Chapter 3. MLOps for Containers and Edge Devices

By Alfredo Deza

> *Split-brain experiments started with the problem of interocular transfer. That is, if one learns with one eye how to solve a problem, then with that eye covered and using the other eye, one readily solves the problem without further learning. This is called "interocular transfer of learning." Of course, the learning is not in the eye and then transferred to the other eye, but that is the way it is usually described. The fact that transfer occurs may seem obvious, but it is in the questioning of the obvious that discoveries are often produced. In this case, the question was: How can the learning with one eye appear with use of the other? Put in experimentally testable terms, where are the two eyes connected? Experiments showed that the transfer actually occurs between the hemispheres by way of the corpus callosum.*
>
> —Dr. Joseph Bogen

When I started in technology, virtual machines (virtualized servers hosted in a physical machine), were well positioned and pervasive—it was easy to find them everywhere, from hosting providers to regular companies with big servers in the IT room. A lot of online software providers were offering virtualized hosting. At work, I honed my skills, trying to learn as much as possible about virtualization. The ability to run virtual machines within some other host offered a lot of (welcomed) flexibility.

Whenever a new technology solves a problem (or any number of issues really), a trail of other problems tags along to be resolved. With virtual machines, one of these problems was to deal with moving them around. If host server *A* needed to have a new operating system installed, a system administrator would need to move the virtual machines over to host server *B*. Virtual machines were as large as the data when initially configured: a 50 GB *virtual drive* meant a file representing the virtual drive existed at 50 gigabytes. Moving around 50 gigabytes from one server to the other would take time. If you are moving around a critical service running a virtual machine, how can you minimize downtime?

Most of these issues had their strategies to minimize downtime and increase robustness: snapshots, recovery, backups. Software projects like the *Xen Project* and *VMWare* specialized in making these issues relatively easy to solve, and cloud providers practically eliminated them.

These days, virtual machines still have an important place in cloud offerings. Google Cloud calls them *Compute Engine*, for example, and other providers have a similar reference. A lot of these virtual machines offer enhanced GPUs to provide better performance targeted at machine learning operations.

Although virtual machines are here to stay, it is increasingly important to grasp two types of technologies for model deployments: containers and

edge devices. It is unreasonable to think that a virtual machine would be well suited to run on an edge device (like a cellphone) or quickly iterate during development with a reproducible set of files. You will not always face a decision of using one or the other, but possessing a clear understanding of these options (and how they operate) will make you a better machine learning engineer.

Containers

With all the power and robustness of virtual machines, it is critical to grasp containers and containerization technology in general. I remember being in the audience at PyCon in Santa Clara in 2013, when Docker was announced. It felt incredible! The *lean virtualization* demoed was not new for Linux. What was new and sort of revolutionary was the tooling. Linux has had *LXC* (or *Linux containers*), which provided a lot of the functionality we take for granted with containers today. But the tooling for LXC is dismal, and Docker brought one key factor to successfully become a leader: easy collaboration and sharing through a registry.

Registries allow any developer to *push* their changes to a central location where others can then *pull* those changes and run them locally. Registry support with the same tooling that deals with containers (and all seamlessly) propelled the technology forward at an incredible pace.

TIP

For this section, make sure you have a container runtime installed. For the examples in this section, it might be easier to use Docker. After installation, ensure that the `docker` command will show the help output to verify a successful install.

One of the most significant descriptions I've seen about containers in comparison to virtual machines comes from Red Hat. In short, containers are all about the application itself, and only what the application is (like the source code and other supporting files) versus what it needs to run (like databases). Traditionally, engineers often use virtual machines like an all-in-one service where the database, the web server, and any other system service are installed, configured, and run. These types of applications are *monolithic*, with tied interdependencies in an all-in-one machine.

A *microservice*, on the other hand, is an application that is fully decoupled from system requirements like databases and can run independently. Although you can use virtual machines as microservices, it is more common to find containers fitting better in that concept.

If you are already familiar with creating and running containers, in "Infrastructure as Code for Continuous Delivery of ML Models" I cover how to build them programmatically with pretrained models, which takes these concepts and enhances them with automation.

Container Runtime

You might've noticed that I've mentioned containers, Docker, and container runtime. These terms might get confusing, primarily when people use them interchangeably. Since Docker (the company) initially devel-

oped the tooling to create, manage, and run containers, it became common to say "Docker container." The runtime—that is, the required software to run a container in a system—was also created by Docker. A few years after the new container technology's initial release, Red Hat (the company behind the RHEL operating system) contributed to making a different way to run containers, with a new (alternative) runtime environment. This new environment also brought a new set of tools to operate containers, with some compatibility with those provided by Docker. If you ever hear about container runtime, you have to be aware that there is more than one.

Some benefits of these new tools and runtime mean that you are no longer required to use a superuser account with extensive privileges, which makes sense for many different use cases. Although Red Hat and many other open source contributors have done an excellent job with these tools, it is still somewhat complicated to run them in operating systems that aren't Linux. On the other hand, Docker makes the work a seamless experience regardless of whether you are using Windows, MacOS, or Linux. Let's get started by going through all the steps needed to create a container.

Creating a Container

The *Dockerfile* is at the heart of creating containers. Anytime you are creating a container, you must have the Dockerfile present in the current directory. This special file can have several sections and commands that allow creating a container image. Open a new file and name it *Dockerfile* and add the following contents to it:

```
FROM centos:8

RUN dnf install -y python38
```

The file has two sections; a unique keyword delimits each one. These keywords are known as *instructions*. The beginning of the file uses the `FROM` instruction, which determines what the base for the container is. The *base* (also referred to as *base image*) is the CentOS distribution at version 8. The version, in this case, is a tag. Tags in containers define a point in time. When there are no tags defined, the default is the *latest* tag. It is common to see versions used as tags, as is the case in this example.

One of the many useful aspects of containers is that they can be composed of many layers, and these layers can be used or reused in other containers. This layering workflow prevents a base layer of 10 megabytes from getting downloaded every time for each container that uses it. In practice, you will download the 10 megabyte layer once and reuse it many times. This is *very* different from virtual machines, where it doesn't matter if these virtual machines all have the same files: you are still required to download them all as a whole.

Next, the `RUN` instruction runs a system command. This system command installs Python 3, which isn't included in the base CentOS 8 image. Note how the `dnf` command uses the `-y` flag, which prevents a prompt for confirmation by the installer, triggered when building the container. It is crucial to avoid any prompts from running commands as it would halt the build.

Now build the container from the same directory where the *Dockerfile* is:

```
$ docker build .
[+] Building 11.2s (6/6) FINISHED
 => => transferring context: 2B
 => => transferring dockerfile: 83B
 => CACHED [1/2] FROM docker.io/library/centos:8
 => [2/2] RUN dnf install -y python38
 => exporting to image
 => => exporting layers
 => => writing
image sha256:3ca470de8dbd5cb865da679ff805a3bf17de9b34ac6a7236dbf0c367e1fb4610
```

The output reports that I already have the initial layer for CentOS 8, so there is no need to pull it again. Then, it installs Python 3.8 to complete image creation. Make sure you start a build pointing to where the *Dockerfile* is present. In this case, I'm in the same directory, so I use a dot to let the build know that the current directory is the one to build from.

This way of building images is not very robust and has a few problems. First, it is challenging to identify this image later. All we have is the sha256 digest to reference it and nothing else. To see some information about the image just built, rerun `docker`:

```
$ docker images
docker images
REPOSITORY    TAG       IMAGE ID            CREATED             SIZE
<none>        <none>    3ca470de8dbd        15 minutes ago      294MB
```

There is no repository or tag associated with it. The image ID is the digest, which gets reduced to only 12 characters. This image is going to be challenging to deal with if it doesn't have additional metadata. It is a good practice to tag it when building the image. This is how you create the same image and tag it:

```
$ docker build -t localbuild:removeme .
[+] Building 0.3s (6/6) FINISHED
[...]
 => => writing
image sha256:4c5d79f448647e0ff234923d8f542eea6938e0199440dfc75b8d7d0d10d5ca9a
 => => naming to docker.io/library/localbuild:removeme
```

The critical difference is that now `localbuild` has a tag of `removeme` and it will show up when listing the images:

```
$ docker images localbuild
REPOSITORY       TAG             IMAGE ID            CREATED             SIZE
localbuild       removeme        3ca470de8dbd        22 minutes ago      294MB
```

Since the image didn't change at all, the build process was speedy, and internally, the build system tagged the already built image. The naming and tagging of images helps when pushing the image to a registry. I would need to own the *localbuild* repository to push to it. Since I don't, the push will get denied:

```
$ docker push localbuild:removeme
The push refers to repository [docker.io/library/localbuild]
```

```
denied: requested access to the resource is denied
```

However, if I re-tag the container to my repository in the registry, pushing will work. To re-tag, I first need to reference the original tag (`localbuild:removeme`) and then use my registry account and destination (`alfredodeza/removeme`):

```
$ docker tag localbuild:removeme alfredodeza/removeme
$ docker push alfredodeza/removeme
The push refers to repository [docker.io/alfredodeza/removeme]
958488a3c11e: Pushed
291f6e44771a: Pushed
latest: digest: sha256:a022eea71ca955cafb4d38b12c28b9de59dbb3d9fcb54b size: 741
```

Going to the registry (in this case, Docker Hub) now shows that the recently pushed image is available (Figure 3-1).

Since my account is open and the registry is not restricting access, anyone can "pull" the container image by running: `docker pull alfredodeza/removeme` . If you've not been exposed to containers or registries before, this should feel revolutionary. Like I mentioned at the beginning of this chapter, it is the foundation of why containers went viral in the developer community. The answer to *"How do I install your software?"* can now be *"just pull the container"* for almost anything.

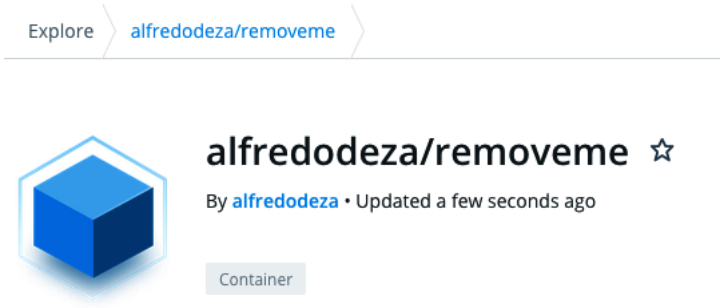

Figure 3-1. Docker Hub image

Running a Container

Now that the container builds with the *Dockerfile*, we can run it. When running virtual machines, it was common practice to enable the SSH (also known as the Secure Shell) daemon and expose a port for remote access, and perhaps even add default SSH keys to prevent getting password prompts. People who aren't used to running a container will probably ask for SSH access to a running container instance. SSH access is not needed; even though you can enable it and make it work, it is not how to access a running container.

Make sure that a container is running. In this example, I run CentOS 8:

```
$ docker run -ti -d --name centos-test --rm centos:8 /bin/bash
1bb9cc3112ed6615116635172498 98bfc9524fc02dedc3ce40b5c4cb982d7bcd
```

There are several new flags in this command. It uses `-ti` to allocate a TTY (emulates a terminal) and attaches `stdin` to it to interact with it in the terminal later. Next, the `-d` flag makes the container run in the back-

ground to prevent taking control of the current terminal. I assign a name (`centos-test`) and then use `--rm` so that Docker removes this container after stopping it. After issuing the command, a digest returns, indicating that the container has started. Now, verify it is running:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND             NAMES
1bb9cc3112ed        centos:8            "/bin/bash"         centos-test
```

Some containers get created with an `ENTRYPOINT` (and optionally a `CMD`) instruction. These instructions are meant to get the container up and running for a specific task. In the example container we just built for CentOS, the `/bin/bash` executable had to be specified because otherwise the container would not stay running. These instructions mean that if you want a long-running container, you should create it with at least an `ENTRYPOINT` that executes a program. Update the *Dockerfile* so that it looks like this:

```
FROM centos:8

RUN dnf install -y python38

ENTRYPOINT ["/bin/bash"]
```

Now it is possible to run the container in the background without the need to specify the `/bin/bash` command:

```
$ docker build -t localbuild:removeme .
$ docker run --rm -it -d localbuild:removeme
023c8e67d91f3bb3998e7ac1b9b335fe20ca13f140b6728644fd45fb6ccb9132
$ docker ps
CONTAINER ID        IMAGE               COMMAND             NAMES
023c8e67d91f        removeme            "/bin/bash"         romantic_khayyam
```

I mentioned before how it is common to use SSH to gain access to a virtual machine and that gaining access to a container is somewhat different. Although you could enable SSH for a container (in theory), I don't recommend it. This is how you can get access to a running container using the container ID and the `exec` subcommand:

```
$ docker exec -it 023c8e67d91f bash

[root@023c8e67d91f /]# whoami
root
```

In that case, I have to use the command I want to run. Since I want to interactively manipulate the container (as I would with a virtual machine), I call out the executable for the Bash (a ubiquitous Unix shell and command language) program.

Alternatively, you may not want to gain access using an interactive shell environment, and all you want to do is run some command. The command must change to achieve this. Replace the shell executable used in the previous example with that of the command to use:

```
$ docker exec 023c8e67d91f tail /var/log/dnf.log
```

```
    python38-setuptools-wheel-41.6.0-4.module_el8.2.0+317+61fa6e7d.noarch

2020-12-02T13:00:04Z INFO Complete!
2020-12-02T13:00:04Z DDEBUG Cleaning up.
```

Since there is no interactive need (I'm not sending any input via a shell), I can omit the `-it` flag.

NOTE

One common aspect of container development is to keep its size as small as possible. That is why a CentOS container will have a lot fewer packages than that of a newly installed CentOS virtual machine. This leads to surprising experiences when you expect a package to be present (e.g., a text editor like Vim) and it isn't.

Best Practices

The first thing I do (and highly recommend) when trying a new language or tool is to find a linter that can help navigate conventions and common usage that I may not be familiar with. There are a few linters for creating containers with a Dockerfile. One of these linters is `hadolint`. It is conveniently packaged as a container. Modify the last *Dockerfile* example, so it looks like this:

```
FROM centos:8

RUN dnf install -y python38

RUN pip install pytest

ENTRYPOINT ["/bin/bash"]
```

Now run the linter to see if there are any good suggestions:

```
$ docker run --rm -i hadolint/hadolint < Dockerfile
DL3013 Pin versions in pip.
  Instead of `pip install <package>` use `pip install <package>==<version>`
```

This is a good suggestion. Pinning packages is always a great idea because you are safe from an update in a dependency being incompatible with the code your application needs. Be aware that pinning dependencies and never going through the chore of updating them isn't a great idea. Make sure you come back to pinned dependencies and see if it would be useful to update.

Since one of the goals of containerizing tools is to keep them as small as possible, you can accomplish a couple of things when creating a Dockerfile. Every time there is a `RUN` instruction, a new layer gets created with that execution. Containers consist of individual layers, so the fewer the number of layers, the smaller the container's size. This means that it is preferable to use a single line to install many dependencies instead of one:

```
RUN apk add --no-cache python3 && python3 -m ensurepip && pip3 install pytest
```

The use of `&&` at the end of each command chains everything together, creating a single layer. If the previous example had a separate `RUN` instruction for each install command, the container would end up being larger. Perhaps for this particular example, the size wouldn't make that much of a difference; however, it would be significant in containers that require lots of dependencies.

There is a useful option that linting can offer: the opportunity to automate the linting. Be on the lookout for chances to automate processes, removing any manual step and letting you concentrate on the essential pieces of the process of shipping models into production (writing a good Dockerfile in this case).

Another critical piece of building containers is making sure there aren't vulnerabilities associated with the software installed. It is not uncommon to find engineers who think that the application is unlikely to have vulnerabilities because they write high-quality code. The problem is that a container comes with preinstalled libraries. It is a full operating system that, at build time, will pull extra dependencies to satisfy the application you are trying to deliver. If you are going to serve a trained model from the container using a web framework like Flask, you have to be well aware that there might be Common Vulnerabilities and Exposures (CVEs) associated with either Flask or one of its dependencies.

These are the dependencies that Flask (at version `1.1.2`) brings:

```
click==7.1.2
itsdangerous==1.1.0
Jinja2==2.11.2
MarkupSafe==1.1.1
Werkzeug==1.0.1
```

CVEs can get reported at any given time, and software systems used to alert on vulnerabilities ensure they are updated several times during the day to report accurately when that happens. A critical piece of your application like Flask may not be vulnerable today for version 1.1.2, but it can undoubtedly be tomorrow morning when a new CVE is discovered and reported. Many different solutions specialize in scanning and reporting vulnerabilities in containers to mitigate these vulnerabilities. These security tools scan a library that your application installs and the operating system's packages, providing a detailed and accurate vulnerability report.

One solution that is very fast and easy to install is Anchore's `grype` command line tool. To install it on a Macintosh computer:

```
$ brew tap anchore/grype
$ brew install grype
```

Or on any Linux machine:

```
$ curl -sSfL \
 https://raw.githubusercontent.com/anchore/grype/main/install.sh | sh -s
```

Using `curl` in this way allows deploying `grype` into most any continuous integration system to scan for vulnerabilities. The `curl` installation

method will place the executable in the current working path under a *bin/* directory. After installation is complete, run it against a container:

```
$ grype python:3.8
 ✓ Vulnerability DB     [no update available]
 ⠦ Loading image        ━━━━━━━━━━━━━━━━━━━━ [requesting image from docker]
 ✓ Loaded image
 ✓ Parsed image
 ✓ Cataloged image      [433 packages]
 ✓ Scanned image        [1540 vulnerabilities]
```

Over a thousand vulnerabilities look somewhat surprising. The output is too long to capture here, so filter the result to check for vulnerabilities with a severity of *High*:

```
$ grype python:3.8 | grep High
[...]
python2.7      2.7.16-2+deb10u1      CVE-2020-8492      High
```

There were a few vulnerabilities reported, so I reduced the output to just one. The CVE is worrisome because it can potentially allow the system to crash if an attacker exploits the vulnerability. Since I know the application uses Python 3.8, then this container is not vulnerable because Python 2.7 is unused. Although this is a Python 3.8 container, the image contains an older version for convenience. The critical difference is that now you know what is vulnerable and can make an executive decision for the eventual deployment of the service to production.

A useful automation enhancement is to fail on a specific vulnerability level, such as `high`:

```
$ grype --fail-on=high centos:8
[...]
discovered vulnerabilities at or above the severity threshold!
```

This is another check that you can automate along with linting for robust container building. A well-written *Dockerfile* with constant reporting on vulnerabilities is an excellent way to enhance containerized models' production delivery.

Serving a Trained Model Over HTTP

Now that a few of the core concepts of creating a container are clear, let's create a container that will serve a trained model over an HTTP API using the Flask web framework. As you already know, everything starts with the Dockerfile, so create one, assuming for now that a *requirements.txt* file is present in the current working directory:

```
FROM python:3.7

ARG VERSION

LABEL org.label-schema.version=$VERSION

COPY ./requirements.txt /webapp/requirements.txt

WORKDIR /webapp
```

```
RUN pip install -r requirements.txt

COPY webapp/* /webapp

ENTRYPOINT [ "python" ]

CMD [ "app.py" ]
```

There are a few new things in this file that I have not covered before. First, we define an argument called `VERSION` that gets used as a variable for a `LABEL`. I'm using a label schema convention that is useful to normalize how these labels are named. Using a version is a helpful way of adding informational metadata about the container itself. I will use this label later when I want to identify the version of the trained model. Imagine a situation where a container is not producing expected accuracy from a model; adding a label helps identify the problematic model's version. Although this file uses one label, you can imagine that the more labels with descriptive data, the better.

NOTE

There is a slight difference in the container image used. This build uses Python 3.7 because at the time of writing, some of the dependencies do not work yet with Python 3.8. Feel free to swap 3.7 for 3.8 and check if it now works.

Next, a *requirements.txt* file gets copied into the container. Create the requirements file with the following dependencies:

```
Flask==1.1.2
pandas==0.24.2
scikit-learn==0.20.3
```

Now, create a new directory called *webapp* so that the web files are contained in one place, and add the *app.py* file so that it looks like this:

```
from flask import Flask, request, jsonify

import pandas as pd
from sklearn.externals import joblib
from sklearn.preprocessing import StandardScaler

app = Flask(__name__)

def scale(payload):
    scaler = StandardScaler().fit(payload)
    return scaler.transform(payload)

@app.route("/")
def home():
    return "<h3>Sklearn Prediction Container</h3>"

@app.route("/predict", methods=['POST'])
def predict():
    """
    Input sample:

        {
            "CHAS": { "0": 0 }, "RM": { "0": 6.575 },
```

```
            "TAX": { "0": 296 }, "PTRATIO": { "0": 15.3 },
            "B": { "0": 396.9 }, "LSTAT": { "0": 4.98 }
        }

    Output sample:

        { "prediction": [ 20.35373177134412 ] }
    """

    clf = joblib.load("boston_housing_prediction.joblib")
    inference_payload = pd.DataFrame(request.json)
    scaled_payload = scale(inference_payload)
    prediction = list(clf.predict(scaled_payload))
    return jsonify({'prediction': prediction})

if __name__ == "__main__":
    app.run(host='0.0.0.0', port=5000, debug=True)
```

The last file needed is the trained model. If you are training the Boston Housing prediction dataset, make sure to place it within the *webapp* directory along with the *app.py* file and name it *boston_housing_prediction.joblib*. You can also find a trained version of the model in this GitHub repository.

The final structure of the project should look like this:

```
.
├── Dockerfile
└── webapp
    ├── app.py
    └── boston_housing_prediction.joblib

1 directory, 3 files
```

Now build the container. In the example, I will use the run ID that Azure gave me when I trained the model as the version to make it easier to identify where the model came from. Feel free to use a different version (or no version at all if you don't need one):

```
$ docker build --build-arg VERSION=AutoML_287f444c -t flask-predict .
[+] Building 27.1s (10/10) FINISHED
 => => transferring dockerfile: 284B
 => [1/5] FROM docker.io/library/python:3.7
 => => resolve docker.io/library/python:3.7
 => [internal] load build context
 => => transferring context: 635B
 => [2/5] COPY ./requirements.txt /webapp/requirements.txt
 => [3/5] WORKDIR /webapp
 => [4/5] RUN pip install -r requirements.txt
 => [5/5] COPY webapp/* /webapp
 => exporting to image
 => => writing image sha256:5487a63442aae56d9ea30fa79b0c7eed1195824aad7ff4ab42b
 => => naming to docker.io/library/flask-predict
```

Double-check that the image is now available after building:

```
$ docker images flask-predict
REPOSITORY      TAG       IMAGE ID       CREATED         SIZE
flask-predict   latest    5487a63442aa   6 minutes ago   1.15GB
```

Now run the container in the background, exposing port `5000`, and verify it is running:

```
$ docker run -p 5000:5000 -d --name flask-predict flask-predict
d95ab6581429ea79495150bea507f009203f7bb117906b25ffd9489319219281
$docker ps
CONTAINER ID IMAGE         COMMAND         STATUS       PORTS
d95ab6581429 flask-predict "python app.py" Up 2 seconds 0.0.0.0:5000->5000/tcp
```

On your browser, open *http://localhost:5000* and the HTML from the `home()` function should welcome you to the Sklearn Prediction application. Another way to verify this is wired up correctly is using `curl`:

```
$ curl 192.168.0.200:5000
<h3>Sklearn Prediction Container</h3>
```

You can use any tool that can send information over HTTP and process a response back. This example uses a few lines of Python with the `requests` library (make sure you install it before running it) to send a POST request with the sample JSON data:

```
import requests
import json

url = "http://localhost:5000/predict"

data = {
    "CHAS": {"0": 0},
    "RM": {"0": 6.575},
    "TAX": {"0": 296.0},
    "PTRATIO": {"0": 15.3},
    "B": {"0": 396.9},
    "LSTAT": {"0": 4.98},
}

# Convert to JSON string
input_data = json.dumps(data)

# Set the content type
headers = {"Content-Type": "application/json"}

# Make the request and display the response
resp = requests.post(url, input_data, headers=headers)
print(resp.text)
```

Write the Python code to a file and call it *predict.py*. Execute the script to get some predictions back on the terminal:

```
$ python predict.py
{
  "prediction": [
    20.35373177134412
  ]
}
```

Containerizing deployments is an excellent way to create portable data that can be tried by others. By sharing a container, the friction of setting up the environment is greatly reduced while ensuring a repeatable sys-

tem to interact with. Now that you know how to create, run, debug, and deploy containers for ML, you can leverage this to start automating non-containerized environments to speed up production deployments and enhance the whole process's robustness. Aside from containers, there is a push to get services closer to the user, and that is what I will get into next with edge devices and deployments.

Edge Devices

The computational cost of (fast) inferencing was astronomical a few years ago. Some of the more advanced capabilities of machine learning that are available today were cost-prohibitive not long ago. Not only have costs gone down, but more powerful chips are getting produced. Some of these chips are explicitly tailored for ML tasks. The right combination of needed features for these chips allows inferencing in devices like mobile phones: fast, small, and made for ML tasks. When "deploying to the edge" is mentioned in technology, it refers to compute devices that are not within a data center along with thousands of other servers. Mobile phones, Raspberry PI, and smart home devices are some examples that fit the description of an "edge device." In the last few years, large telecommunication companies have been pushing toward edge computing. Most of these edge deployments want to get faster feedback to users instead of routing expensive compute requests to a remote data center.

The general idea is that the closer the computational resources are to the user, the faster the user experience will be. There is a fine line that divides what may land at the edge versus what should go all the way to the data center and back. But, as I've mentioned, specialized chips are getting smaller, faster, and more effective; it makes sense to predict that the future means more ML at the edge. And the edge, in this case, will mean more and more devices that we previously didn't think could handle ML tasks.

Most people living in countries with plenty of data centers hosting application data do not experience much lag at all. For those countries that do not, the problem is exacerbated. For example, Peru has several submarine cables that connect it to other countries in South America but no direct connectivity to the US. This means that if you are uploading a picture from Peru to a service that hosts its application in a data center in the US, it will take exponentially longer than a country like Panama with multiple cables going back to North America. This example of uploading a picture is trivial but gets even worse when computational operations like ML predictions are done on the sent data. This section explores a few different ways on how edge devices can help by performing fast inferencing done as close to the user as possible. If long distances are a problem, imagine what happens when there is no (or very limited) connectivity like a remote farm. If you need fast inferencing done in a remote location, the options are limited, and this is where the *deploying to the edge* has an advantage over any data center.

Remember, users don't care that much about the spoon: they are interested in having a seamless way to try the delicious soup.

Coral

The Coral Project is a platform that helps build local (on-device) inferencing that captures the essence of edge deployments: fast, close to the user, and offline. In this section, I'll cover the USB Accelerator, which is an edge device that supports all major operating systems and works well with TensorFlow Lite models. You can compile most TensorFlow Lite models to run on this edge TPU (Tensor Processing Unit). Some aspects of operationalization of ML mean being aware of device support, installation methods, and compatibility. Those three aspects are true about the Coral Edge TPU: it works on most operating systems, with TensorFlow Lite models as long as they can get compiled to run on the TPU.

If you are tasked to deploy a fast inferencing solution at the edge on a remote location, you must ensure that all the pieces necessary for such a deployment will work correctly. This core concept of DevOps is covered throughout this book: repeatable deployment methods that create reproducible environments are critical. To ensure that is the case, you must be aware of compatibility.

First, start by installing the TPU runtime. For my machine, this means downloading and unzipping a file to run the installer script:

```
$ curl -O https://dl.google.com/coral/edgetpu_api/edgetpu_runtime_20201204.zip
[...]
$ unzip edgetpu_runtime_20201204.zip
Archive:  edgetpu_runtime_20201204.zip
   creating: edgetpu_runtime/
  inflating: edgetpu_runtime/install.sh
[...]
$ cd edgetpu_runtime
$ sudo bash install.sh
Password:
[...]
Installing Edge TPU runtime library [/usr/local/lib]...
Installing Edge TPU runtime library symlink [/usr/local/lib]...
```

NOTE

These setup examples use a Macintosh computer, so some of the installation methods and dependencies will vary from other operating systems. Check the getting started guide if you need support for a different computer.

Now that the runtime dependencies are installed in the system, we are ready to try the edge TPU. The Coral team has a useful repository with Python3 code that helps run image classification with a single command. Create a directory to clone the contents of the repository to set up the workspace for image classification:

```
$ mkdir google-coral && cd google-coral
$ git clone https://github.com/google-coral/tflite --depth 1
[...]
Resolving deltas: 100% (4/4), done.
$ cd tflite/python/examples/classification
```

NOTE

The `git` command uses a `--depth 1` flag, which performs a shallow clone. A shallow clone is desirable when the complete contents of the repository are not needed. Since this example is using the latest changes of the repository, there is no need to perform a full clone that contains a complete repository history.

For this example, do not run the *install_requirements.sh* script. First, make sure you have Python3 available and installed in your system and use it to create a new virtual environment; make sure that after activation, the Python interpreter points to the virtual environment and not the system Python:

```
$ python3 -m venv venv
$ source venv/bin/activate
$ which python
~/google-coral/tflite/python/examples/classification/venv/bin/python
```

Now that the *virtualenv* is active, install the two library dependencies and the TensorFlow Lite runtime support:

```
$ pip install numpy Pillow
$ pip install https://github.com/google-coral/pycoral/releases/download/\
release-frogfish/tflite_runtime-2.5.0-cp38-cp38-macosx_10_15_x86_64.whl
```

Both *numpy* and *Pillow* are straightforward to get installed in most systems. The outlier is the very long link that follows. This link is crucial to have, and it has to match your platform and architecture. Without that library, it is not possible to interact with the Coral device. The Python installation guide for TensorFlow Lite is the right source to double-check what link you need to use for your platform.

Now that you have everything installed and ready to perform the image classification, run the *classify_image.py* script to get the help menu. Bringing back the help menu, in this case, is an excellent way to verify all dependencies were installed and that the script works correctly:

```
usage:
  classify_image.py [-h] -m MODEL -i INPUT [-l LABELS] [-k TOP_K] [-c COUNT]
classify_image.py:
  error: the following arguments are required: -m/--model, -i/--input
```

Since I didn't define any flags when I called the script, an error returned, mentioning that I do need to pass some flags. Before we start using the other flags, we need to retrieve a TensorFlow model to work with an image to test it.

The Coral AI site has a models section where you can browse some of the specialized pretrained models it has for doing some image classification. Find the *iNat insects* one that recognizes over a thousand different types of insects. Download both the *tflite* model and the labels.

For this example, download a sample image of a common fly. The original source of the image is on Pixabay, but it is also conveniently accessible in the GitHub repository for this book.

Create directories for the model, labels, and image. Place the required files respectively in their directories. Having this order is not required, but it is useful to start adding more classification models, labels, and images later to play around more with the TPU device.

This is how the directory structure should look now:

```
.
├── README.md
├── classify.py
├── classify_image.py
├── images
│   └── macro-1802322_640.jpg
├── install_requirements.sh
├── labels
│   └── inat_insect_labels.txt
└── models
    └── mobilenet_v2_1.0_224_inat_insect_quant_edgetpu.tflite

3 directories, 7 files
```

Finally, we can try classification operations using the Coral device. Make sure that the device is plugged in with the USB cable, otherwise you will get a long traceback (which unfortunately doesn't really explain what the problem is):

```
Traceback (most recent call last):
  File "classify_image.py", line 122, in <module>
    main()
  File "classify_image.py", line 99, in main
    interpreter = make_interpreter(args.model)
  File "classify_image.py", line 72, in make_interpreter
    tflite.load_delegate(EDGETPU_SHARED_LIB,
  File "~/lib/python3.8/site-packages/tflite_runtime/interpreter.py",
    line 154, in load_delegate
        raise ValueError('Failed to load delegate from {}\n{}'.format(
ValueError: Failed to load delegate from libedgetpu.1.dylib
```

That error means that the device is unplugged. Plug it in and run the classification command:

```
$ python3 classify_image.py \
 --model models/mobilenet_v2_1.0_224_inat_insect_quant_edgetpu.tflite \
 --labels labels/inat_insect_labels.txt \
 --input images/macro-1802322_640.jpg
----INFERENCE TIME----
Note: The first inference on Edge TPU is slow because it includes loading
the model into Edge TPU memory.
11.9ms
2.6ms
2.5ms
2.5ms
2.4ms
-------RESULTS--------
Lucilia sericata (Common Green Bottle Fly): 0.43359
```

The image was classified correctly, and the common fly was detected! Find some other insect pictures and rerun the command to check how the model performs with different inputs.

Azure Percept

When this book was being written, Microsoft announced the release of a platform and hardware called Azure Percept. Although I didn't have enough time to get hands-on practical examples on how to take advantage of its features, I feel it is worth mentioning some of its functionality.

The same concepts that apply to the Coral device in the previous section and to the edge, in general, apply to the devices for Percept: they allow seamless machine learning operations at the edge.

First, it is important to emphasize that although the Percept products are mostly advertised as pieces of hardware, Azure Percept is a whole platform for doing edge computing, from the devices themselves all the way to deployment, training, and management in Azure. There is also support for the major AI platforms like ONNX and TensorFlow, making it easier to try out with prebuilt models.

One downside of the Azure Percept hardware compared to the Coral devices is that it is much more expensive, making it harder to buy one of its bundles to try the new technology. As always, Microsoft has done a stellar job in documenting and adding a good amount of context and examples that are worth exploring if you are interested.

TFHub

A great resource to find TensorFlow models is the TensorFlow Hub. The hub is a repository of thousands of pretrained models ready to be used. For the Coral Edge TPU, not all models will work, though. Since the TPU has separate instructions specific to the device, a model needs to be explicitly compiled for it.

Now that you can run classifications with the Coral USB device, you can use TFHub to find other pretrained models to work with. At the hub, a Coral model format is available; click on it to get to the models ready to use for the TPU as shown in Figure 3-2.

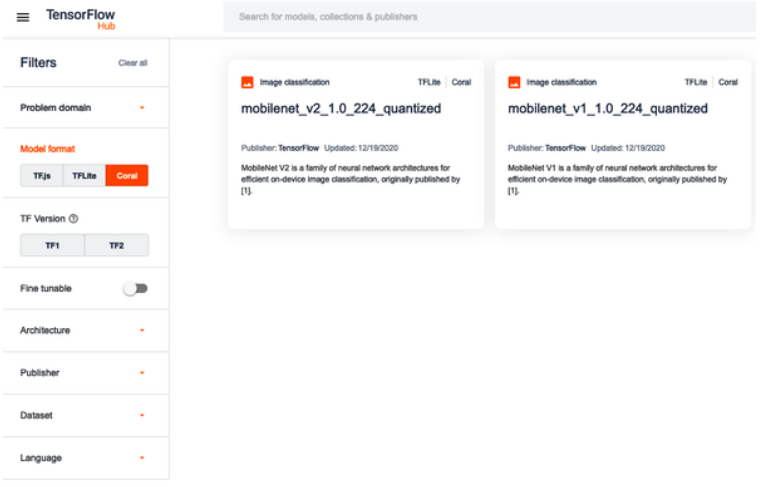

Figure 3-2. TFHub Coral models

Select the *MobileNet Quantized V2* model for download. This model can detect over a thousand objects from images. The previous examples using Coral require the labels and the model, so make sure you download that as well.

NOTE

When these models are presented in the TFHub website, multiple different formats are available. Ensure you double-check which model format you are getting, and that (in this case) it is compatible with the Coral device.

Porting Over Non-TPU Models

You might find that the model you need is available in some situations but not compiled for the TPU device you have. The Coral Edge TPU does have a compiler available, but it isn't installable in every platform as the runtime dependencies are. When such situations come up, you have to get creative on the solutions and always attempt to find the automation within any workarounds possible. The compiler documentation requires a Debian or Ubuntu Linux distribution, and the instructions on how to get everything set up for the compiler are tied to that particular distro.

In my case, I'm working from an Apple computer, and I don't have other computers running Linux. What I *do have* is a container runtime installed locally in which I can run any image from any distro with a few commands. We've already covered how to get started with containers, how to run them, and how to create them. And this is the perfect use case for creating a new Debian-based container with everything installed for the compiler to solve this problem.

Now that we understand the problem and have a solution in mind with containers, create a new *Dockerfile* to build a container image for the compiler:

```
FROM debian:stable

RUN apt-get update && apt install -yq curl build-essential gnupg

RUN curl https://packages.cloud.google.com/apt/doc/apt-key.gpg | \
    apt-key add -

RUN \
 echo "deb https://packages.cloud.google.com/apt coral-edgetpu-stable main" | \
 tee /etc/apt/sources.list.d/coral-edgetpu.list

RUN apt-get update && apt-get install -yq edgetpu-compiler

CMD ["/bin/bash"]
```

With the newly created *Dockerfile*, create a new image to run the compiler:

```
$ docker build -t tpu-compiler .
[+] Building 15.5s (10/10) FINISHED
 => => transferring dockerfile: 408B
[...]
 => [5/5] RUN apt update && apt install -yq edgetpu-compiler
 => exporting to image
 => => exporting layers
 => => writing image
    sha256:08078f8d7f7dd9002bd5a1377f24ad0d9dbf8f7b45c961232cf2cbf8f9f946e4
 => => naming to docker.io/library/tpu-compiler
```

I've identified a model that I want to use with the TPU compiler but doesn't come compiled for it.

NOTE

Only models that are precompiled for TensorFlow Lite and are quantized will work with the compiler. Ensure that models are both *tflite* and *quantized* before downloading them to convert them with the compiler.

Download the model locally. In this case, I use the command line to save it in the current working directory:

```
$ wget -O mobilenet_v1_50_160_quantized.tflite \
 https://tfhub.dev/tensorflow/lite-model/\
 mobilenet_v1_0.50_160_quantized/1/default/1?lite-format=tflite

Resolving tfhub.dev (tfhub.dev)... 108.177.122.101, 108.177.122.113, ...
Connecting to tfhub.dev (tfhub.dev)|108.177.122.101|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1364512 (1.3M) [application/octet-stream]
Saving to: ‘mobilenet_v1_50_160_quantized.tflite’

(31.8 MB/s) - ‘mobilenet_v1_50_160_quantized.tflite’ saved [1364512/1364512]

$ ls
mobilenet_v1_50_160_quantized.tflite
```

Although I've used the command line, you can also download the model by going to the model on the website. Ensure you move the file to the current working directory for the next steps.

We need to get the downloaded model into the container and then copy back the files locally. Docker makes this task somewhat more manageable by using a *bind mount*. This mount operation will link a path from my machine into the container, effectively sharing anything I have into the container. This also works great for files created in the container, and I need them back on the local machine. Those files created in the container will appear automatically in my local environment.

Start the container with the bind mount:

```
$ docker run -it -v ${PWD}:/models tpu-compiler
root@5125dcd1da4b:/# cd models
root@5125dcd1da4b:/models# ls
mobilenet_v1_50_160_quantized.tflite
```

There are a couple of things happening with the previous command. First, I'm using `PWD` to indicate that the current working directory, where the *mobilenet_v1_50_160_quantized.tflite* file exists, is what I want in the container. The destination path within the container is */models*. And lastly, I'm using the container built with the tag `tpu-compiler` to specify the container I need. If you used a different tag when building the image, you would need to update that part of the command. After starting the container, I change directories into */models*, list the directory contents, and find the downloaded model in my local machine. The environment is now ready to use the compiler.

Verify the compiler works by calling its help menu:

```
$ edgetpu_compiler --help
Edge TPU Compiler version 15.0.340273435

Usage:
edgetpu_compiler [options] model...
```

Next, run the compiler against the quantized model:

```
$ edgetpu_compiler mobilenet_v1_50_160_quantized.tflite
Edge TPU Compiler version 15.0.340273435

Model compiled successfully in 787 ms.

Input model: mobilenet_v1_50_160_quantized.tflite
Input size: 1.30MiB
Output model: mobilenet_v1_50_160_quantized_edgetpu.tflite
Output size: 1.54MiB
Number of Edge TPU subgraphs: 1
Total number of operations: 31
Operation log: mobilenet_v1_50_160_quantized_edgetpu.log
See the operation log file for individual operation details.
```

The operation took less than a second to run and produced a few files, including the newly compiled model (*mobilenet_v1_50_160_quantized_edgetpu.tflite*) that you can now use with the edge device.

Finally, exit out of the container, go back to the local machine, and list the contents of the directory:

```
$ ls
mobilenet_v1_50_160_quantized.tflite
mobilenet_v1_50_160_quantized_edgetpu.log
mobilenet_v1_50_160_quantized_edgetpu.tflite
```

This is a handy workaround to requiring an operating system for a tool. Now that this container can compile models for the edge device, it can be automated further by porting over models you need with a few lines in a script. Remember that there are a few assumptions made in the process and that you must ensure these assumptions are all accurate at compile time. Otherwise, you will get errors from the compiler. This process is an example of trying to use a nonquantized model with the compiler:

```
$ edgetpu_compiler vision_classifier_fungi_mobile_v1.tflite
Edge TPU Compiler version 15.0.340273435
Invalid model: vision_classifier_fungi_mobile_v1.tflite
Model not quantized
```

Containers for Managed ML Systems

At the heart of advanced next-generation MLOps workflows are managed ML systems like AWS SageMaker, Azure ML Studio, and Google's Vertex AI. All of these systems build on top of containers. Containers are a secret ingredient for MLOps. Without containerization, it is much more chal-

lenging to develop and use technologies like AWS SageMaker. In Figure 3-3, notice that the EC2 Container Registry is the location where the inference code image and the training code live.

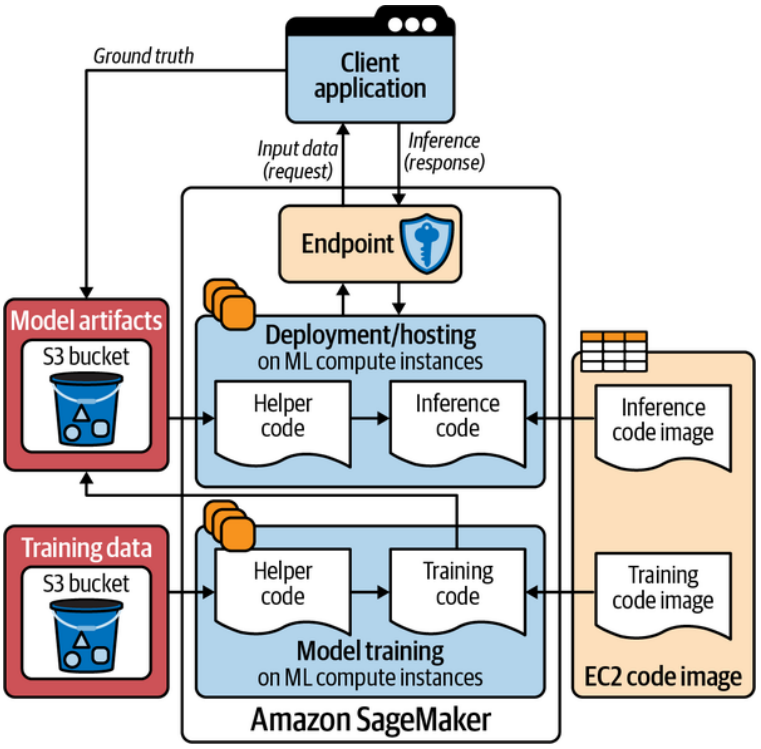

Figure 3-3. SageMaker containers

This process is critically important because it allows DevOps best practices to bake into creating these images—among these most importantly are continuous integrations and continuous delivery. Containers increase the entire ML architecture quality by reducing complexity since the images are already "baked." Intellectual horsepower can shift to other problems like data drift, analyzing the feature store for suitable candidates for a newer model, or evaluating whether the new model solves customer needs.

Containers in Monetizing MLOps

Monetizing MLOps is another crucial problem for both startups and large companies. Containers play a role yet again! In the case of SageMaker, use either an algorithm or a model sold in the AWS Marketplace, as shown in Figure 3-4. They are the mode of delivery for the product sold.

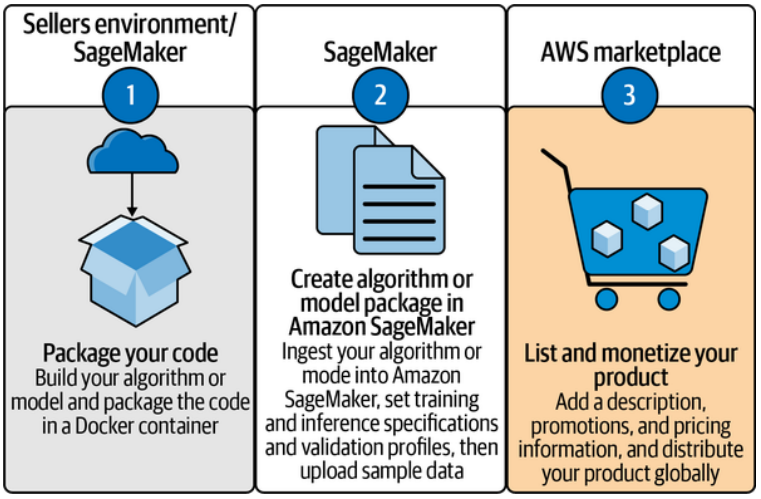

Figure 3-4. SageMaker seller workflow

The advantage of a container as a product is that it is sold much like other products sold in a physical store, such as peanut butter, flour, or milk. In the scenario where a company decides to produce high-quality, organic peanut butter, it may want to focus strictly on making the peanut butter, not building out a network of stores to sell the peanut butter.

Likewise, in companies looking to monetize machine learning, the container is an ideal package for delivering both models and algorithms to customers. Next, let's take a look at how you can build once and run many with containers.

Build Once, Run Many MLOps Workflow

Ultimately a container process for MLOps culminates in many rich options for both product and engineering. In Figure 3-5, you see that a container is an ideal package to monetize intellectual property from a product perspective. Likewise, from an engineering perspective, a container can serve out predictions, do training, or deploy to an edge device like a Coral TPU or Apple iPhone.

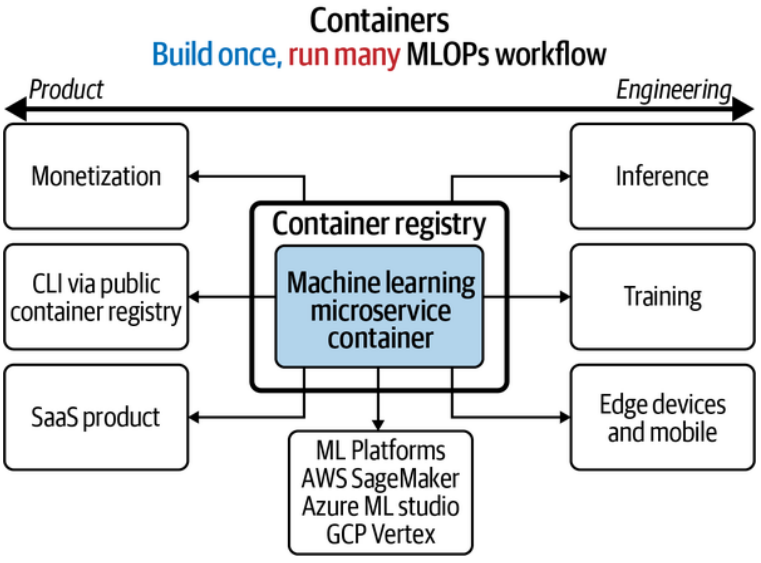

Figure 3-5. Build Once, Run Many MLOps container

MLOps and container technology are complementary in that containers help you deliver business value. MLOps methodologies then build directly on top of this technology to streamline productivity and add value. Next,

let's wrap up the chapter and summarize the essential aspects of containers for MLOps.

Conclusion

When operationalizing ML models, you will often encounter many different possibilities for deployment. It is becoming fairly common to see models getting deployed on mobile phones and other (small) devices that you can plug into any computer with a USB port. The problems that edge inferencing provides (like offline, remote, and fast access) can be transformational, specifically for remote regions without access to a reliable source of power and network. Similar to edge devices, containerization enables faster and more reliable reproduction of environments. Reproducible machine environments were a challenging problem to solve just a few years ago. Containerization is exceptionally relevant in that case. Fast scaling of resources and transitioning deployment environments from cloud providers, or even moving workloads from on-premise (local) to the cloud is far easier to accomplish with containers.

With that covered, our next chapter dives into the continuous delivery process for machine learning models.

Exercises

- Recompile a model to work with the Coral Edge TPU from TFHub.
- Use the MobileNet V2 model to perform inference on other objects and get accurate results.
- Create a new container image, based on the Flask example, that serves a model and that provides examples on a `GET` request to interact with the model. Create another endpoint that provides useful metadata about the model.
- Publish the newly created image to a container registry like Docker Hub.

Critical Thinking Discussion Questions

- Would it be possible to use a container to perform online predictions with an edge TPU device like Coral? How? or Why not?
- What is a container runtime, and how does it relate to Docker?
- Name three good practices when creating a Dockerfile.
- What are two critical concepts of DevOps mentioned in this chapter? Why are they useful?
- Create a definition, in your own words, of what the "edge" is. Give some ML examples that can be applied.