed to allow low frequency elements of a signal to pass through, while removing high frequency elements. The *cutoff frequency* of the filter describes the frequency beyond which high frequency signals will be affected, and the *frequency response* describes how much those signals will be affected.

High-pass filters are the same thing in reverse, allowing frequencies *above* a cutoff frequency to pass, and attenuating those below. A band-pass filter combines the two, allowing frequencies within a certain *band* but attenuating those outside of it.

The purpose of filtering in edge AI is to isolate the useful parts of a signal, removing parts that do not contribute to solving the problem. For example, a speech recognition application could use a band-pass filter to allow frequencies in the normal range of human speech (125Hz to 8kHz) while rejecting information in other frequencies. This could make it easier for a

machine learning model to interpret the speech without being distracted by other information in the signal.

FILTERING NOISE

All signals from sensors contain some level of noise: random fluctuations in the data that happen due to slight inaccuracies in measurement. The “background hum” in audio recordings, or the speckles in a digital camera photograph taken at night, are typical examples of noise.

If the noise is present at specific frequencies, which is quite common, filters can be very useful in removing it. This can make it easier for some AI algorithms to interpret signals. However, some types of algorithms—such as deep learning models—are naturally quite resistant to noise, so it isn’t always necessary to filter it.

Filters can be applied to any type of data. For example, if a low-pass filter is applied to an image, it has a blurring or smoothing effect. If a high-pass filter is applied to the same image, it will “sharpen” details.

One type of low-pass filter is a *moving average filter*. Given a time series, it calculates a moving average of values within a certain window. In addition to smoothing the data, it has the effect of making a single value represent information from a wide range of time.

If several moving averages are calculated and stacked together, each with differing window lengths, a momentary snapshot of the signal (containing several different moving averages) contains information about changes in the signal across a window of time and a number of different frequencies. This can be a helpful technique in feature engineering, since it means an AI algorithm can observe a broad window of time using relatively few data points.

Filtering is an extremely common signal processing operation, and many embedded processors have libraries that provide software and hardware support for doing it efficiently.

Spectral analysis

A time series signal can be said to be in the “time domain”, meaning it represents how a set of variables change over time. Using some common mathematical tools, it’s possible to transform a time series signal into the “frequency domain”. The values obtained through transformation describe how much of the signal lies in various frequency bands over a range of frequencies—a spectrum.

This way of representing a signal can make it much easier for AI algorithms to understand and interpret information. For example, here’s an example of a complex signal represented in both the time and frequency domains:

< diagram >

The frequency domain of the signal in this diagram provides a simple “signature” that represents all of its constituent signals in only a handful of values, instead of hundreds of values in the original signal. This is much easier for algorithms to interpret.

By slicing a signal into multiple, thin windows and then transforming each window into the frequency domain, it’s possible to create a map of how the signal’s frequencies change over time. This map, known as a spectrogram, serves as a very effective input to machine learning models. They are commonly used in real world applications, especially around audio. It’s possible for humans to visually distinguish one word from another while looking at spectrograms—some people have even learned to read them.

< diagram of spectrogram next to the original signal >

There are many algorithms that can transform a signal from the time to the frequency domain, but the most common is the Fourier transform. It’s a very common operation, and there’s often hardware support (or at least optimized implementations) available for performing Fourier transforms on embedded devices.

There are a huge number of algorithms and techniques for digital signals processing and time series analysis; they’re major fields of engineering and study. Some great resources on the subjects are:

- The Scientist and Engineer's Guide to Digital Signal Processing, by Steven W. Smith
- Practical Time Series Analysis, by Aileen Nielsen

Image feature detection

A whole subset of signal processing algorithms are concerned with the extraction of useful features from images. These have traditionally been referred to as *computer vision* algorithms. Some common examples include:

- Edge detection: used to identify boundaries in an image
- Corner detection: used to find points in an image that have interesting two-dimensional structure
- Blob detection: used to identify regions of an image that have something in common
- Ridge detection: used to identify curves within an image

< diagram showing examples of each >

Image feature detection reduces a big, messy image into a more compact representation of the visual structures that are present within it. This can potentially make life easier for any AI algorithms that are operating downstream.

Feature detection is not always necessary when working with images. Typically, deep learning models are able to learn their own ways of extracting features, reducing the utility of pre-processing. However, it's still common to perform feature detection when interpreting image data using other types of edge AI algorithm.

The OpenCV project provides a set of libraries for feature detection (and other image processing tasks) that will run on most SoC devices. For microcontrollers, OpenMV provide an open source library of feature detection algorithm implementations along with hardware designed to run them.

Combining features and sensors

There’s nothing stopping you from combining several different features and signals as the input to your AI algorithms. For example, you could calculate several moving averages of a time series over several different windows and pass them all into a machine learning model together. There are no hard and fast rules, so feel free to experiment and be creative with the way you slice and dice your data. The following chapters will provide a framework for experimentation.

Going beyond combining features from the same signal, *sensor fusion* is the concept of integrating data from multiple sensors together. For example, an edge AI fitness tracker could combine information from an accelerometer, gyroscope, and heart rate sensor to try to detect which sport a wearer is playing.

In a more complex edge AI scenario, the sensors don’t even have to be integrated with the same device. Imagine a smart climate control system that makes use of temperature and occupancy sensors distributed throughout a building to optimize air conditioning usage.

There are three categories of sensor fusion:

- Complementary, where multiple sensors combine to deliver a more complete understanding of a situation than would be possible with a single sensor—for example, the various sensors on our hypothetical fitness tracker.
- Competitive, where multiple sensors measure the same exact thing in order to reduce the likelihood of bad measurements—for example, multiple redundant sensors monitoring the temperature of a critical piece of equipment.
- Cooperative, where information from multiple sensors combines to create a signal that was not otherwise available—for example, two cameras producing a stereo image that provides depth information.

The challenge inherent in sensor fusion is how to combine multiple signals that may even occur at different rates. You should consider the following:

1. Aligning the signals in time. For many algorithms, it's important that all of the signals we intend to fuse are sampled at the same frequency, and that the values reflect simultaneous measurements. This can be achieved through resampling—for example, upsampling a low frequency signal so that it has the same rate as the high frequency signal it is being fused with.
2. Scaling the signals. It's critical that the signals' values are on the same scale, so that a signal with typically large values does not overwhelm a signal with typically smaller ones.
3. Numerically combining the signals. This can be done using simple mathematical operations (addition, multiplication, or averaging) or with more sophisticated algorithms such as the Kalman filter (covered later)—or simply by concatenating the data together and passing it into the algorithm as a single matrix.

You can perform sensor fusion before or after other stages of feature engineering. For an arbitrary example: if you intended to fuse two time series, you might choose to run a low-pass over one of them first, then scale them to the same scale, combine the two through averaging, and transform the combined values into the frequency domain. Don't be afraid to experiment!

FEATURE SCALING

A stream of data from a sensor can have a wide range of values. For example, if the sensor returns measurements as 16 bit unsigned integers their value could be anywhere from 0 to 65,535.

Big ranges like this can make things tricky for some AI algorithms. For example, deep learning models can have a hard time training when their input values are super high.

Additionally, it can be hard to get good results from machine learning models when passing in features that have wildly different scales. The larger values outweigh the smaller ones, which reduces the benefit of having multiple input features. This is also a problem for sensor fusion.

To get around the issue, it's a very good idea to scale your inputs before combining them or sending them into AI algorithms. A common way to do this is called *normalization*. There are a few different varieties of normalization. In the simplest, known as *rescaling*, you determine the maximum and minimum values for a specific feature in a representative sample of your input data (typically using your training data, if you're working with machine learning models). You can then calculate the normalized values using the following formula:

```
normalized_value = (raw_value - minimum) / (maximum -
minimum)
```

This will provide a value between 0 and 1, which can be conveniently compared and combined with other normalized values on the same scale.

We now have some serious tools for processing data. In the next section, we'll explore the AI algorithms that will help us understand it.

Algorithms for edge AI

There are two ways to think about AI algorithms. One is based on functionality: what are they designed to do? The other is based on implementation: how do they work? Both aspects are important. Functionality is critical to the application you are trying to build, and implementation is important when thinking about your constraints—which generally means your dataset and the device you will be deploying to.

Algorithm types by functionality

First up, let's look at the most important types of algorithm from a functional perspective. Mapping the problem you are trying to solve to these algorithm types is known as *framing*, and we'll be diving deep into framing in the next chapter.

Classification

Classification algorithms try to solve the problem of distinguishing between various *types*, or *classes*, of thing. This could mean:

- A fitness monitor with an accelerometer classifying walking versus running.
- A security system with an image sensor classifying an empty room versus a room with a person present.
- A wildlife camera classifying four different species of animal

< diagram with an example of image classification >

With classification, you always need at least two classes. Even if there's only one thing you care about (for example, a person in the room), you also need a class that represents everything that you *don't* care about (for instance, rooms that don't have people in them).

Regression

Regression algorithms try to come up with numbers. This could mean:

- A smart thermostat that predicts the temperature in an hour's time.

- A virtual scale that estimates the weight of a food product using a camera.
- A virtual sensor that estimates a motor's speed of rotation based on its sound.

Virtual sensors, like the latter two examples, are a particularly interesting case of regression. They can use available sensor data to predict measurements from different types of sensor—without actually requiring those sensors to be present.

Object detection and segmentation

Object detection algorithms take an image or video and identify the locations of specific objects within them, often by drawing *bounding boxes* around them. They combine classification and regression, identifying specific types of objects and predicting their numeric coordinates.

< example of object detection output >

Specialized object detection algorithms exist for particular types of object. For example, pose estimation models are designed to recognize human body parts and identify their locations within an image:

< pose estimation image >

Segmentation algorithms are similar to object detection algorithms, but they classify images at a pixel level, creating a *segmentation map*:

< segmentation image >

Here are some example use cases for object detection and segmentation:

- A farm monitor that uses cameras to count the number of animals in a field.
- A home fitness system that gives people feedback on their form during workouts.
- An industrial camera that measures how much of a container is filled with product.

Anomaly detection

Anomaly detection algorithms recognize when a signal has deviated from its normal behavior. They are useful in many applications:

- An industrial predictive maintenance system that can recognize when a motor has started to break down by its current draw.
- A robot vacuum that can identify when it is driving on an unusual surface using an accelerometer.
- A trail camera that knows when an unknown animal has walked past.

Anomaly detection algorithms are very useful for predictive maintenance. They're also very helpful when paired with machine learning models. Many machine learning models will produce spurious, random results if they are presented with an input that isn't in their training set. To avoid this, an ML model can be paired with an anomaly detection algorithm that tells it when something is *out of distribution* so that its spurious results can be discarded.

Clustering

Clustering algorithms try to group inputs by similarity, and can recognize when an input is not similar to what it has seen before. They are often used when an edge AI device needs to learn from its environment, including for anomaly detection applications. For example, consider:

- A voice assistant that learns which voice belongs to each of its users.
- A predictive maintenance application that learns a "normal" state of operation and can detect deviations from it.
- A vending machine that can recommend drinks based on a user's previous choices.

A clustering algorithm can either learn its clusters on the fly (after deployment) or have them configured ahead of time.

Dimensionality reduction

Dimensionality reduction algorithms take a signal and produce a representation of it that contains equivalent information but takes up a lot less space. The representations of two signals can then be compared to one another easily. Here are some example applications:

- Compression of audio, to make it cheaper to transmit sounds from a remote device.
- Fingerprint recognition, ensuring a fingerprint matches the owner of a device.
- Facial recognition, recognizing individual faces in a video feed.

Dimensionality reduction tends to be used alongside other AI algorithms, as opposed to being used on its own. For example, it can be used in conjunction with a clustering algorithm to identify similar signals in complex data types, like audio and video.

Transformation

Transformation algorithms take one signal and output another. Here are some examples:

- Noise cancelling headphones that identify and remove specific noises in a signal.
- A car reversing camera that enhances the image in dark or rainy conditions.
- A speech recognition device that takes an audio signal and outputs a transcription.

< image of audio in, text out >

The input and output of transformation algorithms can be extremely different. In the case of transcription, the input is a stream of audio data and the output is a sequence of words.

COMBINING ALGORITHMS

There's no reason you can't mix different types of algorithms in the same application. Later in this section we'll explore techniques for "Combining algorithms".

Algorithm types by implementation

Exploring algorithms by functionality helps us understand what they are used for, but from an engineering perspective it's important to get a sense for the different ways these functionalities can be implemented. There are hundreds of different ways to build a classification algorithm, for example, resulting from decades of computer science research. Each method has its own unique strengths and weaknesses which are amplified by the constraints posed by edge AI hardware.

In the following section we'll explore the most important ways that edge AI algorithms are implemented. Bear in mind that this isn't an exhaustive list—we're focused on edge AI, so we're focused on technologies that work well on-device.

Conditionals and heuristics

The simplest type of AI algorithms are based on conditional logic: simple `if` statements that result in decisions. Let's look back at the code snippet we explored in "Artificial intelligence (A.I.)":

```
current_speed = 10 # In meters per second
distance_from_wall = 10 # 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();
}
```

This simple algorithm does a basic calculation using some human-defined values (`seconds_to_stop`, etc.) and makes a decision whether to apply a car's brakes. Does this count as AI? It's a question that might stimulate debate—but the answer is emphatically yes.[1]

The common understanding of artificial intelligence is that it's a quest to create machines that can think like human beings. The engineering definition is much more realistic: AI allows computers to do tasks that typically require human intelligence. In this case, controlling a car's brakes to avoid a collision is definitely something that has typically required human intelligence. It would have been considered extremely impressive twenty years ago, but automatic braking is a common feature in modern vehicles.

> **NOTE**
>
> Before you laugh at the idea that `if` statements can be artificial intelligence, consider that *decision trees*—one of the most popular and effective categories of machine learning algorithm—are just `if` statements under the hood. These days, even deep learning models can be implemented as binary neural networks, which are essentially conditional logic. Intelligence comes from the application, not the implementation!

The conditional logic in our car braking algorithm is actually an implementation of classification. Given an input (the speed of the car and the distance from a wall), the algorithm classifies the situation into one of two types: safe driving, or impending crash. Conditional logic is naturally used for classification, since its output is categorical; an `if` statement gives us either one output or another.

Conditional logic is connected to the idea of *heuristics*. A heuristic is a hand-crafted rule that can be applied to a situation in order to help understand or react to it. For example, our car braking algorithm uses the heuristic that if we have less than 4 seconds before hitting a wall, we should apply the brakes.

Heuristics are designed by human beings, using domain knowledge. This domain knowledge can be built on data that has been collected about a real world situation. In that respect, our seemingly simple car braking algorithm might actually represent some deep, well-researched understanding of the real world. Perhaps the value of `seconds_to_stop` was arrived at after millions of dollars worth of crash tests, and represents the ideal value for the constant. With this in mind, it's easy to see how even an `if` statement can represent a significant amount of human intelligence and knowledge, captured and distilled into a simple and elegant piece of code.

Our car braking example is very simple—but when paired with signal processing, conditional logic can make some quite sophisticated decisions. For example, imagine you are building a predictive maintenance system that aims to alert workers of the health of an industrial machine based on the sounds it makes. Perhaps the machine makes a characteristic high-pitched whine when it is about to break down. If you capture audio and translate it into the frequency domain using a Fourier transform, you can use a simple `if` statement to determine when the whine is happening and let the workers know.

Beyond `if` statements, you can use more complex logic to interpret situations based on known rules. For example, an industrial machine may use a hand-coded algorithm to avoid damage by varying its speed based on measurements of internal temperature and pressure. The algorithm might take the temperature and pressure and directly calculate an RPM, using human insight that is captured in the code.

If it works for your situation, conditional logic and other hand-coded algorithms can be amazing. It is easy to understand, easy to debug, and easy to test. There's no risk of unspecified behavior: the code either branches one way or another, and all paths can be exercised with automated tests. It runs incredibly fast, and will work on any imaginable device.

There are two major downsides of heuristics. Firstly, developing them may require significant domain knowledge and programming expertise. Domain knowledge is not always available—for example, a small company might not have the resources to conduct the expensive research necessary to understand the fundamental mathematical rules of a system. In addition, even given domain knowledge, not everyone has the expertise required to design and implement a heuristic algorithm in efficient code.

The second big downside is the idea of *combinatorial explosion*. The more variables that are present in a situation, the more difficult it is to model with traditional computer algorithms. A good example of this is the game of chess: there are so many pieces, and so many possible moves, that deciding what to do next requires a vast amount of computation. Even the most advanced chess computers built using conditional logic can easily be beaten by expert human players.

Some edge AI problems are *far* more complex than games of chess. For example, imagine trying to hand-write conditional logic that can determine whether a camera image shows a cat or a dog. With some tricks (*"dog photos often have a green, grassy background"*) you might succeed for some categories of images—but it would be impossible to make it generalize.

A good rule of thumb for hand-coded logic is that the more data values you have to deal with, the more difficult it is going to be to get a satisfactory solution. Fortunately, there are plenty of algorithms that can step in when a hand-coded approach fails.

Classical machine learning

Machine learning is a special approach to creating algorithms. Where heuristic algorithms are created by hand-coding logic based on known rules, machine learning algorithms discover their own rules—by exploring large amounts of data.

The following description, taken from the book TinyML, introduces the basic ideas behind machine learning.

> *To create a machine learning program, a programmer feeds data into a special kind of algorithm and lets the algorithm discover the rules. This means that as programmers, we can create programs that make predictions based on complex data without having to understand all of the complexity ourselves. The machine learning algorithm builds a model of the system based on the data we provide, through a process we call training. The model is a type of computer program. We run data through this model to make predictions, in a process called inference.*
>
> —TinyML, O'Reilly 2019

Machine learning algorithms can perform all of the functional tasks described earlier in this chapter, from classification to transformation. The key requirement for using machine learning is that you have a *dataset*. This is a large store of data, generally collected under real-world conditions, that is used to train the model.

Typically, the data needed to train a machine learning model is gathered during the development process, aggregated from as many sources as possible. As we'll see in later chapters, a large and varied dataset is critical for working with edge AI—but especially machine learning.

Since machine learning depends on large datasets, and because *training* a machine learning model is computationally expensive, the training part generally happens before deployment, with *inference* happening on the edge. It's certainly possible to train machine learning models on-device, but the lack of data combined with the small amount of compute make it a challenge.

In edge AI, there are two main ways to work with machine learning datasets:

- Supervised learning, where the dataset has been *labelled* by an expert to assist the machine learning algorithm in understanding it.
- Unsupervised learning, where the algorithm identifies structures in the data without human help.

Machine learning has a major dataset-related drawback. ML algorithms depend entirely on their training data to know how to respond to inputs. As

long as they are receiving inputs that are similar to their training data, they should work well. However, if they receive an input that is significantly dissimilar from their training dataset—known as an *out-of-distribution* input—they will produce an output that is completely useless.

The tricky part is that there is no obvious way of telling, from the output, that an input was out-of-distribution. This means that there's always a risk that a model is providing useless predictions. Avoiding this problem is a core concern when working with machine learning.

There are many different types of machine learning algorithms. *Classical* machine learning encompasses the vast majority of them used in practice, with the major exception of *deep learning* (which we'll explore in the next section).

INTERPRETABILITY AND EXPLAINABILITY

When a machine learning model makes a prediction, it's great if we can also understand *why* it made that particular prediction—as opposed to a different one. The property of making human-comprehensible decisions is known as *interpretability* or *explainability*.

Some machine learning algorithms are more interpretable than others. Whether this is important or not depends on your use case. For example, if a machine learning model is being used to assist with medical diagnosis, doctors may trust it more if it can explain its predictions.

Interpretable algorithms are easier to work with, since debugging them is straightforward—if they produce an incorrect output, you can directly understand why and attempt to address it.

Here are some of the most useful types of classical ML algorithm for edge AI:

Regression analysis

Learns the mathematical relationships between input and output to predict a continuous value. Easy to train, fast to run, low data

requirements, and highly interpretable, but can only learn simple systems.

Logistic regression

A classification-oriented type of regression analysis, logistic regression learns the relationship between input values and *categories* of output—for relatively simple systems.

Support vector machine

Uses fancy mathematics to learn much more complex relationships than basic regression analysis. Low data requirements, fast to run, can learn complex systems, but difficult to train and low interpretability.

Decision trees and random forests

Uses an iterative process to construct a series of `if` statements that predict an output category or value. Easy to train, fast to run, highly interpretable, can learn complex systems, but may require a lot of training data.

Kalman filter

Predicts the next datapoint given a history of measurements. Can factor in multiple variables to improve precision. Often trained on device, low data requirements, fast to run, easy to interpret, but can only model relatively simple systems.

Nearest neighbors

Classifies data by how similar it is to known datapoints. Often trained on device, low data requirements, easy to interpret, but can only model relatively simple systems and can be slow with lots of datapoints.

Clustering

Learns to group inputs by similarity but does not require labels. Often trained on device, low data requirements, fast to run, easy to interpret, but can only model relatively simple systems.

Classical ML algorithms are an incredible set of tools for interpreting the output of your feature engineering pipeline and making decisions with data. They cover the spectrum from highly efficient to highly flexible, and they can perform many functional tasks. Another major benefit is that they tend to be very explainable—it's easy to understand how they are making their decisions. And depending on the algorithm, the data requirements can be quite low (deep learning typically requires very large datasets).

The diverse pool of classical ML algorithms (there are literally hundreds) are both a blessing and a curse for edge AI. On one hand, there are algorithms well suited to many different situations, which makes it possible to find one that is—theoretically—ideal for a particular use case. On the other hand, the large constellation of algorithms can be challenging to explore.

While libraries like scikit-learn make it easy to try out many different algorithms, there's an art and a science to tuning each one to perform optimally, and to interpreting their results. In addition, if you're hoping to deploy to a microcontroller, you may have to write your own efficient implementation of an algorithm—there are not many open source versions available yet.

A major downside of classical ML algorithms is that they run into a relatively low ceiling in terms of complexity of the systems they can model. This means that to get the best results, they often have to be paired with heavy feature engineering—which can be complex to design, and computationally costly. Even with feature engineering, there are some tasks —such as the classification of image data—where classical ML algorithms just don't perform well.

That said, classical ML algorithms are a fantastic set of tools for making on-device decisions. But if you hit their limitations, deep learning might help.

Deep learning

Deep learning is a type of machine learning that focuses on neural networks. These have proven such an effective tool that deep learning has

grown into a gigantic field, with deep neural networks being applied to every possible type of application.

> **NOTE**
>
> This book focuses on the important properties of deep learning algorithms from an engineering perspective. The underlying mechanics of deep learning are interesting, but they're not required knowledge for building an edge AI product. Using modern tools, any engineer can deploy deep learning models without a formal background in machine learning. We'll share some of the tools for doing that in the tutorial chapters later on.

Deep learning shares the same principles as classical ML. A dataset is used to train a model, which can be implemented on a device to perform inference. There isn't anything magical about a model—it's just a combination of an algorithm and a collection of numbers that are fed into it, along with the model's input, in order to produce the desired output.

The numbers in the model are called *weights*, or *parameters*, and they're generated during the training process. The term *neural network* refers to the way that the model combines its input with its parameters, which was inspired by the way neurons in an animal brain connect to one another.

Many of the most mind-blowing feats of AI engineering that we've seen over the past decade have made use of deep learning models. Here are some popular highlights:

- AlphaGo, a computer program that used deep learning to beat the best players at *https://en.wikipedia.org/wiki/Go_(game)Go*, an ancient game once thought impossible for computers to master.
- GPT-3, a model that can generate written language that is indistinguishable from human writing.
- Fusion reactor control, using deep learning to control the shape of plasma within a fusion reactor.

- Dall-E, a model that can generate realistic images and abstract art based on text prompts.
- Tesla Autopilot, a consumer-ready self-driving system that can drive and park a car.
- GitHub Copilot, software that assists software engineers by automatically writing code.

Beyond the fancy stuff, deep learning excels at all of the tasks in our list of "Algorithm types by functionality". It has proven to be flexible, adaptable, and an incredibly useful tool in allowing computers to understand and influence the world.

Deep learning models are effective because they work as *universal function approximators*. It's been mathematically proven that, as long as you can describe something as a continuous function, a deep learning network can model it. This basically means that for any dataset that shows various inputs and desired outputs, there's a deep learning model out there that can convert one into the other.

< diagram of a few different inputs, outputs, and models >

A really exciting result of this ability is that during training, deep learning models can figure out how to do their own feature engineering. If a special transformation is needed to help interpret the data, a deep learning model can potentially learn how to do it. This doesn't make feature engineering obsolete, but it definitely reduces the burden on the developer to get things exactly right.

The reason deep learning models are so good at approximating functions is that they can have very large numbers of parameters. With each parameter, the model gets a little bit more flexibility, allowing it to describe a slightly more complex function.

This property leads to the two major drawbacks of deep learning models. Firstly, finding the ideal values for all of these parameters is a difficult process. It involves training a model with lots of data. Data is often a rare and precious resource, difficult and expensive to obtain, so this can be a

major obstacle. Fortunately, there are many techniques that can help make the most of limited data—we'll cover them later in the book.

The second major drawback is the risk of *overfitting*. Overfitting is when a machine learning model learns a dataset *too* well. Instead of modeling the general rules that lead from outputs to inputs in its dataset, it memorizes the dataset completely. This means that it won't perform well on data that it hasn't seen before.

Overfitting is a risk with all machine learning models, but it's especially a challenge for deep learning models because they can have so many parameters. Each additional parameter provides the model with slightly more ability to memorize its dataset.

There are a lot of different types of deep learning models. Here are some of the most important for edge AI:

Fully connected models

> The simplest type of deep learning model, fully connected models consist of stacked *layers* of *neurons*. The input of a fully connected model is fed directly in as a long series of numbers. Fully connected models are capable of learning any function, but they are mostly blind to spatial relationships in their inputs (for example, which values in an input are next to one another).
>
> In an embedded context, this means they work well for discrete values (for example, if the input features are a set of statistics about a time series) but they aren't as great with raw time series or image data.
>
> Fully connected models are very well supported on embedded devices, with hardware and software optimizations commonly available.

Convolutional models

> Convolutional models are designed to make use of the spatial information in their inputs. For example, they can learn to recognize shapes in images, or the structures of signals within time series sensor data. This makes them extremely useful in embedded applications, since spatial information is important in so many of the signals we deal with.

Like fully connected models, convolutional models are very well supported on embedded devices.

Sequence models

Sequence models were designed originally for use on sequences of data, like time series signals or even written language. To help them recognize long-term patterns in time series, they often include some internal “memory”.

It turns out that sequence models are very flexible, and there’s increasing evidence that they can be very effective on any signal where spatial information is important. Many people believe they will eventually take over from convolutional models.

Sequence models are currently less well supported than convolutional and fully connected models on embedded devices; there are few open source libraries that provide optimized implementations for them. This is more due to inertia than technical limitations, so the situation is likely to change over the next couple of years.

Embedding models

An embedding model is a pre-trained deep learning model that is designed for dimensionality reduction—it takes a big, messy input and represents it as a smaller set of numbers that describes it within a certain context. They are used in the same way a signal processing algorithm would be: they produce features that can be interpreted by another ML model.

Embedding models are available for many tasks, from image processing (turning a big messy image into a numeric description of its contents) to speech recognition (turning raw audio into a numeric description of the vocal sounds within it).

The most common use for embedding models is *transfer learning*, which is a way of reducing the amount of data required to train a model. We’ll learn more about that later.

Embedding models can be fully connected, convolutional, or sequence models, so their support on embedded devices varies—but convolutional embedding models are the most common.

MODEL ARCHITECTURES

Deep learning models are flexible and modular—they are composed of *layers* and *operations* that can be stacked and combined in an infinite number of ways.

Different arrangements are known as *architectures*, and many architectures have been designed that are optimized for different tasks. You'll often see references to deep learning model architectures in online articles and scientific literature.

Some noteworthy architectures for edge AI include:

- MobileNet and EfficientNet, families of architectures designed to run efficiently on mobile devices.
- YOLO, a family of architectures designed to perform object detection.
- Transformers, a family of architectures designed to translate between sequences of data.

It's only in recent years that deep learning models have been brought to edge AI hardware. Since they are often large and involve significant computation to run, it's been the advent of high end MCUs and SoCs with relatively powerful processors and large amounts of ROM and RAM that have enabled them to make the leap.

It's possible to run a small deep learning model using just a few kilobytes of memory, but for models that do more complex things—from audio classification to object detection—it is common for models to require dozens or hundreds of kilobytes as a minimum.

This is already impressive, since traditional server-side machine learning models can be anywhere from tens of megabytes to several terabytes in

size. Using clever optimization, and by limiting scope, embedded models can be made much smaller—we'll introduce some of these techniques shortly.

There are various ways to run a deep learning model on an embedded device. Here's a quick summary:

Interpreters

Deep learning interpreters, like TensorFlow Lite for Microcontrollers, use an interpreter execute a model that is stored as a file. They are flexible and easy to work with, but they come with some computational and memory overhead, and they don't support every type of model.

Code generation

Code generation tools, like EON, take a trained deep learning model and translate it into optimized embedded source code. This is more efficient than an interpreter-based approach, and the code is human-readable so can still be debugged, but it still doesn't support every possible model type.

Compilers

Deep learning compilers, like microTVM, take a trained model and generate optimized bytecode that can be included into embedded applications. The implementation they generate can be highly efficient, but it's not as easy to debug and maintain as actual source code. They can support model types not explicitly supported by interpreters and code generation. It's common for embedded hardware vendors provide custom interpreters or compilers to assist with running deep learning models on their hardware.

Hand-coding

It's possible to implement a deep learning network by writing code by hand, incorporating the parameter values from a trained model. This is a difficult and time consuming process, but it allows full control over optimization and allows you to support any model type.

The environment for deploying deep learning models is very different between SoCs and microcontrollers. Since SoCs run full modern operating systems, they also support most of the tools that are used to run deep learning models on servers. This means that pretty much any type of model will run on a Linux SoC. That said, the latency of the model will vary depending on the architecture of the model and the SoC's processor.

There are also interpreters designed specifically for SoC devices. For example, TensorFlow Lite provides tools that allow deep learning models to be run more efficiently on SoCs—typically those that are used in smartphones. They include optimized implementations of deep learning operations that make use of features available in some SoCs, such as GPUs.

The SoCs that have integrated deep learning accelerators are a special case. Typically, the hardware vendor will provide a special compiler or interpreter that allows the model to make use of hardware acceleration. Accelerators typically only accelerate certain operations, so the amount of speed-up depends on the architecture of the model.

Since microcontrollers don't run full operating systems, the standard tools for running deep learning models aren't available. Instead, frameworks like TensorFlow Lite for Microcontrollers provide a baseline of model support. They tend to lag behind the standard tools a little in terms of operator support, meaning they will not run some model architectures.

Typical high-end microcontrollers have hardware features such as SIMD instructions that will drastically improve the performance of deep learning models. TensorFlow Lite for Microcontrollers includes optimized implementations of operators, making use of these instructions, for several vendors. Like with SoCs, the vendors of microcontroller-based hardware accelerators often provide custom compilers or interpreters that allow models to run on their hardware.

The core advantages of deep learning are its flexibility, reduced requirements for feature engineering, and ability to make use of large amounts of data due to the high parameter counts of models. Deep learning is noteworthy for its ability to model complex systems, going beyond simple prediction to perform tasks such as generating art and accurately

recognizing objects in images. Deep learning provides so much freedom that researchers have only just begun to explore its potential.

The core disadvantages are its high data requirements, its propensity towards overfitting, the relatively large size and computational complexity of deep learning models, and the complexity of the training process. Additionally, deep learning models can be hard to interpret—it can be challenging to explain why they make one prediction over another. That said, there are tools and techniques that help mitigate most of these drawbacks.

WHY NOT ALWAYS USE DEEP LEARNING?

Since deep learning is so capable, you may wonder why we would use any other deep learning algorithms. Deep learning is a powerful general-purpose tool that can model pretty much any relationship between input and output variables. However, just because it *can* doesn't mean it's always the *best* at it. Depending on the situation, classical ML algorithms can outperform deep learning in terms of:

- Explainability. Nothing beats the interpretability of a decision tree, if your use case allows it.
- Efficiency. Classical ML algorithms are typically much easier to compute than deep learning models.
- Portability. Since they are simpler, classical ML algorithms can be deployed to the most basic devices (such as low-end MCUs).
- Effectiveness. Some classical algorithms work better than deep learning in certain situations, especially when there is not much data available.
- On-device training. Deep learning training is difficult to perform on-device, while some classical algorithms are easy to train in the field.

It all comes down to your individual use case. That said, if you were only going to take a deep dive into one technique for edge AI algorithm development then it would probably make sense to choose deep learning.

Combining algorithms

A single edge AI application can make use of multiple different types of algorithm. Here are some typical ways this is done:

Ensembles

An *ensemble* is a collection of machine learning models that are fed the same input. Their outputs are combined mathematically in order to make a decision. Since every ML model has its own strengths and weaknesses, an ensemble of models is often more accurate together than its constituent parts. The downside of ensembles is the additional complexity, memory, and compute required to store and run multiple models.

Cascades

A *cascade* is a set of ML models that are run in sequence. For example, in a cellphone with a built-in digital assistant, a small, lightweight model is run constantly to detect any signs of human speech. Once speech is detected, a larger, more computationally expensive model is woken up in order to determine what was said.

Cascades are a great way of saving energy since they allow you to avoid unnecessary computation. In a heterogeneous compute environment, where multiple types of processor are available, the individual components of a cascade can even be run on different processors.

Feature extractors

As we learned earlier, embedding models take a highly dimensional input, like an image, and distil it down to a set of numbers that describe its content. The output of an embedding model can be fed into another model, designed to make predictions based on what the embedding model describes about the original input. In this case, the embedding model is being used as a *feature extractor*.

If a pre-trained embedding model is used, this technique—known as *transfer learning*—can massively reduce the amount of data required to train a model. Instead of learning how to interpret the original highly

dimensional input, the model only needs to learn how to interpret the simple output returned by the feature extractor.

Many pre-trained deep learning feature extractors are available under open source licenses. They are commonly used for image related tasks.

Multi-modal models

A *multi-modal model* is a single model that takes inputs of multiple types of data simultaneously. For example, a multi-modal model might accept both audio and accelerometer data together. This technique can be used as a mechanism for sensor fusion, using a single model to combine disparate data types.

Post-processing algorithms

On edge AI devices, we typically work with streams of data—for example, a continuous time series of audio data. When we run an edge AI algorithm on that stream of data, it will produce a second time series that represents the outputs of the algorithm over time.

< diagram showing both streams >

This poses a problem. How do we interpret this second time series in order to make a decision? For example, imagine we are analyzing audio to detect when somebody says a keyword so that we can trigger some functionality on a product. What we *really* want to know is “when did we hear the keyword?”.

Unfortunately, the time series of inference results is not ideal for this purpose. Firstly, it contains many events that do not represent a keyword being detected. To clean these up, we can remove any whose confidence that a keyword was spotted is below a certain threshold.

< diagram showing removing events with low confidence >

Secondly, the model may occasionally (and briefly) detect a keyword when a keyword was not actually spoken. We need to filter out these blips to clean up our output. This is equivalent to running a low-pass filter on the time series.

< diagram showing blips being removed >

Finally, instead of telling us *each* time the keyword was spoken, the raw time series tells us at a set rate whether the keyword is *currently* being spoken. This means we need to do some output gating to get the information we really want.

< diagram showing filtering down to a single event per utterance >

After cleaning up the raw output, we now have a signal that tells us when a keyword was actually spotted. This is something we can use in our application logic to control our device.

This sort of post-processing is extremely common in edge AI applications. The exact post-processing algorithm used, and its particular parameters (for example, the threshold for considering something a match), can be determined on a case by case basis. Tools like Edge Impulse's Performance Calibration (link when available) allow developers to automate discovery of the ideal post-processing algorithm for their application.

< screenshot of performance calibration >

Fail-safe design

There are many things that can go wrong with an edge AI application, so it's critical that there are always safeguards in place to protect against unexpected issues.

For example, imagine a wildlife camera that uses a deep learning model to identify when an animal of interest has been photographed and uploads the animal's image via a satellite connection. Under normal operation, it may send a few photographs a day—not costing very much in data fees.

But out in the field, a physical problem with the camera hardware—such as dirt or reflections on the lens—might result in images being taken that are very different from those in the original training dataset. These out-of-distribution images could lead to unspecified behavior from the deep learning model—which could mean that the model begins to constantly report that the animal of interest is present.

These false positives, caused by out-of-distribution inputs, might result in hundreds of images being uploaded via satellite connection. Not only would the camera be rendered useless, but it could potentially cost large amounts in data transfer fees.

In real world applications, there's no way to avoid things like damage to sensors, or unexpected behavior from algorithms. Instead, it's important that you design your application to be fail-safe. This means that if part of the system were to fail, the application would minimize harm.

The best way to do this varies between situations. In the case of a wildlife camera, it could be wise to build in a rate limit that kicks in if an unreasonable number of photographs are being uploaded. In another application, you might shut a system down entirely rather than risk harm being caused.

Building fail-safe applications is an important part of ethical AI—and good engineering in general. It's something to think about from the very beginning of any project.

Optimization for edge devices

With machine learning models, and particularly deep learning models, there's often a trade-off between how well a model performs its task and how much memory and compute the model requires.

This trade-off is extremely important for edge AI. Edge devices are typically computationally constrained. They are designed to minimize cost and energy usage, not to maximize compute. At the same time they are expected to deal with real-time sensor data, often at high frequencies, and potentially react in real time to events in the data stream.

Finding the correct balance between *task performance* and *computational performance* is essential in any application. It's a matter of juggling constraints. On the one hand, there's a minimum standard for performance at a given task. On the other hand, hardware choices create hard limits on available memory and latency.

Later chapters will provide a framework for thinking about this trade-off. For now, let's explore some of the options we have available for making

models work well on-device.

Choice of algorithm

Every edge AI algorithm has a slightly different profile of memory usage and computational complexity. The constraints of your target hardware should inform your choice of algorithm. Typically, classical ML algorithms are smaller and more efficient than deep learning algorithms. However, it's commonly the case that feature engineering algorithms use vastly more compute than either, making the choice between classical ML and deep learning less significant. The exception to this rule is the analysis of image data, which typically requires little feature engineering but relatively large deep learning models.

Here are some common ways to reduce the latency and memory required by your choice of algorithms:

- Reduce the complexity of feature engineering. More math means higher latency.
- Reduce the amount of data that reaches the AI algorithm.
- Use classical ML instead of deep learning.
- Trade complexity between feature engineering and machine learning model, depending on which runs more efficiently on your device.
- Reduce the size (the number of weights and layers) of deep learning models.
- Choose model types that have accelerator support on your device of choice.

Data and model compression

There are many optimization techniques designed to reduce the amount of data and computation required by a given algorithm. Here are some of the most important types:

Quantization

One way to reduce the amount of memory and computation required by an algorithm or model is to decrease the precision of its numeric representations. As mentioned in "How are values represented?", there are many different ways to represent numbers in computation—some that have more precision than others.

Quantization is the process of taking a set of values and reducing their precision while preserving the important information they contain. It can be done for both signal processing algorithms and ML models. It's especially useful for deep learning models, which by default tend to have 32-bit floating point weights. By reducing the weights to 8-bit you can reduce a model to 1/4th its size—typically without much reduction in accuracy.

Another advantage of quantization is that the code to perform integer math is faster and more portable than the code for floating point math. This means that quantization results in a significant speedup on many devices, and that quantized algorithms will run on devices that lack floating point units.

Pruning

Pruning is a technique applied during the training of a deep learning model. It forces many of the model's weights to have a value of zero, creating what is known as a *sparse* model. In theory, this should allow for faster computation, since any multiplication involving a zero weight will invariably result in a zero.

However, at this point in time there is very little edge AI hardware and software designed to take advantage of sparse weights. This will change over the next few years, but for now the main benefit of pruning is that sparse models are easier to compress, due to their large blocks of identical values. This is helpful when models need to be sent over-the-air.

Knowledge distillation

Knowledge distillation is a deep learning training technique that enables a large "teacher" model to help train a smaller "student" model to

reproduce its functionality. It takes advantage of the fact that there is typically a lot of redundancy in the weights of a deep learning model, meaning that it's possible to find an equivalent model that is smaller but performs almost as well.

Knowledge distillation is a bit fiddly, so it's not yet a common technique—but it's likely to become a best practice over the next few years.

Binary neural networks

Binary neural networks are deep learning models where every weight is a single binary number. Since binary arithmetic is extremely fast on computers, binary neural networks can be very efficient to run. However, they are a relatively new technology and the tooling for training and running inference with them is not yet in broad use.

On-device training

In the vast majority of cases, machine learning models used in edge AI are trained before being deployed to a device. Training required large amounts of data, typically annotated with labels, and involves significant computation—the equivalent of hundreds or thousands of inferences per datapoint. This limits the utility of on-device training, since by nature edge AI applications are subject to severe constraints in memory, compute, energy, and connectivity.

That said, there are a few scenarios where on-device training makes sense. Here's an overview:

Predictive maintenance

A common example of on-device training happens in predictive maintenance, when a machine is being monitored to determine whether it is functioning normally. A small on-device model can be trained with data that represents a "normal" state. If the machine's signals start to deviate from that baseline, the application can notice and take action.

This use case is only possible when it can be assumed that abnormal signals are rare, and that at any given moment the machine is likely to

be operating normally. This allows the device to treat the data being collected as having an implicit “normal” label. If abnormal states were common, it would be impossible to make assumptions about the state at any given moment.

Personalization

Another example where on-device training makes sense is when a user is asked to deliberately provide labels. For example, some smartphones use facial recognition as a security method. When the user sets up the device, they are asked to enrol images of their face. A numeric representation of these facial images is stored.

These types of applications tend to use carefully designed embedding models that convert raw data into compact numeric representations of their content. The embeddings are designed in such a way that the Euclidean distance between two embeddings corresponds to the similarity between them. In our face recognition example, this makes it easy to determine whether a new face matches the representations stored during set-up: the distance between the new face and the enroled faces is calculated, and if it is sufficiently close then the faces are considered the same.

This form of personalization works well because, typically, the algorithm used to determine embedding similarity can be very simple; either a distance calculation or a nearest neighbors algorithm. The embedding model has done all the hard work.

Implicit association

A further example of on-device training is when labels are available by association. For example, battery management features such as Apple’s Optimized Battery Charging train models on-device to predict what time a user is likely to be using their device. One way to do this would be to train a forecasting model to output a probability of usage at a specific time, given a log of the previous few hours’ usage.

In this case, it’s easy to collect and label training data on a single device. Usage logs are collected in the background, and labels are

applied according to some metric (such as whether the screen was activated). The implicit association between time and log content allow the data to be labelled. A simple model can then be trained.

Federated learning

One of the obstacles to training on-device is a lack of training data. In addition, on-device data is often private and users are not comfortable with it being transmitted. Federated learning is a way of training models in a distributed manner, across many devices, while preserving privacy. Instead of raw data being transmitted, partially trained models are passed around between devices (or between each device and a central server). The partially trained models can be combined and distributed back to devices once they are ready.

Federated learning often seems attractive, since it appears to provide a way for models to learn and improve while in the field. However, it has some serious limitations. It is computationally expensive and requires large amounts of data transfer, which runs counter to the core benefits of edge AI. The training process is very complex, and requires both on-device and server-side components, which increases project risk.

Since data is not stored globally, there is no way to validate that the trained model is performing well across the entire deployment. The fact that models are uploaded from local devices presents a vector for security attacks. Finally, and most importantly, it does not solve the problem of labels. If labelled data is not available, federated learning is useless.

Over-the-air updates

Although not actually an on-device training technique, the most common way to update models in the field is via over-the-air updates. A new model can be trained in the lab, using data collected from the field, and distributed to devices via firmware updates.

This depends on network communication, and it doesn’t solve the problem of obtaining labelled data, but it’s the most common way to keep models up-to-date over time.

Summary

This chapter has introduced the core technologies that define artificial intelligence on the edge. In the following chapters, we'll build on this knowledge to assemble a workflow that will help you develop successful applications of your own.

---

1 There's a well-documented phenomenon known as the "AI effect", where the moment AI researchers figure out how to make a computer do a task, critics no longer consider that task representative of intelligence. *https://en.wikipedia.org/wiki/AI_effect*

Chapter 4. Understanding and Framing Problems

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book. Please note that the GitHub repo will be made active later on.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the authors at dan@situnayake.com and jplunkett@utexas.edu.

The next five chapters provide a roadmap for working with edge AI. We'll establish best practices for:

- Viewing the problems you want to solve through the lens of edge AI
- Building datasets that allow you to you train models and evaluate algorithms
- Designing applications that make use of edge AI technologies
- Developing effective applications through an iterative process
- Testing edge AI applications and monitoring them in the field

For this chapter in particular, we'll start by introducing a high level, general workflow for edge AI projects. This should give you a sense of how everything will fit together. After that we'll learn how to evaluate projects to make sure they are a good fit for edge AI, then walk through the process

of identifying which types of algorithms and hardware make sense for a given problem—and start to think about planning our implementation.

The edge AI workflow

Like any sophisticated engineering project, a typical edge AI project involves multiple tracks of work, some of which run in parallel. The following diagram shows the stages:

The process can be split roughly into two chunks—labelled in the diagram as *Discover* and *Test and iterate*. The first chunk, *discover*, involves developing a deep understanding of the problem you are trying to solve, the resources you have at your disposal, and the space of possible solutions. This is where you do the up-front work of figuring out what you would like (and what is realistic) to achieve.

The second chunk, *test and iterate*, is a continual process of refinement that takes you from initial prototype through to a production-ready application. It spans time before and after development—in machine learning, your application is never truly *finished* but needs to be monitored, supported, and iterated on after being deployed in the field. This continual improvements happens across all parts of your project in parallel—application, dataset, algorithms, and hardware.

The most important part of this process is the *feedback loop* that enables continuous improvement. The more feedback you can create between different aspects of your project, the more successful your project will be. For example, the results of your model's performance on different types of data can be fed back into the data gathering process, helping you build a diverse and representative dataset that covers the entire space of potential inputs.

<diagram of feedback loops>

We'll be covering this whole workflow over the next few chapters. The following section provides an index from workflow stages to the relevant pages of this book:

> **NOTE**
>
> Core to the success of any technology project (and arguably any project in general) is the task of managing risk. Edge AI projects are uniquely risky, thanks to their combination of hardware and software and their dependence on complex algorithms and data-driven development.
>
> At each stage in the workflow we'll learn techniques you can use to keep risk to a minimum and improve your chances of success.

Ethical AI in the edge AI workflow

As we've learned, AI applications are especially vulnerable to ethical issues—and many types of ethical issues can lead to unexpectedly poor performance in the real world. This makes ethical analysis a critical part of the edge AI development workflow.

It isn't enough to do a single ethical review at the start of a project, or a final one at the end. Since new information will come to light over the course of a project, and many decisions will be taken that have downstream effects, ethical analysis needs to be happening at every stage along the way, giving you time to course-correct if required.

In this book we'll be integrating ethical consideration into every step of the process. You shouldn't think of this as an optional extra—it's a part of the core engineering and product management work that is necessary for a successful project. The nightmare scenario for teams working with edge AI is that issues are discovered only after a system has been deployed to production. Nobody wants to be responsible for a product recall!

By applying ethical analysis throughout the development process we'll maximize our ability to catch issues before they reach production and increase the quality of our work.

Do I need edge AI?

Artificial intelligence and edge compute are both sophisticated technologies, each involving an entire landscape of considerations. Working

with either of them involves making trade-offs between capability and complexity. For many projects, the burden of complexity may outweigh the benefits in capability that come from working with them.

With this in mind, for any potential application it's very important to try to understand whether the risk is worth the reward. The answer depends heavily on context, including elements such as:

- The specific requirements of the application
- The skills of the team that will be building it
- The available budgets for engineering, data collection, and long term support
- The amount of time available for delivery

In the following sections we'll break down the questions that we need to ask in order to decide whether a project is a good fit for edge AI technologies. This is a great exercise to begin with, since it will also shed light on many of the other necessary considerations for the Discover phase of a project.

Describing a problem

Describing a problem is the first step in figuring out whether edge AI is a good fit to solve it. You should try to summarize a problem in a few sentences and bullet points—keep it short and to the point. A good description should include:

- High level summary of application
- Problems currently faced
- Constraints that must be worked around

In "Deep dive: Spotting rare wildlife with trail cameras" we discussed a possible application for edge AI in wildlife monitoring. Here's an example of how we might capture that use case as a problem description:

PROBLEM DESCRIPTION: TRAIL CAMERAS

Summary: Wildlife researchers install “trail cameras” in remote locations to monitor specific animal species. The devices typically use a passive infrared (PIR) motion sensor to detect motion, triggering a camera to take a photograph. Photographs are saved to a memory card. The memory card is collected periodically to obtain the photos, which are then analyzed by researchers wishing to estimate animal population and activity.

Problems:

- The PIR can be triggered by non-target species or moving vegetation, filling up the memory card with useless photos and reducing battery life.
- No animal activity data is available until the memory card has been collected and analyzed.
- It’s expensive to send somebody to collect a memory card from a remote location.
- If the memory card is collected too infrequently, it will fill up and important data will be missed.
- If the memory card is collected too frequently then money is being wasted on travel expenses.

Constraints:

- Trail cameras run on battery power and must be energy efficient.
- High bandwidth data connections are expensive in the field.
- Research budgets are typically low.

The exact format of your problem description doesn’t matter as much as the content. By capturing the exact problems and constraints, we can consider them while evaluating possible solutions.

Do I need to deploy to the edge?

At this point in the book, we're very familiar with the "To understand the benefits of edge AI, just BLERP" model for expressing the benefits of edge AI:

- Bandwidth
- Latency
- Economics
- Reliability
- Privacy

BLERP is the perfect tool to help us analyze our problem description and evaluate whether it might benefit from an edge architecture. A good way to do this is to create bullet points for each BLERP term.

To illustrate, let's explore **bandwidth**:

- Due to cost, trail cameras don't have access to much bandwidth. This makes doing work on device important.
- If we could analyze photos on-device, we could send the resulting information (much smaller than raw images) up to the cloud.
- This could help avoid expensive trips into the field to collect memory cards.

By brainstorming the potential impact of each term, we can start to understand whether BLERP benefits are important for this problem. Once you're finished brainstorming and summarizing you'll end up with something like the following BLERP analysis, taken from "Deep dive: Spotting rare wildlife with trail cameras":

BLERP ANALYSIS: TRAIL CAMERAS

- Bandwidth: Camera traps are often deployed in remote areas with low connectivity—perhaps with expensive, low-bandwidth satellite as the only option. With edge AI, the number of photos taken can be reduced enough to make it possible to transmit them all.
- Latency: Without edge AI, the latency involved with sending a researcher to collect photos from camera traps could be measured in months! With edge AI and a low-power radio connection, it's possible to analyze photos immediately and obtain useful information without having to wait.
- Economics: Avoiding trips out into the field saves large amounts of money; so does avoiding unnecessary use of expensive satellite radio.
- Reliability: If useless photos can be discarded, the memory card will take longer to fill up.
- Privacy: An edge AI camera can discard photos of humans on the trail, preserving the privacy of other trail users (such a local people, or hikers).

In this case, there are clear and obvious benefits to deploying on the edge across multiple BLERP terms. In other cases, it may not be so evident—for example, there might not be benefits under every single term. That doesn't necessarily mean that edge deployment isn't a good fit. As long as there's enough compelling benefit in any category it is worth considering further.

Things that don't work well on the edge

In some situations, you may find that your problem does not fit BLERP very well at all. Here's an example problem description for a different problem:

PROBLEM DESCRIPTION: MEDICAL IMAGING

A medical imaging device creates images that represent the interior of a patient's body. The device is very large and is typically located in a hospital. After scanning a patient, the device stores images on a hard disk attached to a computer network. Specially trained doctors use these images to help diagnose certain medical conditions. Special software must be used to view the images.

Problems:

- Diagnosing medical conditions by looking at images is challenging and requires medical training.
- If trained doctors are not available, patients may have to wait for a diagnosis.
- Doctors can only view images on certain computers that have the imaging software installed.

Constraints:

- Images represent sensitive patient information that must be kept secure.
- Imaging devices are very large and cannot be moved around.
- Imaging devices are very expensive.

From the description, it's clear that there are some problems worth solving here: it's challenging for people to diagnose medical conditions based on imaging data, and patients may have to wait for a diagnosis depending on availability of equipment or experts. Perhaps there's some potential for AI to help doctors analyze images.

However, the question we need to answer is whether this is a good problem to try to solve using edge computing. To do this, let's try and brainstorm some potential benefits via BLERP:

BLERP ANALYSIS: MEDICAL IMAGING

- Bandwidth: None. Machines are located in hospitals, which typically have good Internet connections, and they are already connected to a computer network. No benefit from reducing bandwidth requirements.
- Latency: It would be helpful for patients to have faster access to diagnosis.
- Economics: Performing analysis using AI would reduce reliance on doctors' time, which is expensive.
- Reliability: Doctors could use AI analysis to help improve their success at diagnosis.
- Privacy: AI analysis could reduce the need to expose sensitive patient data.

Superficially, these sound like compelling reasons. However, if we dig a little deeper, it's clear that most of these benefits are attainable *without* edge computing. Because the imaging device is located in a hospital, there's no major benefit to performing analysis on a legitimate "edge" device. Instead we could use a standard computer, either attached to the hospital's network or in the cloud.

In this case, a single fact—that a reliable network connection is already available—makes it unnecessary to use edge compute. But why not use it anyway? Does it make a difference whether we run compute on the edge or not?

Disadvantages of edge compute

While edge compute can have some massive benefits, especially in conjunction with AI, there are some very good reasons why most compute has moved into the cloud over the past decade. If the BLERP framework doesn't highlight some very good reasons to do work on the edge, you may be better off doing your information processing on a cloud server.

Here are some things that can make edge applications a challenge:

Development complexity

Writing and maintaining embedded applications is difficult, especially with smaller targets. The simpler your embedded code, the better. Even if an embedded device is required in order to collect data, it may make sense to simplify engineering by hosting the more complex application logic in the cloud.

Staffing

Embedded development requires very specific skills, and while a cloud application can be built and maintained by many types of engineers, embedded engineering talent can be harder to find. If your organization doesn't have access to embedded engineering talent, it may make sense to de-risk a project by leaving compute in the cloud.

Limited compute

Even the most powerful edge devices are nowhere near as capable as a beefy cloud server with access to a GPU. Some applications require levels of compute that would be unreasonable to deliver in the field—for example, some language models are gigabytes in size and require GPU to achieve low latency.

Deployment complexity

If you plan to update your application after it has been deployed, edge compute can create some problems. Updating edge firmware can be risky—devices can be "bricked" by a bug or a power outage at the wrong moment. Managing the application versions installed across a fleet of devices can also be a challenge. Working around these challenges is possible, but it requires engineering time. It may be simpler just to host your application logic in the cloud, where it can be updated with minimal fuss.

Hardware and support costs

Deploying and supporting a network of edge devices can be expensive. The expense can grow even higher if you require high-end devices with acceleration for machine learning workloads, or custom hardware designed for a specific purpose. Depending on the application, it may be cheaper to use less capable devices to collect data and send it to the cloud for processing.

Flexibility

If the workloads you wish to run outgrow your edge hardware, or your application changes substantially, you may need to buy new hardware to replace it. In contrast, cloud workloads can be scaled and modified at the click of a button.

Security

There is some security risk involved with allowing physical access to the implementations of your AI algorithms. In some cases, cloud compute may help reduce the risk. There'll be more on security later in the book.

As we saw in "Multi-device architectures", it's possible to split compute between edge devices and the cloud. This can be a helpful way to blend the benefits of each, especially when devices are deployed within controlled environments where reliable connectivity and power are available, such as homes or factories. For instance, a smart speaker can preserve privacy by doing wakeword detection on the edge while still benefitting from powerful cloud servers to run large, highly sophisticated transcription and NLP models. In an industrial setting, an edge computer vision system could identify potential manufacturing defects with extremely low latency before invoking a cloud model to precisely categorize the defects and determine the appropriate response.

Do I need machine learning?

As we learned in Chapter 1, AI doesn't always require machine learning. As a category, ML algorithms have various benefits and drawbacks that make

them ideal for some applications but of limited utility for others.

> **NOTE**
>
> It's important to identify whether your use case is a good fit for ML early on in the development process. ML-based projects involve a substantially different workflow, which will impact your timeline and budget.

For a given edge AI problem, you'll typically have to choose between a machine learning solution and a rule-based or heuristic solution. As we learned in "Conditionals and heuristics"), rule-based systems are designed by human beings using domain knowledge. They can make use of anything from basic arithmetic to incredibly complex physics equations. Here are some examples of applied heuristic algorithms in edge devices:

- A water kettle that shuts off when the temperature reaches boiling point.
- A diabetic insulin pump that dispenses precise doses of insulin based on blood glucose levels.
- A driver assist feature that uses traditional computer vision ("Image feature detection") to identify lane markings and center a car between them.
- The autopilot that flies a jumbo jet aircraft on international routes.
- The guidance systems of a space rocket headed to Mars.

Each of these examples, from simple to sophisticated, relies on domain knowledge. For example, the insulin pump's algorithm is based on knowledge of the human blood sugar regulation system, and the space rocket's guidance system is based on knowledge of physics, aerodynamics (at least for part of the trip), and the handling characteristics of the vehicle.

In each case, the systems involved are governed by strict rules. These rules may be complex, and their discovery might have have taken thousands of

years of human history, but at the end of the day they can be described with acceptable accuracy by engineers using mathematical formulae.

Often, digital signal processing algorithms are used in conjunction with rules-based systems. A little processing can go a long way in making it possible to react to input using simple rules. For example, a driver assist feature might use DSP in the form of image feature detection in order to reduce a complex image into a set of simple vectors representing lane markings. This makes it much easier to determine whether to steer the car left or right.

The nice thing about rule-based systems is that they can be *proven to work*. A heuristic algorithm is based on a system that is well-understood. It's possible to establish the mathematical correctness of an algorithm with regards to the underlying rules it is designed to model. This makes them reliable, trustworthy, and safe.

> **NOTE**
>
> If there's a rule-based solution to your problem, you should almost certainly choose it. Many problems can be solved in an elegant manner using rules and heuristics, and they can prove much easier to develop, support, and interpret than the machine learning alternative. They also tend to be far less demanding in terms of computational power.
>
> ML sounds exciting, but it's risky to use it unless you have a clear need. Heuristics are what landed man on the moon; there's a fair chance your problem is easier than that.

Unfortunately, not all problems can be solved with rule-based algorithms. Back in "Conditionals and heuristics" we encountered their two main weak points:

1. Problems with rules that are prohibitively difficult to discover.

   For instance, it could require a huge amount of research and development to discover the system of algorithms that underlies your complex, noisy, high frequency sensor data. Even if it's possible to

describe a system mathematically, it could be out of reach given your budget and time frame.

2. Problems with large numbers of variables.

   For example, there may simply be too many inputs for a rule-based system to be feasible. This is a common problem with image data, which is incredibly highly dimensional and also very noisy. It's tough to write an equation that describes the appearance of a dog.

For non-trivial problems, good implementations of rule-based algorithms may depend on extensive research, domain knowledge, and relevant engineering skills. These are not always available for a given project. The weak spots for heuristics provide an opportunity for ML to shine.

Reasons to use ML

While rule-based systems often depend on a scientific understanding of the processes they interact with, ML algorithms can learn an approximation of the relationships between variables through exposure to the data itself.

This can certainly make life easier. Here are some situations where it could make sense to consider trying ML, in order of validity:

- Your situation and data are too complex or noisy to model by conventional means.
- Too much fundamental research would be required to find rule-based solutions.
- You do not have access to the domain expertise necessary to implement a rule-based system.[1]

If you find yourself in one of these situations, machine learning can be a huge help. "Building an artificial nose" provides a nice example.

BUILDING AN ARTIFICIAL NOSE

IoT engineer Benjamin Cabé wanted to build an artificial nose—a device that can identify objects and substances by their distinctive smell. His initial goal was to try and distinguish between different types of alcoholic spirits: vodka, rum, and Scotch whisky.

< image of artificial nose >

Benjamin had access to a cheap gas sensor designed to measure levels of several different types of gases. To build a rule-based algorithm for distinguishing between the spirits he would have had to perform a chemical analysis of the various drinks, understanding their composition, and worked backwards from there to determine which gases to look out for. He'd also have to account for any gases that might be present in the surrounding environment, since they might lead to false positives.

This type of research was beyond the scope of his project. Fortunately for Benjamin, he was familiar enough with machine learning that he knew it might be able to help.

Benjamin used his gas sensor to capture a small dataset of samples from several different types of drinks. He used this data to train a simple machine learning classification model to identify which gas readings were associated with which drink. The project was a success! Benjamin's system was able to discern between different types of drinks—even to the level of telling one brand of whisky from another.

The ability of machine learning to extract rules from data allowed Benjamin to build a successful project without having to invest in chemical analysis of the drinks, which could have been time consuming and expensive. It also allowed him to avoid writing the sensitive, hand-tuned logic required to account for other gases present in the environment, since the dataset—collected in the real world—already factored this in.

It turns out that complex, noisy data are very common. In fact, most real world data is a bit of a mess! One of the strengths of machine learning, especially deep learning models, is that given enough data they can learn to account for noise. During training, the parameters of the model are tuned in such a way that they filter out the noise from the data, leaving just the important information—which can be used to make a decision.

In addition, machine learning models are great at identifying the hidden patterns that exist within their training data. Relationships that would be invisible to the human eye, or too complex for our hand-coded rules to represent, can become quite clear to machine learning models when they are provided with enough training data.

These advantages make machine learning a great pick if you are confronted with a bunch of noisy data describing unclear relationships. However, ML's use of data can also provide some risk.

The drawbacks of ML

From an engineering perspective, there are three major drawbacks to machine learning: Data requirements, explainability, and bias.

It's well known that today's ML depends heavily on data[2]. Large amounts of data are often required to train and test machine learning systems. Finding adequate data, and ensuring its quality, is the biggest challenge and expense associated with machine learning.

Data may seem plentiful—after all, don't we have "data warehouses" and "data lakes", filled with decades of IoT sensor data that has been carefully captured and logged? Unfortunately, raw data isn't enough. Most of today's machine learning techniques require data that has been *labelled*, meaning it has been tagged with information describing what it means. This tedious task often falls to humans, which makes it expensive and risky (since it's easy to get things wrong).

In addition, machine learning models can only make sense of situations they have seen before. This makes datasets highly context-dependent. A model trained on dataset collected using a specific type of sensor may not perform well if fed data captured by a different brand. A dataset of typical

household items from one country may not be any help when trying to identify items from another.

Research is ongoing to mitigate these issues, and amazing progress is being made, but the fact remains that machine learning typically requires a lot of data. We'll learn more about this topic in the next chapter.

The second big drawback of machine learning, *explainability*, we already touched on in "Interpretability and explainability". While there are some highly explainable ML models, the more sophisticated a model gets the more challenging it can be to pinpoint exactly why it is making the predictions it does.

This problem is compounded by the fact that, thanks to their origins in statistics, many types of ML models don't give definitive answers. Ask a question of a rule-based system and it will give you a nice, firm response, with clearly visible workings that you can double-check and review. Ask the same question of deep learning model and you'll get a fuzzy probability distribution that indicates the *potential* answer. Trace the answer back through the system and you'll meet an inscrutable mess of linear algebra that is beyond the comprehension of a human mind.

Their probabilistic nature means that ML models are great for dealing with fuzzy situations and non-obvious rules. Unfortunately, this means that their output has some of the same properties. This can be a challenge for many applications.

For example, the code that powers safety-related devices in fields like medical technology, automotive, and aerospace is often expected (through best practices and government regulation) to be *provably* correct. It's very difficult to meet this bar with a probabalistic model whose internal rules can only be gleaned through probing and experimentation.

The third big drawback of ML, *bias*, is a direct result of the first two challenges. We encountered this issue back in "Black boxes and bias". When we create ML models, we rely on our datasets to both train them and to validate their performance. Our goal is to produce a model that works well in the real world. However, the real world is a big place, and it's quite challenging to capture all of its possible variation in a finite dataset.

If our dataset only includes a subset of all possibilities, our model may fail to perform correctly on the others. Even worse, because we don't have any examples of those other possibilities in our dataset, we'll have no idea that this is even a problem. Our model may appear to be working great when in fact it has some major issues.

To compound the issue, our model won't necessarily even *tell us* when it is having trouble. Instead, it will just make its best guess—which could be catastropically wrong. Without data to test it, and without an easy way to analyze the internal rules of the model and understand where it might fall short, we'll have no way of knowing there is something wrong-beyond our application failing.

A SIMPLE EXAMPLE OF ML BIAS

Imagine you're building a system that uses audio to identify faults in an industrial machine. Working in your research and development lab, you collect a big dataset with thousands of audio samples representing correct and faulty operation. When you train and test the model in the lab, it works great. But when you deploy it to a customer's factory, it identifies far more faults than expected.

On investigation, you discover that the model is detecting faults when *adjacent* machines are running. Because your dataset only contains data from your quiet lab, the model never learned to account for the ambient sounds of a factory floor. The model's biases reflect the lab conditions under which it was trained. It does not perform well in the real world.

Since we can never hope to sample the entire world in our dataset, bias is inevitable. We can merely try to manage the risk. However, some situations come with so much risk of bias—or the consequences of any bias may be so high—that they are not well suited to ML.

Knowing when to use ML

There's a common saying about solving problems: "If all you have is a hammer, everything looks like a nail". Machine learning is far more

exciting than the shiniest of hammers, and it's tempting to try to use it everywhere. Sadly, its complexity, limitations, and inherent risks make it a poor choice in many situations. Mat Kelcey, principal ML engineer at Edge Impulse, is fond of saying "the best ML is no ML at all."

There's no shame in using traditional algorithms to solve problems. As we saw in "Artificial intelligence (A.I.)", the *intelligence* part of AI comes from "knowing the right thing to do at the right time". It doesn't matter to your users whether this knowledge is embedded in the form of an `if` statement or in the form of a deep learning model.

With that in mind, here's a checklist you can use to help decide whether ML might be appropriate for your application.

- There is no existing rule-based solution, and you don't have the resources to discover one.
- You have access to a high quality dataset, or collecting one is within your budget.
- Your system can be designed to make use of fuzzy, probabilistic predictions.
- You do not need to explain the exact logic behind your system's decisions.
- Your system will not be exposed to inputs beyond those reflected in its training data.

Determining feasibility

So you have an idea for an edge AI project? Your first goal should be to determine its feasibility. There are many things to consider. Perhaps your project would be better served using server-side AI—or maybe your solution would require machine learning, but it's not feasible to collect a dataset. Alternatively, perhaps it's the perfect fit!

The first step is to try and come up with an *ideal* solution to the problem in your problem description. If you try to forget about technological

limitations and think at an extremely high level, what would you want the system to do?

PRE-CONCEIVED NOTIONS

The purpose of our ideal solution is to give us something to aim for. If we constrain our search for ideas by what we think is feasible, we may miss promising solutions that are not immediately obvious. By keeping things *ideal*, we make sure we don't limit our own creativity.

The ideal solution also helps us avoid the temptation to use a certain technology because we are excited about it. It's a very common trap: we've just learned about some fascinating new technique and have been looking for an excuse to try it—so we overlook another method that would have given better results.

Once we have an ideal solution we can begin to consider feasibility from a few different angles: technology, dataset, business, and ethical. For a project to be feasible overall, it needs to sit nicely in the sweet spot of each.

< Venn diagram showing technology, dataset, business, ethical >

Let's explore each of these angles one by one. To illustrate, we'll use an example application.

EDGE AI FOR WAREHOUSE SECURITY

A warehouse filled with valuable products could be an attractive prospect to thieves, and a single security guard can't watch every corner at the same time. Perhaps an AI-powered security system could help.[3]

In terms of our warehouse security project, one ideal solution might be a system that is aware of every human being on the premises, understands contextually which of those people are *supposed* to be present, and informs a security guard of the location of any others.

Technological feasibility

In Chapter 3 we took a long walk through the technologies most important to edge AI. That material will be a great resource as you try to understand the feasibility of your ideas.

The first step is to map your idea onto edge AI concepts and methodologies. Here are some of the key ones, all of which are featured earlier in the book.

- Sensors: How will you collect the data you need?
- Data formats: What kinds of signals will your sensors output?
- Feature engineering: What are the options available for processing raw signals?
- Processors: How much compute can you afford, budgeting by cost and energy usage?
- Problem types: Do you need to perform classification, regression, or something else?
- Rule-based or ML: Is it necessary to use machine learning, or can you get away with a rule-based or heuristic approach?
- Choice of ML algorithm: Will classical ML suffice, or do you need a deep learning model?
- Application architectures: Will you need a single edge device, or a more complex arrangement?

At this point, it's still too early to try to answer all of these questions definitively. Instead, it makes sense to brainstorm a handful of possible solutions: start with 4 or 5 rough ideas, but feel free to capture more if they come easily to you. They don't all have to be fully thought out, but you should try to capture elements of the above.

Here's an example in our warehouse security context:

BRAINSTORMING IDEAS FOR WAREHOUSE SECURITY

Even though we're trying to brainstorm an edge AI system, in most cases it's helpful to begin by establishing a cloud-powered baseline. This gives us a known quantity to measure our edge solutions against. Of course, in some cases there may not *be* a cloud-powered baseline.

Solution 1: Cloud AI. The warehouse has numerous hard-wired cameras, each streaming video to the cloud via networked Internet access. A cloud server runs deep learning person detection on every video stream concurrently, messaging the security guard via an app if a person is detected in an unauthorized location.

We now have a cloud-based system to reason about. Let's see what happens if we push some of the compute back down to the edge.

Solution 2: Edge server. The warehouse has numerous hard-wired cameras, each streaming video to an on-site edge server via a network. The edge server runs deep learning person detection on every video stream concurrently, messaging the security guard via an app if a person is detected in an unauthorized location.

This sounds interesting! It sounds achievable, and there are certainly some technology benefits—for example, we're no longer reliant on an Internet connection. Now let's see if we can push even *more* compute to the edge:

Solution 3: On-device compute. The warehouse has numerous hard-wired cameras, each of which is equipped with a high-end MCU. The MCU runs deep learning person detection on the cameras's video stream, messaging the security guard via an app if a person is detected in an unauthorized location.

In addition to edge-versus-cloud, there are many other axes we can explore for ideas. For example, how about varying up the sensor type?

Solution 4: On-device compute with sensor fusion. The warehouse has numerous hard-wired devices, each equipped with multiple sensors including video, audio, and radar, along with a high-end MCU. The

MCU uses sensor fusion to detect people, messaging the security guard via an app if a person is detected in an unauthorized location.

We now have four possible solutions to explore. Each one has its own benefits and drawbacks that can be analyzed, compared, and debated. It's not always obvious which solution makes the most sense; the correct answer will vary depending on everything from business requirements to the skillset of an organization. The key is to produce a portfolio of ideas so that you can begin to explore the options.

Framing problems

In order to fully explore the technology requirements of any AI solution, we need to be able to *frame* our problem in terms of the tools available to us. In "Algorithm types by functionality" we met an assortment of useful techniques:

- Classification
- Regression
- Object detection and segmentation
- Anomaly detection
- Clustering
- Dimensionality reduction
- Transformation

To solve any problem, we first need to break it down into chunks that can be addressed using these techniques. A given problem may require multiple techniques to solve. For example, identifying intruders in unauthorized locations might involve both object detection (for spotting people) and anomaly detection (to identify when a person is behaving unusually, like sneaking around a warehouse aisle at night).

Each technique may require a different type of algorithm: for example, object detection might require a deep learning model, and anomaly

detection could be done using classical ML. By breaking down a problem into these techniques we can better understand the computational burden of the work that must be done—which will aid with our solutions brainstorming and help inform our hardware choices.

Any given problem can typically be broken down (or framed) in many different ways, each with its own set of technology requirements. For example, it might also be possible to spot intruders in a warehouse using dimensionality reduction: we could use an embedding model to describe any people in view, then compare their embeddings to a database, allowing us to identify when a non-employee is in the warehouse.

This would have different technology, data, business, and ethical considersations than an object detection based system. As such, framing gives us another tool we can use to explore the space of possible solutions and find something that fits our unique requirements.

Device capabilities and solution choice

For every edge AI project there are innumerable hardware options. For instance, our warehouse security brainstorm resulted in solutions that could be implemented using MCUs, with an edge server, or in the cloud. Each individual solution has its own gigantic space of hardware choices—for example, for an MCU-based project we must select hardware from a list including dozens of silicon vendors, each with dozens of chips, all configurable in endless ways.

From a feasibility standpoint, we need to understand which of these hardware options are reasonable given the constraints of our problem—as described by our problem description. Constraints might include cost, in-house expertise, existing brown field (“Green and brown field projects”) systems, or supply chain considerations. Capturing these constraints shrinks both the space of hardware options and the space of possible solutions.

After applying constraints, we might discover that there is no solution that fits. For example, in a brown field project the only available hardware might not have enough memory to run an object detection model with suitable performance.

Table 3-1 provides a quick reference you can use to understand whether your application is in the ballpark of feasibility for the hardware options available to you. Remember, you can always split your application across multiple device types if you need additional flexibility. We'll cover this in depth in Chapter 6.

Dataset feasibility

Alongside technical feasibility, edge AI application development is constrained by data that is available. Machine learning is famously data-hungry, but even hand-coded, rule-based approaches require substantial amounts of data to develop and test.[4]

Data collection is difficult, time consuming, and expensive, so it is challenging but vital to understand the data requirements of your project during the feasibility assessment stage. There are two steps to understanding data feasibility:

1. Estimating how much data is required to solve your problem.
2. Understanding whether you will be able to obtain enough of it.

Both of these topics are covered in detail in Chapter 5. As we'll see, understanding data requirements involves both research and engineering work. This means that you won't know your actual data requirements until you have invested a significant amount of time in a project.

At this stage, it's okay just to have a ballpark idea, perhaps based on some precedents that you have found via research (as we will discover in Chapter 5). If it doesn't look like you will have enough data, your project may not be feasible. It's critical that you rule this out at an early stage to avoid wasted development effort.

DATASET FEASIBILITY FOR WAREHOUSE SECURITY

Our security application revolves around detecting people in a warehouse. To begin understanding the data requirements, it's helpful to identify some similar applications in scientific literature. For example, some web searching around the topic of person detection might unearth a reference to the *Visual Wake Words dataset* (A. Chowdhery et al., 2019), a dataset of 115,000 images of people in a wide range of contexts.

The literature shows it is possible to get >95% accurate performance on the dataset with a model that will run on a high-end MCU. This gives us at least some assurance that our use case might be feasible. In addition, the fact that it is publicly available means that we could potentially use it to help train our own model.

At this stage, this might be enough to convince us that our project is feasible from a dataset perspective. There's always some risk that things may not work out—for instance, we may discover that detecting people in a dark warehouse is more challenging than in the typical contexts that appear in the Visual Wake Words dataset. It's up to us to decide the level of risk we are willing to accept, and we can always reduce the risk through experimentation.

Data issues are one of the leading causes of ML project failure, so if you see negative signals during this part of your feasibility check, don't just cross your fingers and hope for the best.

Business feasibility

There are two main ways that organizational issues can impact the feasibility of a project. Firstly, for an AI application to be successful it needs to provide some clear benefit. In a business context, this could be to customers, executives, or the balance sheet. In a scientific context, it might mean allowing more work to be done with the same budget. At the outset of any project, it is vital to make sure that the proposed work will, if successful, do some real good.

Secondly, AI application development is limited by the practical constraints faced by organizations. For example, there may not be enough budget to collect a dataset of sufficient size to train an effective ML model. Other common constraints include time, expertise, and long term support from stakeholders.

Proving benefit

One particularly effective way of proving (and demonstrating) the benefits of an edge AI application is called a “Wizard of Oz prototype”. In the story of The Wizard of Oz, the titular wizard is first introduced as an impressive supernatural being. However, he is later revealed to be a normal man, hidden behind a curtain, who is controlling the illusion remotely.

In Wizard of Oz prototyping, a mock version of an AI product is produced. It is superficially functional, but in reality its functionality is controlled by hidden human actions. The mock product can be tested and experienced by stakeholders, allowing people to get a sense for how it might work in the real world.

WIZARD OF OZ TESTING FOR WAREHOUSE SECURITY

To test our warehouse security concept, a basic mobile application is developed. A security guard, equipped with the application, is tasked with his normal patrol. Periodically, a notification is sent to the guard by a human tester in order to simulate what would happen if an AI system detected an intruder. The guard reacts by investigating the issue. He records the amount of time it takes for him to respond.

Subsequent analysis finds that, given the response time, the application is not helpful: in the large warehouse complex, it takes so long for the guard to reach the site that any thief would have long departed. The expense of developing the application would not be worth it, and the money would be better spent hiring an additional guard.

Alternatively, it may have been found that the guard was able to attend to each situation rapidly, and that the application would help improve their ability to protect the site. The findings help persuade management that the investment is worthwhile.

In either case, the outcome of the Wizard of Oz test helps the organization save money.

This can be an extremely helpful exercise. Even without the technology portion of the project, the user experience can be explored, analyzed, and refined. If the experience is impressive, it can go a long way towards convincing stakeholders that a project is worthwhile. If things do not work well even in an artificial form, it’s a good signal that you should go back to the drawing board.

For any evaluation to be successful, it’s important to get stakeholder agreement on what “good” looks like. In our warehouse example, this might mean setting the minimum time it takes for a single guard to respond to an intrusion. During the test, we set up instrumentation that helps us monitor these metrics, deploy a baseline version of our application (for example, the fake Wizard of Oz application), and then evaluate it.

During this process, it's really important to test out the *current* solution alongside the proposed ones. This will provide a clear signal as to whether the solutions being evaluated are genuinely beneficial. This is an important thing to check throughout the development process, not just at the start.

Since we've established a specific threshold for "good", it may even turn out that the current solution is good enough to satisfy the stakeholders, in which case the organization can save money by *not* adopting an AI-based solution. But if we prove that there's a benefit, the project can proceed with confidence.

Understanding constraints

Of course, beyond Wizard of Oz prototyping, there are many ways that organizations work to determine the risk, reward, and benefit of new projects. AI projects are no different, and establishing that your idea is a good fit for the needs of the organization you work within will be critical to ensuring its success.

Part of this is identifying your organizational constraints and making sure they will not pose any problems. Here are some of the top constraints when developing AI applications:

- **Expertise.** Although AI engineering is increasingly accessible, having experienced AI experts on hand will help de-risk your project. Make sure the skills you need are available to you before you begin, whether in-house or via consultants. Chapter 6 has some information about hiring for edge AI roles.
- **Timeline.** AI development is a data-driven, iterative process, and it's even more challenging to produce accurate time estimates for than traditional software engineering. Make sure you have room for contingencies, such as discovering late in the process that you need to collect more data.
- **Budget.** The three most expensive parts of AI development are payroll, data collection, and testing in the field. If you are an individual or a small organization, compute time for training can become

significant—although it will typically not exceed more than a few thousand dollars for anything other than a gargantuan project.

- **Long term support.** As we'll see in [Link to Come], AI applications need long term support to remain effective. As the world changes around them, ML models and hard-coded rules will need to be updated and refined. If your project will be deployed for more than a few months, it's important to know whether your organization can afford (and is willing) to support it.

You are the expert in the way your organization works, and it's up to you to understand whether it is capable of supporting the project you wish to develop.

Ethical feasibility

Ethical review is one of the most important parts of any feasibility analysis. A project that fails due to unaccounted technology risk may cost a lot of wasted money, but damage resulting from ethical issues is potentially unbounded—it can easily cost a company its reputation, trigger punitive regulatory measures, and cause direct harm to human beings.

"Building ethical applications" introduced some of the ethical issues that can affect AI products. As part of a feasibility analysis, it's critical to rigorously explore and document any potential ethical issues that could result from your application. Beyond the product itself, it's important to also understand any potential risks that come from the development process: data collection, deployment, and support.

Some key questions to ask include:

- Is the product itself ethical? *Example: Some products (such as AI weapons) are designed to directly cause harm, while others may cause harm inadvertently.*
- Can the required data be obtained ethically? *Example: Some datasets may not be feasible to collect without violating user privacy.*

- Is it possible to test the product ethically? *Example: An application that provides medical advice may cause harm during testing if the advice given is not reliable.*
- Have the potential risks been documented? How can they be mitigated? *Example: What is the potential harm if an application produces incorrect predictions?*
- Does the application work for all of its potential users? *Example: A safety-related speech detection application might be ineffective with speakers who have regional accents.*

WAREHOUSE SECURITY APPLICATION ETHICAL FEASIBILITY REVIEW

A good way to begin an ethical review is to brainstorm a list of potential issues. Here's an example brainstorm for our warehouse security application. This is not intended to be an exhaustive list!

Is the product itself ethical? - The product might cause harm if it labels innocent people as suspicious. - The product might cause harm if it results in the security team being reduced in size.

Can the required data be obtained ethically? - Available datasets may include photographs of individuals who did not consent to being included. - If data is collected at the warehouse, employees may not feel comfortable with being included.

Is it possible to test the product ethically? - False alarms could lead to harm if an innocent person is labelled as suspicious. - The testing process could distract the security team and lead to security issues.

Does the application work for all of its potential users? - The application may classify people as suspicious at different rates depending on their appearance, not their intent. - The application may be difficult to use for security guards who are unfamiliar with similar technology.

One way to reduce the risk of unforeseen ethical issues is to make sure the ethical review is conducted by a diverse group of people from a representative set of demographics. In addition to the product's development and design team, this group should include business stakeholders, potential users, non-users who may be affected by the product, and members of the general population. Ideally, your review would also include people who have direct expertise in the ethical evaluation of AI systems—there are many consulting firms who will perform this service.

> **NOTE**
>
> In a group setting, some reviewers may not feel comfortable speaking up—for example, an employee may be hesitant to be seen as the one who derailed an important project. For this reason, it's a good idea to have reviewers submit their thoughts anonymously.

Making a final decision

At this point, we've reviewed the feasibility of our project from the perspective of technology, dataset, business, and ethics. We should have enough information to make a call. If none of the solution ideas we have brainstormed seem like a good fit, our next steps should be as follows:

1. Update the problem description with the new constraints that we have identified during the review process.
2. Perform a new brainstorm of solutions, coming up with a new set of possible solutions that factor in our newly identified constraints.
3. Go through the same feasibility review process with our new solutions.

As with everything in AI, this can be an iterative process. You may have to repeat these steps several times in order to clarify your understanding of the constraints and arrive at a solution that works. It's worth having patience and being willing to revisit your assumptions. You will likely end up with a potential solution, even if it isn't what you envisoned at the start.

That said, in some cases there may simply not be a good edge AI solution for the problem you are trying to solve. If that happens, take note of the reasons. Perhaps they are ethical, and a sign that the project is too ethically risky to consider. Alternatively, they may be purely technological—which could mean the project may become feasible some point down the line, as new hardware and techniques become available.

In any case, even if you have not been able to identify a promising solution, the process of exploring the solution space from a feasibility standpoint will have been incredibly instructive. You now likely know more about this problem space, from an AI perspective, than anybody out there.

If a solution does not pass the feasibility test, resist the temptation to continue anyway. If you've proven that it is too risky, trying to develop the project will result in wasted time and potential harm.

> **NOTE**
>
> Even the knowledge that there is no reasonable edge AI solution to a problem is valuable information, the fact that you are aware of it is a form of competitive advantage. You may see other organizations wasting time pursuing it—but you will have the confidence to know that their efforts will fail and you will get better results focusing elsewhere.

If you have reached this point and your project seems feasible, congratulations—it's time to start making it a reality.

Planning an edge AI project

Edge AI development is a multi-stage process involving iterative development and potentially unbounded tasks (such as data collection, which as a process is never truly finished). With this in mind, it's important to come up with a plan before developing a solution.

In "The edge AI workflow" we saw the various workflow stages and how they are connected by a multitude of feedback loops. As an iterative

process, you can spend as long as you like on each section. The two most important aspects of planning are:

1. Defining acceptable performance
2. Understanding time and resource constraints

Defining acceptable performance

The very first stage of your planning process should be to come up with a set of concrete standards for what acceptable performance will look like for your system. This must be done in conjunction with business stakeholders and ethical analysis, since an underperforming system may create risks in these areas.

Your task during development will be to step through the iterative process until you have met your goals with regards to acceptable performance. Once they have been satisfied, you can confidently sign off on the project knowing that it is working well enough. Your goals should be set realistically—they need to be achievable—but they should also be set in such a way that the project will genuinely deliver value if they are met.

Understanding time and resource constraints

It’s very important to understand both how much time you have and the resources you have available to help deliver a project—including funds, expertise, and hardware. Estimating development time is famously hard, and this is especially true with AI projects.

Due to the iterative nature of AI development, a traditional waterfall-style development model will not work well. Instead, you’ll need to understand and manage risk as you move forward. A good approach is to aim to have something working end-to-end as rapidly as possible.

You can then iterate on the entire system as well as its individual components. Your very first version may be a “Wizard of Oz” prototype that uses off-the-shelf hardware and simple logic. You might the iterate on this, training a simple ML model to replace the “Wizard of Oz” component and then creating a custom hardware design.

At every point during the process, your entire system can be tested end-to-end to determine whether it meets the performance standards you have defined. One benefit of this is that it removes the risk that you will spend all of your time on one stage of the project—for example, training a model—and not leave enough time for the remainder. Another benefit is that, sometimes, you may find that a simpler system than you originally envisioned is more than adequate to meet your performance goals.

We'll dig deeper into the topic of planning when we reach Chapter 7.

HARDWARE IS HARD

Hardware projects come with unique challenges, and it's often important to begin the process of hardware design early in the process, concurrently with any software work. Once you have identified a suitable processor, obtain a development board so that you can begin working directly with hardware as soon as possible. This will help reduce the risk of unexpected friction when you try to deploy your logic to the hardware.

Due to the realities of the hardware supply chain, you may find that you need to pull the trigger on hardware orders before you feel entirely confident in your application's requirements. This is a major source of risk, but it may be an unavoidable one. It's best managed by getting your application deployed onto some development hardware as quickly as possible. Most silicon manufacturers provide plenty of development boards for this exact purpose.

If you are particularly worried it may be helpful to select hardware with some excess capacity, giving you some headroom if you underestimate the requirements of your application code.

Summary

We now understand the general workflow that applies to edge AI projects, and we've learned how to evaluate problems and generate promising solutions.

In the next chapter, we'll move forward with the first part of the edge AI workflow—collecting an effective dataset.

---

1 As we'll see, domain expertise is still critical in training and evaluating ML models. However, ML-based approaches potentially enable experts with knowledge of a specific problem area to get good results without having to enlist as much outside help in other areas (such as signal processing).

2 The effort to reduce the data requirements of ML algorithms is one of the most important and fascinating strands of ML research.

3 In the real world we would base our ideal solution on a detailed problem description. To keep things lightweight, we'll go with this simplified statement.

4 Even if you are reproducing a well-known algorithm from a textbook, you'll still need to run data through it to make sure your application works end to end.