Appendix A. Introduction to Infrastructure for Machine Learning

This appendix gives a brief introduction to some of the most useful infrastructure tools for machine learning: containers, in the form of Docker or Kubernetes. While this may be the point at which you hand your pipeline over to a software engineering team, it's useful for anyone building machine learning pipelines to have an awareness of these tools.

What Is a Container?

All Linux operating systems are based on the filesystem, or the directory structure that includes all hard drives and partitions. From the root of this filesystem (denoted as `/`), you can access almost all aspects of a Linux system. Containers create a new, smaller root and use it as a "smaller Linux" within a bigger host. This lets you have a whole separate set of libraries dedicated to a particular container. On top of that, containers let you control resources like CPU time or memory for each container.

Docker is a user-friendly API that manages containers. Containers can be built, packaged, saved, and deployed multiple times using Docker. It also allows developers to build containers locally and then publish them to a central registry that others can pull from and immediately run the container.

Dependency management is a big issue in machine learning and data science. Whether you are writing in R or Python, you're almost always dependent on third-party modules. These modules are updated frequently and may cause breaking changes to your pipeline when versions conflict. By using containers, you can prepackage your data processing code along with the correct module versions and avoid these problems.

Introduction to Docker

To install Docker on Mac or Windows, visit *https://docs.docker.com/install* and download the latest stable version of Docker Desktop for your operating system.

For a Linux operating system, Docker provides a very convenient script to install Docker with just a couple of commands:

```
$ curl -fsSL https://get.docker.com -o get-docker.sh
$ sudo sh get-docker.sh
```

You can test whether your Docker installation is working correctly using the command:

```
docker run hello-world
```

Introduction to Docker Images

A Docker image is the basis of a container, and it consists of a collection of changes to the root filesystem and the execution parameters to run the container. The image must first be "built" before it can be run.

A useful concept behind Docker images is storage layers. Building an image means installing almost a whole dedicated Linux OS for your package. To avoid running this operation every time, Docker uses a layered filesystem. Here is how it works: if the first layer contains Files A and B, and the second layer adds File C, the resulting filesystems show A, B, and C. If we want to create a second image that uses Files A, B, and D, we only need to change the second layer to add File D. This means that we can have base images that have all the basic packages, and then we can focus on changes specific to your image, as shown in Figure A-1.

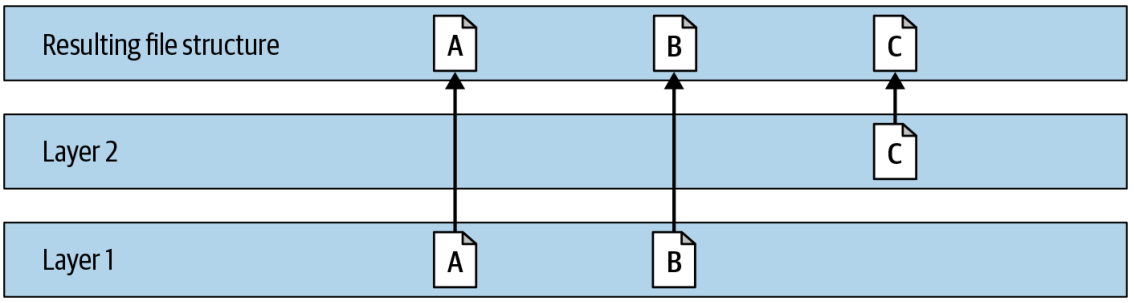

Figure A-1. Example of a layered filesystem

Docker image names are called *tags*. They follow the pattern `docker registry/docker namespace/image name:tag`. For example, *docker.io/tensorflow/tensorflow:nightly* would point to the `tensorflow` image in DockerHub in the `tensorflow` namespace. The tag is usually used to mark versions of a particular image. In our example, the tag `nightly` is reserved for nightly builds of TensorFlow.

Docker images are built based on a `Dockerfile`. Each line in a *Dockerfile* starts with one of a few clauses. The most important are:

FROM

Indicates the Docker base container to build from. We will always want to use this clause. There are many base containers available to download, such as `ubuntu`.

RUN

Runs bash. This is the bread and butter of most Docker images. This is where we want to do package installations, directory creation, etc. Because each line will create a layer in the image, it's good to have package installations and other long tasks as one of the first lines in *Dockerfile*. This means that during rebuilds, Docker will try to use layers from the cache.

ARG

Builds arguments. It's useful if you want to have multiple flavors of the same image, for example *dev* and *production*.

COPY

Copies files from a context. The path to the context is an argument used in `docker build`. The context is a set of local files that are exposed to Docker during the build, and it only uses them in the process. This can be used to copy your source code to a container.

ENV

Sets an environment variable. This variable will be part of the image and will be visible in build and run.

CMD

This is the default command for a container. A good practice in Docker is to run one command per container. Docker will then monitor this command, exit when it exits, and post STDOUT from it to `docker logs`. Another way to specify this command is by using `ENTRYPOINT`. There are several subtle differences between these, but here we'll focus on `CMD`.

USER

The default user in the container. This is different from host system users. You should create a user during the build if you want to run commands as one.

WORKDIR

The default directory in the image. This will be the directory the default command will be run from.

EXPOSE

Specifies ports the container will use. For example, HTTP services should have `EXPOSE 80`.

Building Your First Docker Image

Let's build our first image!

First, we need to create a new directory for our small Docker project:

```
$ mkdir hello-docker
$ cd hello-docker
```

In this directory, create a file called *Dockerfile* with the following contents:

```
FROM ubuntu
RUN apt-get update
RUN apt-get -y install cowsay
CMD /usr/games/cowsay "Hello Docker"
```

To build it, use the command `docker build . -t hello-docker`. The `-t` flag specifies the tag for this image. You will see a series of commands that are run in the container. Each layer in our image (corresponding to each command in the `Dockerfile`) is called in the temporary container running the previous layers. The difference is saved, and we end up with a full image. The first layer (which we don't build) is based on Ubuntu Linux. The `FROM` command in `Dockerfile` tells Docker to pull this image from a registry, in our case DockerHub, and use it as the base image.

After the build is finished, calling `docker images` should show something like this:

```
REPOSITORY      TAG         IMAGE ID          CREATED           SIZE
hello-docker    latest      af856e494ed4      2 minutes ago     155MB
ubuntu          latest      94e814e2efa8      5 weeks ago       88.9MB
```

We should see the base Ubuntu image and our new image.

Even though we have built this image, that doesn't mean it's ready to use. The next step is to run the image. `docker run` is arguably the most important command in Docker. It creates a new container from an existing image (or, if an image is not present on the system, it will try to pull it from the registry). To run our image, we should call `docker run -it hello-docker`. This should show us the output of our `cowsay` command.

DOCKER REGISTRIES

One of the great strengths of Docker is the ease of publishing a built image. The repository of Docker images is called the *registry*. The default Docker registry, called DockerHub, is supported by Docker, Inc. An account on DockerHub is free and lets you push public images to it.

Diving into the Docker CLI

The Docker CLI is the main way to interact with images and containers on your local machine. In this section, we'll discuss the most important commands and options for it. Let's start with `docker run`.

There are many important options we could pass to `docker run`. With these, we can override most of the options set in the `Dockerfile`. This is important because many Docker images will have a set basic default command, but often that's not exactly how we want to run them. Let's look at our `cowsay` example:

```
docker run -it hello-docker /usr/games/cowsay "Our own message"
```

The argument that comes after the image tag will override the default command we have set in `Dockerfile`. This is the best way to specify our own command-line flags to the default binaries. Other useful flags for `docker run` include:

`-it`

Means "interactive" (i) and `tty` (t), which allows us to interact with the command being run from our shell.

`-v`

Mounts a Docker volume or host directory into the container—for example, a directory containing datasets.

`-e`

Passes configurations through environment variables. For example, `docker run -e MYVARNAME=value image` will create the `MYVARNAME` env variable in a container.

`-d`

Allows the container to be run in detached mode, making it perfect for long running tasks.

`-p`

Forwards a host's port to a container to allow external services to interact with the container over a network. For example, `docker run -d -p 8080:8080 imagename` would forward `localhost:8080` to the container's port 8080.

DOCKER COMPOSE

`docker run` can get pretty complex when you start mounting directories, managing container links, and so on. *Docker Compose* is a project to help with that. It allows you to create a *docker-compose.yaml* file in which you can specify all your Docker options for any number of containers. You can then link containers together over a network or mount the same directories.

Other useful Docker commands include:

`docker ps`

Shows all the running containers. To also show exited containers, add the `-a` flag.

`docker images`

Lists all images present on the machine.

`docker inspect container id`

Allows us to examine the container's configuration in detail.

`docker rm`

Deletes containers.

`docker rmi`

Deletes images.

`docker logs`

Displays the STDOUT and STDERR information produced by a container, which is very useful for debugging.

`docker exec`

Allows you to call a command within a running container. For example, `docker exec -it` *`container id`* `bash` will allow you to enter into the container environment with bash and examine it from inside. The `-it` flag works in the same way as in `docker run`.

Introduction to Kubernetes

Up to now, we've just talked about Docker containers running on a single machine. What happens if you want to scale up? Kubernetes is an open source project, initially developed by Google, that manages scheduling and scaling for your infrastructure. It dynamically scales loads to many servers and keeps track of computing resources. Kubernetes also maximizes efficiency by putting multiple containers on one machine (depending on their size and needs), and it manages the communications between containers. It can run on any cloud platform—AWS, Azure, or GCP.

Some Kubernetes Definitions

One of the hardest parts of getting started with Kubernetes is the terminology. Here are a few definitions to help you:

Cluster

A cluster is a set of machines that contains a central node controlling the Kubernetes API server and many worker nodes.

Node

A node is a single machine (either a physical machine or a virtual machine) within a cluster.

Pod

A pod is a group of containers that run together on the same node. Often, a pod only contains a single container.

Kubelet

A kubelet is the Kubernetes agent that manages communication with the central node on each worker node.

Service

A service is a group of pods and the policies to access them.

Volume

A volume is a storage space shared by all containers in the same pod.

Namespace

A namespace is the virtual cluster that divides up the space in a physical cluster into different environments. For example, we can divide a cluster into development and production environments or environments for different teams.

ConfigMap

A ConfigMap provides an API for storing nonconfidential configuration information (environment variables, arguments, etc.) in Kubernetes. ConfigMaps are useful to separate the configuration from container images.

kubectl

kubectl is the CLI for Kubernetes.

Getting Started with Minikube and kubectl

We can create a simple, local Kubernetes cluser using a tool called Minikube. Minikube makes it easy to set up Kubernetes on any operating system. It creates a virtual machine, installs Docker and Kubernetes on it, and adds a local user connected to it.

DON'T USE MINIKUBE IN PRODUCTION

Minikube should not be used in production; rather, it is designed to be a quick and easy local environment. The easiest way to gain access to a production-quality Kubernetes cluer is by purchasing managed Kubernetes as a service from any major public cloud provider.

First, install kubectl, the Kubernetes CLI tool.

For Mac, kubectl can be installed using `brew`:

```
brew install kubectl
```

For Windows, see their resources.

For Linux:

```
curl -LO https://storage.googleapis.com/kubernetes-release\
/release/v1.14.0/bin/linux/amd64/kubectl
chmod +x ./kubectl
sudo mv ./kubectl /usr/local/bin/kubectl
```

To install Minikube, we'll first need to install a *hypervisor* that creates and runs virtual machines, such as VirtualBox.

On a Mac, Minikube can be installed using `brew`:

```
brew install minikube
```

For Windows, see the resources.

For Linux machines, use the following steps:

```
curl -Lo minikube \
https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
chmod +x minikube
sudo cp minikube /usr/local/bin && rm minikube
```

Once installation is complete, start a simple Kubernetes cluster in a single command:

```
minikube start
```

To quickly check if Minikube is good to go, we can try to list the nodes in the cluster:

```
kubectl get nodes
```

Interacting with the Kubernetes CLI

The Kubernetes API is based on *resources*. Almost everything in the Kubernetes world is represented as a resource. `kubectl` is built with this in mind, so it will follow a similar pattern for most of the resource interactions.

For example, a typical `kubectl` call to list all pods would be:

```
kubectl get pods
```

This should produce a list of all the running pods, but since we haven't created any, the listing will be empty. That doesn't mean that no pods are currently running on our cluster. Most of the resources in Kubernetes can be placed in a namespace, and unless you query the specific namespace, they won't show up. Kubernetes runs its internal services in a namespace called `kube-system`. To list all pods in any namespace, you can use the `-n` option:

```
kubectl get pods -n kube-system
```

This should return several results. We can also use `--all-namespaces` to show all pods regardless of namespace.

You can use the name only to display one pod:

```
kubectl get po mypod
```

You can also filter out by label. For example, this call should show all pods that have the label `component` with the value `etcd` in `kube-system`:

```
kubectl get po -n kube-system -l component=etcd
```

The information displayed by `get` can also be modified. For example:

```
# Show nodes and addresses of pods.
kubectl get po -n kube-system -o wide
# Show the yaml definition of pod mypod.
kubectl get po mypod -o yaml
```

To create a new resource, `kubectl` offers two commands: `create` and `apply`. The difference is that `create` will always try to create a new resource (and fail if it already exists), whereas `apply` will either create or update an existing resource.

The most common way to create a new resource is by using a YAML (or JSON) file with the resource definition, as we'll see in the next section. The following kubectl commands allow us to create of update Kubernetes resources, (e.g., pods):

```
# Create a pod that is defined in pod.yaml.
kubectl create -f pod.yaml
# This can also be used with HTTP.
kubectl create -f http://url-to-pod-yaml
```

```
# Apply will allow making changes to resources.
kubectl apply -f pod.yaml
```

To delete a resource, use `kubectl delete`:

```
# Delete pod foo.
kubectl delete pod foo
# Delete all resources defined in pods.yaml.
kubectl delete -f pods.yaml
```

You can use `kubectl edit` to update an existing resource quickly. This will open an editor where you can edit the loaded resource definition:

```
kubectl edit pod foo
```

Defining a Kubernetes Resource

Kubernetes resources are most often defined as YAML (although JSON can also be used). Basically, all resources are data structures with a few essential sections.

`apiVersion`

: Every resource is part of an API, either supplied by Kubernetes itself or by third parties. The version number shows the maturity of the API.

`kind`

: The type of resource (e.g., pod, volume, etc.).

`metadata`

: The data required for any resource.

`name`

: The key that every resource can be queried by, which must be unique.

labels

Each resource can have any number of key-value pairs called labels. These labels can then be used in selectors, for querying resources, or just as information.

annotations

Secondary key-value pairs that are purely informational and cannot be used in queries or selectors.

namespace

A label that shows a resource belongs to a particular namespace or team.

spec

Configuration of the resource. All information required for the actual runtime should be in `spec`. Each `spec` schema is unique to a particular resource type.

Here is an example *.yaml* file using these definitions:

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

In this file, we have `apiVersion` and `kind`, which define what this resource is. We have `metadata` that specifies the name and the label, and we have `spec`, which makes up the body of the resource. Our pod consists of a single container, running the command `sh -c echo Hello Kubernetes! && sleep 3600` in the image `busybox`.

Deploying Applications to Kubernetes

In this section, we will walk through the full deployment of a functional Jupyter Notebook using Minikube. We will create a persistent volume for our notebooks and create a NodePort service to allow us access to our notebook.

First, we need to find the correct Docker image. *jupyter/tensorflow-notebook* is an official image maintained by the Jupyter community. Next, we will need to find out which port our application will listen on: in this case, it's `8888` (the default port for Jupyter Notebooks).

We want our notebook to persist between sessions, so we need to use PVC (persistent volume claim). We create a *pvc.yaml* file to do this for us:

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: notebooks
spec:
  accessModes:
    – ReadWriteOnce
  resources:
    requests:
      storage: 3Gi
```

Now we can create this resource by calling:

```
kubectl apply -f pvc.yaml
```

This should create a volume. To confirm, we can list all volumes and PVCs:

```
kubectl get pv
kubectl get pvc
kubectl describe pvc notebooks
```

Next up, we create our deployment *.yaml* file. We will have one pod that will mount our volume and expose port 8888:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: jupyter
  labels:
    app: jupyter
spec:
  selector:
    matchLabels:
      app: jupyter  ❶
  template:
    metadata:
      labels:
        app: jupyter
    spec:
      containers:
        - image: jupyter/tensorflow-notebook  ❷
          name: jupyter
          ports:
          - containerPort: 8888
            name: jupyter
          volumeMounts:
          - name: notebooks
            mountPath: /home/jovyan
      volumes:
      - name: notebooks
        persistentVolumeClaim:
          claimName: notebooks
```

❶ It's important that this selector matches the labels in the template.

❷ Our image.

By applying this resource (in the same way we did with PVC), we will create a pod with a Jupyter instance:

```
# Let's see if our deployment is ready.
kubectl get deploy
# List pods that belong to this app.
kubectl get po -l app=jupyter
```

When our pod is in the `Running` state, we should grab a token with which we'll be able to connect to our notebook. This token will appear in logs:

```
kubectl logs deploy/jupyter
```

To confirm that the pod is working, let's access our notebook with `port-forward`:

```
# First we need the name of our pod; it will have a randomized suffix.
kubectl get po -l app=jupyter
kubectl port-forward jupyter-84fd79f5f8-kb7dv 8888:8888
```

With this, we should be able to access a notebook on *http://localhost:8888*. The problem is, nobody else will be able to since it's proxied through our local `kubectl`. Let's create a `NodePort` service to let us access the notebook:

```
apiVersion: v1
kind: Service
metadata:
  name: jupyter-service
  labels:
    app: jupyter
spec:
  ports:
    - port: 8888
      nodePort: 30888
  selector:
    app: jupyter
  type: NodePort
```

When this is created, we should be able to access our Jupyter! But first, we need to find the IP address of our pod. We should be able to access Jupyter under this address and port 30888:

```
minikube ip
# This will show us what is our kubelet address is.
192.168.99.100:30888
```

Others can now access the Jupyter Notebook by using the obtained IP address and the service port (see Figure A-2). Once you access the address with your browser, you should see the Jupyter Notebook instance.

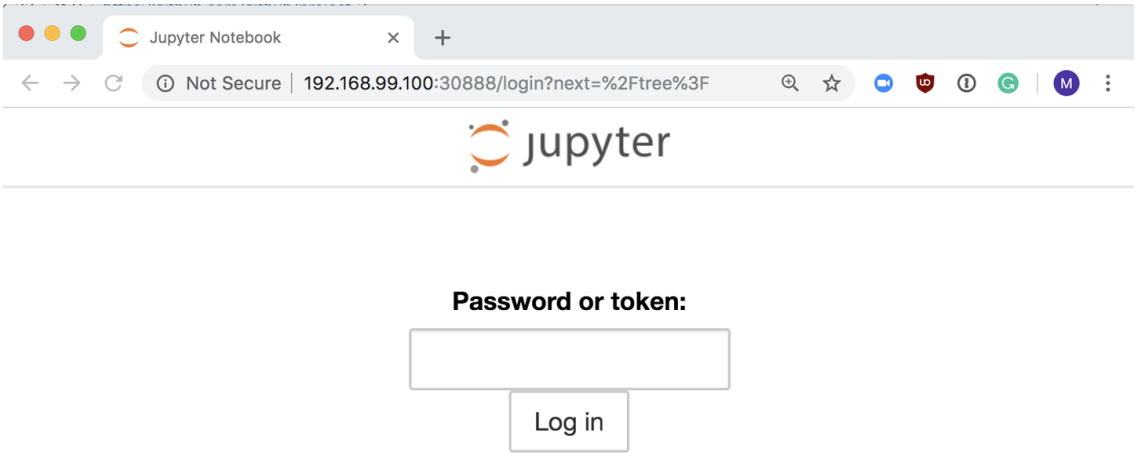

Figure A-2. Jupyter Notebook running on Kubernetes

This was a brief overview of Kubernetes and its parts. The Kubernetes ecosystem is very extensive, and a brief appendix can't provide a holistic overview. For more details regarding Kubeflow's underlying architecture Kubernetes, we highly recommend the O'Reilly publication *Kubernetes: Up and Running* by Brendan Burns et al.