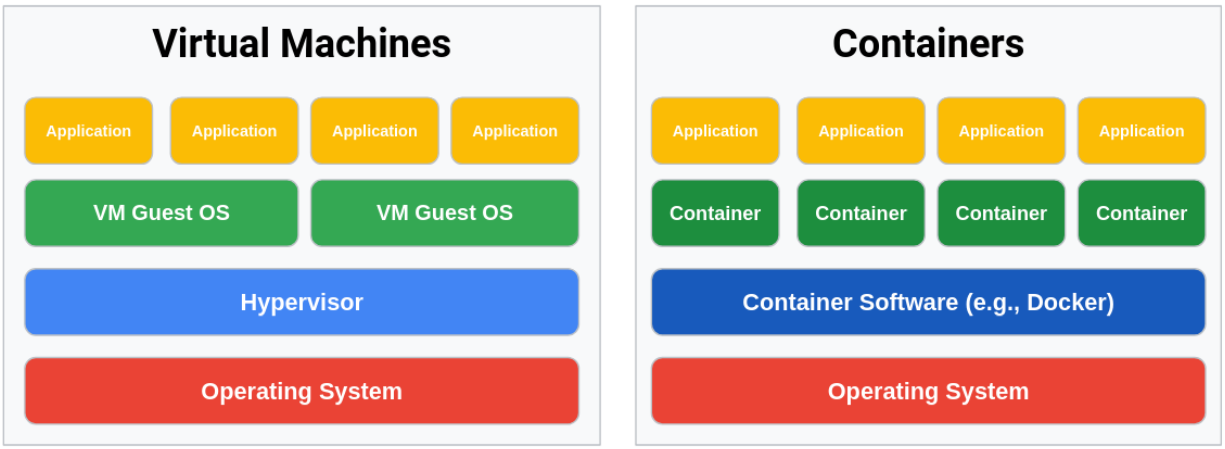
Docker vs. Virtual Machines (vs. Virtual Environments)



About this Reading

In this reading, we get a quick overview of dockers, virtualization and virtual environments. We want to learn about these things so that we can understand how they are useful for ML model deployments. And this ultimately leads us to the question. *Do we have such things for TinyML?*

Docker, virtual machines (VMs), and virtual environments are all powerful computing resources at our disposal, but are often confused or poorly understood. This is likely because the difference between them is nuanced and non-obvious even when you have used them. In this reading, we will try to get a clearer understanding of Docker, VMs, and virtual environments, and the pros and cons of each technology.

Docker and VMs are virtualization technologies that allow a user to run separately running systems within a single host system, allowing the segmentation of computational resources. However, the way Docker and VMs achieve this virtualization is slightly different.

Virtual Machines

VMs achieve this at the operating system level, allowing multiple different operating systems to be run on a single machine's hardware. The VM operating system instances are run on a hypervisor, which manages resources across the different operating systems. VMs allow all of the compute resources given to a particular operating system instance to be used across all applications running on the virtual machine. If we have ten apps, they will all share the resources as if we were running it on our desktop computer. We can essentially think of VMs as multiple physical computers compressed into one, instead of needing to buy both a Mac and a Windows, we need only buy one and run both operating systems on top of it.

Docker

Docker achieves a similar goal but at a higher level of abstraction, often called containerization, segmenting computational resources within a single operating system. This is similar to the way that we can have multiple Jupyter Notebooks running on our computer, each with their own separate kernel and some resources attached to the runtime. However, Docker goes one step further than the Jupyter notebook, allowing us to run different operating systems on our main system through something called a container engine (kind of like the hypervisor, but it runs on the operating system and manages the containers). As an example, let's imagine that I have a MacBook Pro and am interested in running an Apache Web server as well as an Ubuntu instance. The Ubuntu instance will be used to run a complex machine learning model which requires a lot of special setup and is not compatible with MacOS. In this case, we can take a Dockerfile with all of the configuration details, outlining how to configure system-level dependencies (not just Python dependencies), and run it on our system. This Dockerfile will allow us to create a barebones Ubuntu instance which is perfectly configured to run our code, despite the fact that we are running on a MacOS.

What's The Difference?

Virtual machines have been around for quite a while, and are the basis for most cloud computing, but containers have become very popular in recent years, especially in the academic community as they provide a useful way of quickly verifying publication results and building upon code related to publications. Containers are great in a number of ways: they are very fast to spin up compared to virtual machines, and are smaller in size since only necessary functionality is added to the container. They are also easier to manage and configure, as they can be done using Infrastructure as Code tools such as Ansible. There are also online repositories with thousands of useful Dockerfiles that can be downloaded to run various operating systems, servers, and models. A popular online repository is DockerHub. However, one downside to containers is that applications cannot share resources in the same way as on a virtual machine. If I have ten containers, I must give each a designated amount of resources, and if one needs more resources than I have available, it will crash. Thus, there are load balancing advantages to using virtual machines, as well as some security benefits. However, for most non-specialized uses, containers are often preferable.

Where Do Virtual Environments Fit In?

In your education journey so far, you may have also come across the term virtual environment, which is also related to Docker and VMs and thus worth explaining. A virtual environment is like a container for a particular Python environment. It manages all of our dependencies and ensures that we do not get stuck in "dependency hell" where none of our code will run due to incompatible libraries. The most popular virtual environment today is Pipenv, which comes in the form of a "Pipfile" and a "Pipfile.lock". The former stores a list of all the dependencies, while the latter stores the dependencies along with hashes of their libraries.

You can think of a virtual environment as a Docker container which only manages Python dependencies. In fact, many Docker containers used for machine learning often utilize Pipenv within them to manage their Python dependencies!

The difference between these three is a common source of confusion among students of ML, and thus we recommend spending some time becoming familiar with the differences between them to make sure you understand the differences. If you are feeling confident, maybe try running a virtual environment within a couple of Docker containers, where one of each is placed on a separate virtual machine - good luck!

Why Dockers for ML Deployment?

Now that we have a basic understanding of Dockers, lets see why they are useful for machine learning deployment? Creating a machine learning model for our computer is simple. But deploying the model at scale, on various types of servers globally, is more challenging and difficult. It is highly possible that if you design a model and then move it to production or another server, it may not operate properly on other systems. Many things can go wrong. You can run into performance issues, crashes, and even poor optimization. Even while our machine learning model can be implemented in a single programming language (e.g. Python), the application will need to connect with apps written in other programming languages. With Docker, you get granular updates and reproducibility. With Docker, you can shrink wrap things as "microservices," allowing for scalability and easy addition or deletion of independent services.

When a model is complete, the ML engineer or the data scientist responsible often fears that it will not reproduce real-life findings or that it will be shared with teammates. Sometimes it's not the model, but the need to recreate the entire stack. Docker allows you to simply replicate the machine learning model training and testing environment anywhere. A training model can be built locally and quickly moved to external clusters with greater GPUs, memory, or CPU power. Moreover, it is easy to distribute the model globally by packaging it as an API in a container and using technology like OpenShift, a Kubernetes distribution (which we will learn about a bit later).

Containerization of ML applications is also simple since we can develop containers using templates and use an open-source registry of user-contributed containers. Docker allows developers to track container image versions, see who produced them, and rollback to prior versions. Finally, your machine learning program can execute even if one of its services is updating, fixing, or unavailable. For example, updating an output message integrated in the entire solution does not necessitate updating the entire program and disrupting other services.

Hopefully now you see how Dockers are powerful constructs for ML deployment. But do such constructs exist for TinyML deployments that don't have even a bare metal operating system? We'll dig into that next.