Practical TinyMLOps: How to build an MLOps System for TinyML

TinyML is the process of deploying models on microcontrollers and other resource constrained devices. Preparing models for TinyML requires extra steps which a standard MLOps pipeline does not have. Many existing steps need to augmented with TinyML specific implementations or even removed. This is why we need TinyMLOps

In this book we talk about what TinyMLOps is, the different challenges that a TinyML Engineer can face when building such a pipeline and the tools you can use to build one. Finally we show you how to build a TinyMLOps system for your project.

Note: This book is currently a work in progress. You can start by reading about how to build a TinyMLOps pipeline for training models here

Acknowledgements

- Prof Vijay Janapa Reddi (Harvard)
  - Discussions on TinyMLOps
  - Feature Engineering and Data Drifts
- Alessandro Grande, Aurelien Lequertier (Edge Impulse)
  - Discussions on OTA Model Updates
  - Showing Edge Impulse's TinyMLOps features
- Edge Impulse
  - Creating this platform and conference
  - Community work and discussions

What is MLOps

What is TinyML

MLOps Tools and Techniques

What is TinyMLOps?

Building products for TinyML introduces new challenges and tools as well as changes a typical MLOps pipeline by introducing new steps.

One of the challenges is how to update models in deployed devices. If your target device has an internet connection, then you could have the device query an endpoint regularly checking for updates (or you could send a request to the device). But if your TinyML devices are deployed in an area where it cannot get access to internet or any other network, then updating your models, and even monitoring the performance of your device gets challenging.

TinyMLOps are a set of best practices that can help you run machine learning applications on TinyML or edge computing devices successfully.

In the next chapter, we will dive deeper into these challenges, some of the tools available and take a look at the components of a TinyMLOps pipeline.

MLOps for TinyML

Need for TinyMLOps

TinyMLOps Challenges

Tools for TinyMLOps

A Typical TinyMLOps Pipeline

Previously you saw what a MLOps pipeline looks like. Now we will take a look at the components of a TinyMLOps pipeline.

Hardware for TinyMLOps

Data Challenges

Data Collection

Preprocessing Data

Training Models for TinyML

When training models for large systems, we almost always focus on improving the accuracy of the model. Two methods that will generally increase your model's accuracy are to either use more high quality data or to use larger model architectures with complex layers, operations and data flows. In the first case, we have already talked about how data can be collected and preprocessed as well as how we can apply feature engineering techniques to improve the performance of TinyML models.

However, unlike MLOps, when we train models for TinyML we cannot use large and complex custom layers. In fact, we cannot use most of the commonly available model architectures or layers and operations that are typically used in large models since they are not supported on many TinyML devices. This creates a dilemma since we need to train models such that they can maintain appreciable accuracy while still being computationally simple and occupy less memory.

In this chapter, we will learn more about how we can design and train models such that we can get the least accuracy degradation while still being computationally cheap when we deploy it to a TinyML device.

Challenges in Training TinyML Models

When ML engineers train models, they usually do not think about where the model will be deployed. Their models are usually deployed in large multi-core systems with support for many instruction sets, compilers and toolkits that can abstract away the complications of the underlying hardware. Moreover, if a model does have poor execution speeds, it is easy to either add more compute resources (like more CPU cores or GPUs) to it or parallelize the execution of the model across multiple devices to improve its latency.

However this is not true in the case of TinyML where the final hardware will be resource constrained and have limited instruction set support. Therefore, for TinyML, the final hardware is a bottleneck that we need to keep in mind when training our models. Unfortunately for us, the constraints of the deployment hardware creates a domino effect that affects not only our final model's accuracy, but also the kind of layers we can use and how secure and robust we can make our model.

Model Accuracy

TinyML devices have less computational power and they do not support many operations. Moreover they do not have a lot of memory. This means that the models we deploy to TinyML devices need to be small, efficient and simple. However these qualities are completely opposite to what makes models accurate. And this is one of the biggest challenges in

training TinyML models: How do we make models tiny, while still making sure that they are accurate enough to perform our tasks?

Model Size

To execute a neural network in a TinyML device, we need to load the model architecture and weights into memory and also be able to store intermediate activations when executing the model. Since TinyML devices usually have less memory, we need to make sure that the sum of size of the model, its parameters (weights, biases and other internal values) and the largest generated activation can fit in the memory at the same time. While larger model architectures are more accurate, they also have more parameters and generate larger activations. Larger models will not only take more time to load to memory, but it will also take longer to execute since more operations need to performed. So even if your model can fit into memory, a larger model will have more latency. So when we train our model, we need to make sure that the model size is small such that we can load it into the final hardware we are deploying our model in and its latency is low.

Model Architecture

Another factor affecting latency is the architecture of the model. Models with

Model Security and Robustness

Finally, many TinyML devices are deployed in remote or open places where they can be affected by environmental factors like sunlight, temperature, dust as well as be prone to physical attacks (for instance to steal model details). For instance, a model that checks for wildlife in a jungle needs to work not only in the morning and night, but also at times with low light, changing shadows, rain, dust and so on. Lately techniques have been developed to train our models such that they are more robust to such environmental changes and to sensor or battery degradation.

Since many deployed models may have been trained with sensitive data or may have IP, they can become targets for attackers looking to steal the model architecture and weights. Recent work in side-channel attacks have shown that this is not only possible to be done with high accuracy, but it can be also done quickly and with cheap easily available tools. Thankfully, there are also some ways to keep create models that can make such attacks either difficult or impossible to do.

We will talk more about model security and robustness in a later chapter, but now let’s take a look at how we can train models for TinyML while solving these challenges.

Techniques for Training TinyML Models

Previously we looked at some of the challenges of training TinyML models. In short, we need to train our models while recognizing the constraints of the hardware we are deploying to and the overall systems requirements like latency and accuracy.

As engineers, if we train our models while being *aware* of our final hardware limitations, then we can take steps to reduce our model’s performance drop when we deploy our model. Most of our model’s performance will drop at two steps in our TinyML pipeline: when we optimize our model and when we deploy our model.

Optimization reduces either the size or the number of operations in our model and it is the main place where our model’s performance can drop. Deploying can reduce our model’s performance due to constraints that are mostly out of our control like the size and speed of the memory, the presence of floating point units and the types of ops supported. However we can take into account these hardware constraints when we build our model architecture to make our system more efficient.

The Look-Ahead Method

I like to call this technique of being aware of our entire pipeline the “Look-Ahead” method. Being able to look-ahead when building our system is an extremely powerful tool. However it is not possible to do it if we have not at least planned for and finallized the broad requirements and steps in our TinyMLOps pipeline.

Some of these steps will be constant for almost all pipelines. For instance we will almost always have to optimize our trained model for TinyML with algorithms like quantization or pruning. So we can train our initial model in a way such that the accuracy drop after optimization is reduced. We will look more into these techniques in a bit.

Other steps in a machine learning pipeline may be dependent on other factors like client requirements or the choice of hardware. In these cases it is not possible to be able to look-ahead and take into consideration any constraints the system may have when training our model. The easiest way to get around this is to finalize these requirements and decisions before training models.

todo

You could go the other way around where you choose a hardware based on how big a model is, but this is not always going to be the right decision in the long-run. This is mostly because models are software based and can be updated more easily and frequently (we will talk more about this in a later chapter) as compared to hardware. New model architectures and data preprocessing and optimization techniques are invented every few weeks that will improve the performance of your model. Moreover as you run your system and collect more data, you can retrain your model to improve its performance and eventually train smaller architectures that run more efficiently.

However, if you choose a more powerful hardware that consumes more power or has a larger form-factor than the requirements, then updating your hardware can be a very painful process, especially if you already have a fleet of many TinyML devices already deployed. Moreover, to make your system efficient, your software will be tightly coupled with your device and changing it could also result in you having to rethink your model and other software systems as well.

Optimization Look-Ahead

Tools for Training TinyML Models

Pipeline for Training TinyML Models

Optimizing Models for TinyML

Challenges in Optimizing TinyML Models

Techniques for Optimizing TinyML Models

Tools for Optimizing TinyML Models

Pipeline for Optimizing TinyML Models

What is Benchmarking

Challenges in Benchmarking TinyML Systems

How to Benchmark TinyML Systems

Pipeline for Benchmarking TinyML Systems

Deploying TinyML Systems

Challenges in Deploying TinyML Systems

How to Deploy TinyML Systems

Pipeline for Deploying TinyML Systems

Monitoring and Updating TinyML Systems

Challenges in Monitoring and Updating TinyML Systems

Techniques for Monitoring and Updating TinyML Systems

Pipeline for Monitoring and Updating TinyML Systems

Secure and Robust TinyML Systems

Challenges in Designing Secure and Robust TinyML Systems

Creating Secure and Robust TinyML Systems

Architecture and Pipeline

By Soham Chatterjee