Ultimate Guide to Machine Learning for Embedded Systems

Jeff Sieracki, Engineering Director, Renesas Electronics

Introduction

What is Machine Learning for Embedded Systems?

Machine learning is a subfield of Artificial Intelligence which gives computers an ability to learn from data in an iterative manner using different techniques. Our aim here being to learn and predict from data. This is a big diversion from other fields which poses the limitation of programming instructions instead of learning from them. Machine Learning in Embedded Systems specifically target embedded systems to gather data, learn and predict for them. These systems typically consist of low memory, low Ram and minimal resources compared to our traditional computers.

So now you know a little more about what we mean by “machine learning for embedded systems”, but maybe you’re still unsure about where or how to start? That’s why we’ve created the ultimate guide to machine learning for embedded systems. Over the last few years, as sensor and MCU prices plummeted and shipped volumes have gone thru the roof, more and more companies have tried to take advantage by adding sensor-driven embedded AI to their products.

Automotive is leading the trend – the average non-autonomous vehicle now has 100 sensors, sending data to 30-50 microcontrollers that run about 1m lines of code and generate 1TB of data per car per day. Luxury vehicles may have twice as many, and autonomous vehicles increase the sensor count even more dramatically. But it’s not just an automotive trend. Industrial equipment is becoming increasingly “smart” as makers of rotating, reciprocating and other types of equipment rush to add functionality for condition monitoring and predictive maintenance, and a slew of new consumer products from toothbrushes, to vacuum cleaners, to fitness monitors add instrumentation and "smarts".

Real-Time, at the Edge, and a Reasonable Price Point

What these applications have in common is the need to use real-time, streaming, complex sensor data – accelerometer, vibration, sound, electrical and biometric signals – to find signatures of specific events and conditions, or detect anomalies, and do it locally on the device: with code that runs in firmware on a microcontroller that fits the product's price point.

When setting out to build a product with these kinds of sensor-driven smarts, there are three main challenges that need to be overcome.

Simultaneous challenges when using sensors with Embedded AI:

- Variation in target and background
- Real-time detection
- Constraints – size, weight, power consumption, price

Variation

Real world data is noisy and full of variation – meaning that the things you're looking for may look different in different circumstances. You will face variation in your targets (want to detect sit-ups in a wearable device? First thing you will hit is that people do them all slightly differently, with myriad variations). But you will also face variation in backgrounds (vibration sensors on industrial equipment will also pick up vibrations transmitted thru the structure from nearby equipment). Background variation can sometimes be as important as target variation, so you'll want to collect both examples and counter-examples in as many backgrounds as possible.

Real-Time Detection in Firmware

The need to be able to accomplish detections locally that provide a user with a "real-time" experience, or to provoke a time-sensitive control response in a machine, adds complexity to the problem.

Constraints – Physical, Power, and Economic

With infinite computing power, lots of problems would be a lot easier. But real-world products have to deliver within a combination of form factor, weight, power consumption and cost constraints.

Traditional Engineering vs Machine Learning

To do all of this simultaneously, overcome variation to accomplish difficult detections in real-time, at the edge, within the necessary constraints is not at all easy. But with modern tools, including new options for machine learning on signals (like Renesas Reality AI) it is becoming easier.

Certainly, traditional engineering models constructed with tools like Matlab are a viable option for creating locally embeddable detection code. Matlab has a very powerful signal processing toolbox which, in the hands of an engineer who really knows what she is doing, can be used to create highly sophisticated, yet computationally compact, models for detection.

Why Use Machine Learning?

Why is machine learning increasingly a tool of choice?

For starters, the more sophisticated machine learning tools (like Reality AI from Renesas) that are optimized for real-time analytics of signal data and embedded deployment can cut months, or even years, from an R&D cycle. They can get to answers quickly, generating embeddable code fast, allowing product developers to focus on their functionality rather than on the mathematics of detection.

But more importantly, they can often accomplish detections that elude traditional engineering models. They do this by making much more efficient and effective use of data to overcome variation. Where traditional engineering approaches will typically be based on a physical model, using data to estimate parameters, machine learning approaches can learn independently of those models. They learn how to detect signatures directly from the raw data and use the mechanics of machine learning (mathematics) to separate targets from non-targets without falling back on physics.

Different Approaches for Different Problems

It is also important to know that there are several different approaches to machine learning for this kind of complex data. The one getting most of the press is “Deep Learning”, a machine learning method that uses layers of convolutional and/or recurrent neural networks to learn how to predict accurately from large amounts of data. Deep Learning has been very successful in many use cases, but it also has drawbacks – in particular, that it requires very large data sets on which to train, and that for deployment it typically requires specialized hardware ($$$). Other approaches, like the one we take at Renesas, may be more appropriate if your deployment faces cost, size or power constraints.

Three things to keep in mind when building products with Embedded AI

If you’re thinking of using machine learning for embedded product development, there are three things you should understand:

1. Use rich data, not poor data. The best machine learning approaches work best with information-rich data. Make sure you are capturing what you need.
2. It’s all about the features. Once you have good data, the features you choose to employ as inputs to the machine learning model will be far more important than which algorithm you use.
3. Be prepared for compromises and trade-offs. The sample rate at which you collect data and the size of the decision window will drive much of the requirements for memory and clock-speed on the controller you select. But they will also affect detection accuracy. Be sure to experiment with the relationship between accuracy, sample rate, window size, and computational intensity. **The best tools in the market will make it easy for you to do this.**

Machine Learning: Lab vs Real World

"In theory there's no difference between theory and practice. In practice there is."

—YOGI BERRA

Not long ago, TechCrunch ran a story reporting on Carnegie Mellon research showing that an **"Overclocked smartwatch sensor uses vibrations to sense gestures, objects and locations."** These folks at the CMU Human-Computer Interaction Institute had apparently modified a smartwatch OS to capture 4 kHz accelerometer waveforms (most wearable devices capture at rates up to 0.1 kHz), and discovered that with more data you could detect a lot more things. They could detect specific hand gestures, and could even tell a what kind of thing a person was touching or holding based on vibrations communicated thru the human body. (Is that an electric toothbrush, a stapler, or the steering wheel of a running automobile?")

"DUH!"

To those of us working in the field, including those at Carnegie Mellon, this was no great revelation. "Duh! Of course, you can!" It was a nice-but-limited academic confirmation of what many people already know and are working on. TechCrunch, however, in typical breathless fashion, reported as if it were news. Apparently, the reporter was unaware of the many commercially available products that perform gesture recognition (among them Myo from Thalmic Labs, using its proprietary hardware, or some 20 others offering smartwatch tools). It seems he was also completely unaware of commercially available toolkits for identifying very subtle vibrations and accelerometry to detect machines conditions in noisy, complex environments (like Reality AI solutions for industrial equipment monitoring), or to detect user activity and environment in wearables (Reality AI for consumer products).

But my purpose is not to air sour grapes over lazy reporting. Rather, I'd like to use this case to illustrate some key issues about using machine learning to make products for the real world: Generalization vs Overtraining, and the difference between a laboratory trial (like that study) and a real-world deployment.

Generalization and Overtraining

Generalization refers to the ability of a classifier or detector, built using machine learning, to correctly identify examples that were not included in the original training set. Overtraining refers to a classifier that has learned to identify with high accuracy the specific examples on which it was trained, but does poorly on similar examples it hasn't seen before. An overtrained classifier has learned its training set "too well" – in effect memorizing the specifics of the training examples without the ability to spot similar examples again in the wild.

That's ok in the lab when you're trying to determine whether something is detectable at all, but an overtrained classifier will never be useful out in the real world. Typically, the best guard against overtraining is to use a training set that captures as much of the expected variation in target and environment as possible. If you want to detect when a type of machine is exhibiting a particular condition, for example, include in your training data many examples of that type of machine exhibiting that condition, and exhibiting it under a range of operating conditions, loads, etc. It also helps to be very skeptical of "perfect" results. Accuracy nearing 100% on small sample sets is a classic symptom of overtraining.

It's impossible to be sure without looking more closely at the underlying data, model, and validation results, but this CMU study shows classic signs of overtraining. Both the training and validation sets contain a single example of each target machine collected under carefully controlled conditions. And to validate, they appear to use a group of 17 subjects holding the same single examples of each machine. In a nod to capturing variation, they have each subject stand in different rooms when holding the example machines, but it's a far cry from the full extent of real-world variability. Their result has most objects hitting 100% accuracy, with a couple of objects showing a little lower. Small sample sizes. Reuse of training objects for validation. Limited variation. Very high accuracy... Classic overtraining.

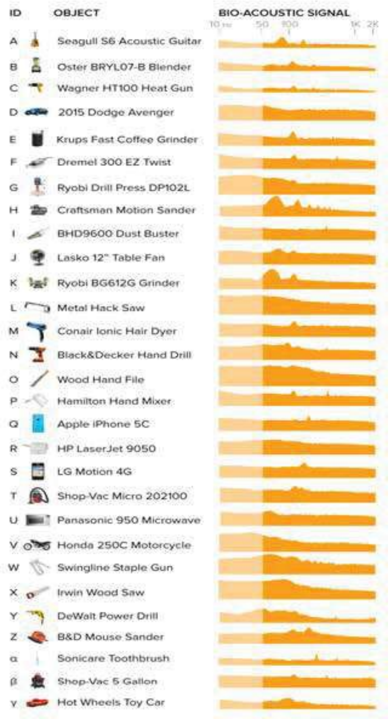


Illustration from the CMU study using vibrations captured with an overclocked smartwatch to detect what object a person is holding.

Detect Overtraining and Predict Generalization

It is possible to detect overtraining and estimate how well a machine learning classifier or detector will generalize. At Renesas, our go-to diagnostic is the K-fold validation, generated routinely by our tools. K-fold validation involves repeatedly; 1) holding out a randomly selected portion of the training data (say 10%), 2) training on the remainder (90%), 3) classifying the holdout data using the 90% trained model, and 4) recording the results.

Generally, hold-outs do not overlap, so, for example, 10 independent trials would be completed for a 10% holdout. Holdouts may be balanced across groups and validation may be averaged over multiple runs, but the key is that in each iteration the classifier is tested on data that was not part of its training. The accuracy will almost certainly be lower than what you compute by applying the model to its training data (a stat we refer to as “class separation”, rather than accuracy), but it will be a much better predictor of how well the classifier will perform in the wild – at least to the degree that your training set resembles the real world. Counter-intuitively, classifiers with weaker class separation often hold up better in K-fold. It is not uncommon that a near perfect accuracy on the training data drops precipitously in K-fold while a slightly weaker classifier maintains excellent generalization performance.

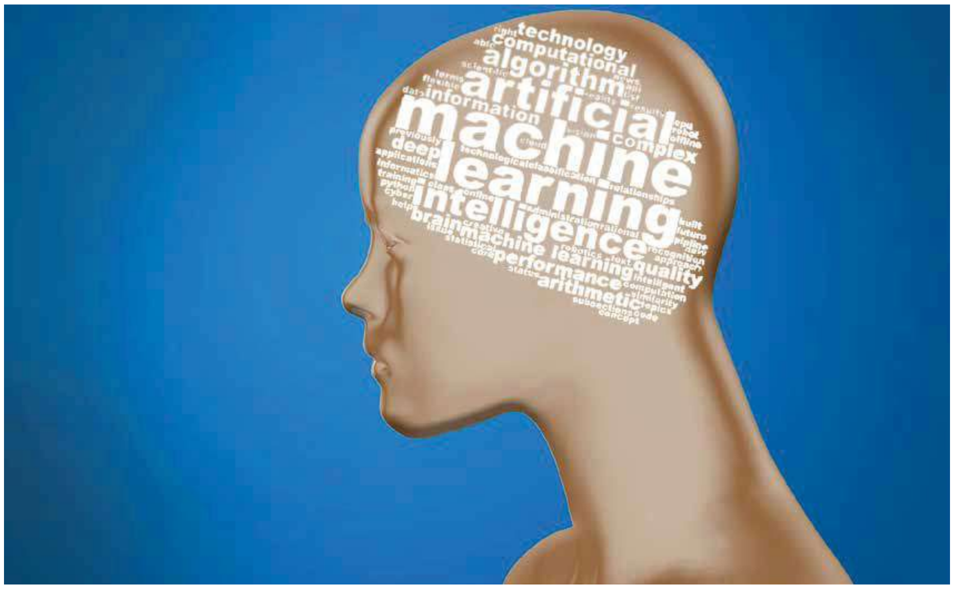
And isn’t that what you’re really after? Better performance in the real world on new observations? Getting high-class separation, but low K-fold? You have a model that has been overtrained, with poor ability to generalize. Back to the drawing board. Maybe select a less aggressive machine learning model, or revisit your feature selection. Reality AI solutions do this automatically. Be careful, though, because the converse is not true: A good K-fold does not guarantee a deployable classifier. The only way to know for sure what you’ve missed in the lab is to test in the wild. Not perfect? No problem: collect more training data capturing more examples of underrepresented variation. A good development tool (like ours) will make it easy to support rapid, iterative improvements of your classifiers.

Lab Experiments vs Real World Products

Lab experiments like this CMU study don’t need to care much about generalization – they are constructed to illustrate a very specific point, prove a concept, and move on. Real-world products, on the other hand, must perform a useful function in a variety of unforeseen circumstances. For machine learning classifiers used in real-world products, the ability generalize is critical. But it’s not the only thing. Deployment considerations matter too. Can it run in the cloud, or is it destined for a processor-, memory- and/or power-constrained environment? (To the CMU guys – good luck getting acceptable battery life out of an overclocked smartwatch!) How computationally intensive is the solution, and can it be run in the target environment with the memory and processing cycles available to it?

What response-time or latency is acceptable? These issues must be factored into a product design, and into the choice of machine-learning model supporting that product. Tools like Reality AI can help. R&D engineers use Reality AI Tools® to create machine learning-based signal classifiers and detectors for real-world products, including wearables and machines and can explore connections between sample rate, computational intensity, and accuracy. They can train new models and run k-fold diagnostics (among others) to guard against overtraining and predictability to generalize. And when they’re done, they can deploy to the cloud, or export code to be compiled for their specific embedded environment. R&D engineers creating real-world products don’t have the luxury of controlled environments – overtraining leads to a failed product. Lab experiments don’t face that reality. Neither do TechCrunch reporters.

How To Succeed with Machine Learning



At Renesas we see a lot of machine learning projects that have failed to get results, or are on the edge of going off the rails. Often, our tools and structured approach can help, but sometimes not.

Here are 3 ways to succeed with Machine Learning:

Number 1: Get Ground Truth

Machine learning isn't a magic wand, and it doesn't work by telepathy. Algorithms need

data and examples of what it is trying to detect, as well as examples of what it is not trying to detect, so that it can tell the difference. This is particularly true of "supervised learning" algorithms, where the algorithm must train on sufficient numbers of examples in order to generate results. But it also applies to "unsupervised learning" algorithms, which attempt to discover hidden relationships in data without being told ahead of time, as well. If relationships of interest don't exist in the data, no algorithm will find them.

Number 2: Curate the Data

Data should be clean and well curated. Meaning that to get the best results, it is important to have faith in the quality of the data. Misclassifications in training data can be particularly damaging in supervised learning situations -- some algorithms (like ours) can compensate for occasional miss-classifications in training data, but pervasive problems can be hard to overcome.

Number 3: Don't Overtrain

Overtraining is a situation where a machine learning model can predict training examples with very high accuracy but cannot generalize to new data, leading to poor performance in the field. Usually, this is a result of too little data, or data that is too homogenous (i.e. does not truly reflect natural variation and confounding factors that will be present in deployment), but it can also result from poor tuning of the model.

Overtraining can be particularly pernicious, as it can lead to false optimism and premature deployment, resulting in a visible failure that could easily have been avoided. Renesas AI engineers oversee and check customer's model configurations to prevent this unnecessary pitfall.

Example: AI for machine health and preventative maintenance

(Names and details have been changed to protect the inexperienced.)

For example, we recently had a client trying to build a machine health monitoring system for a refrigerant compressor. These compressors were installed in a system subject to rare leaks, and they were trying to detect in advance when refrigerant in the lines has dropped to a point that put the compressor at risk -- before it causes damage, overheats, or shuts down through some other mechanism. They were trying to do this via vibration data, using a small device containing a multi-axis accelerometer sensor mounted on the unit.

Ideally, this client would have collected a variety of data with the same accelerometer under known conditions: including many examples of the compressor running in a range of normal load conditions, and many examples of the compressor running under adverse low refrigerant conditions in a similar variety of loads. They could then use our algorithms and tools in confidence that the data contains a broad representation of the operating states of interest, including normal variations as load and uncontrolled environmental factors change. It would also contain a range of different background noises and enough samples so that the sensor and measurement noise is also well represented.

But all they had was 10 seconds of data of a normal compressor and 10 seconds with low refrigerant collected in the lab. This might be enough for an engineer to begin to understand the differences in the two states -- and a human engineer working in the lab might use his or her domain knowledge about field conditions to begin extrapolating how to detect those differences in general. But a machine learning algorithm knows only what it sees. It would make a perfect separation between training examples, showing a 100% accuracy in classification, but that result would never generalize to the real world. In order to consider all the operational variation possible, the most reliable approach is to include examples in the data of a full range of conditions, both normal and abnormal, so that the algorithms can learn by example and tune themselves to the most robust decision criteria.

Reality AI Tools automatically do this by using a variety of methods for feature discovery and model selection. To help detect and avoid overtraining, our tools also test models with "K-fold validation," a process that repeatedly retrains, but holds out a portion of the training data to for testing. This simulates how the model will behave in the field, when it attempts to operate on new observations it had not trained on. K-fold accuracy is almost never as high as training separation accuracy, but it's a better indicator of likely real-world performance – at least to the degree that the training data is representative of the real world.

To understand our machine learning tools more fully and how they can be applied to your data, read our Reality AI Tools technology page.

Machine Learning for Embedded Software is Not As Hard As You Think

Machine Learning is hard and the challenges have been nearly insurmountable for most designers...until now. The modern Renesas Reality AI Tools make it possible to use highly sophisticated machine learning models on all the Renesas processing platforms. From low-end 16-bit (RL78) to 32-bit (RA and RX) and up to 64-bit (RZ) families, end point-capable embedded solutions are available unlike ever before. But there are some considerations to keep in mind.

Has the Machine Learning Module been thoroughly tested and validated?

It will go without saying to experienced embedded engineers, that the decision to commit any code module into an embedded system product must be careful and deliberate.

Firmware pushed to products must be solid, and reliable. Usually, products are expected to have a long lifetime between updates, if updates are even feasible at all. And they are expected to work out of the box. In the case of a machine learning-based classifier or detector module, the usual validation concerns apply, but the caveats we've talked about elsewhere apply in spades: In particular, whether the system has been trained and tested on a large enough variety of REAL WORLD data to give confidence that it will hold up, unsupervised in the wild.

What are the risks and trade-offs of incorrect results?

If you know that the machine learning models results are useful, but not perfect, can customers be educated to work within known limitations? Is the use case safety critical, and if so, what are the fail-safe backups?

Testing and validation is much better done in an easy to use, sandbox environment. Our tools, for example provide the user the ability to experiment, retrain, and test with as much data as they choose before the classifier ever leaves the safety of the cloud. We even support "live" testing through cloud APIs, so that the customer can have every confidence they have tested and characterized a classifier or detector module before ever committing the resources to push it to firmware and customer devices.

Will it translate to your Embedded Hardware?

Processing speed, memory, energy use, physical size, cost, time to deployment: all of these are critically balanced elements in an embedded processing system design. The resources required to deploy the classifier and to execute operations - often within critical real-time latency constraints - will make or break your design. Machine learning approaches like Deep Learning are very powerful, but only if you've got space, power, and budget in your design to support its rather intensive requirements. Other methods can be reduced to sequences of standard signal processing or linear equation type operations at execution time, like ours, may be less general but can comfortably run in real-time on an inexpensive microcontroller. Don't forget the math: it is easy and natural these days to develop detectors and classifiers in high-precision, floating point tools on large desktop or cloud-based systems.

Ensuring the dynamic range and precision is sufficient to reproduce similar performance on, a low-bit depth, fixpoint platform can also put notable constraints on embeddable algorithms.

It is important to know what the requirements will be, as there is no sense spending expensive R&D efforts on a project that cannot possibly be deployed in your target system.

Is there an early feedback mechanism?

Your detector looks good enough to release, but is it perfect? Iteration is the key to perfecting AI classifiers and detectors. Will the designers have access to what really happens in the wild? Can you set up a pilot with one or more friendly customers to give you both quantified validation of the system in a real setting, and, even better, access to data and signals that may have caused problems so that they can be incorporated into future training sets? Remember that most machine learning is data driven. Though you have analyzed the problem and synthesized a wide variety of test vectors back at the lab, you'll never cover all possible variations. Hopefully, you'll find the first-cut classifier or detector is robust and does its job. But few good plans survive first contact with the enemy, and having data from even one or two real customers will quickly introduce situations, noises, systematic errors, and edge cases for which no one fully planned.

How fast can you respond with improvements?

Most companies that ship products with embedded software have a long and (deliberately) onerous review and quality control process for getting new code intro production.

You can't afford to mess up, so validation, walk-throughs, and other process steps necessarily take time and money -- to ensure that only quality goes out the door. Of course this is exactly the opposite of what you need for fast, iterative product improvements. But there are compromises available. Perhaps the classifier design can be reduced to a standard module, which is fully code validated, but operation of which is in part defined by a changeable set of coefficient data (something we support with Reality AI). In a design like this, updates to just the operational coefficients can be pushed, in many cases, without requiring a major code revision and trigger revalidation procedures. Hot updates are comparatively quick and safe, and the results can be revalidated in vivo, so to speak.

So what does it take to embed my classifier?

In the old school world of signal and machine learning R&D, the distance between a laboratory test prototype and an embedded deployment is substantial. The successful custom prototype was only the start of a long process of analysis of requirements and completely re-coding for the target device. But that's changing. If you know you are targeting and embedded platform, and you have some idea of the requirements, modern AI tools can help plan for it from the start. They can be configured to choose from among proven, highly deployable AI module designs, and provide for thorough testing and validating end to end in the cloud. With these tools generating an embedded classifier module may be as simple as linking in a small library function into your code base. An update as simple as pushing a small binary data file to the end device.

Conclusion

Edge AI and TinyML have paved the way for enterprises to build smart product features that use machine learning running on highly constrained edge nodes. The sophisticated Reality AI machine learning tools from Renesas are optimized for real-time analytics of signal data and embedded deployment can cut months or even years from an R&D cycle. Even more importantly, they can often accomplish detections that elude traditional engineering models by making much more efficient and effective use of data to overcome variation.

The complete suite of Reality AI software and tools running on Renesas processors help deliver endpoint intelligence in your products. Learn more, review example projects and request a demo at renesas.com/realityai.

Renesas Electronics Corporation
www.renesas.com

Doc Number: R30WP0007EU0100