Chapter 19. Training and Deploying TensorFlow Models at Scale

Once you have a beautiful model that makes amazing predictions, what do you do with it? Well, you need to put it in production! This could be as simple as running the model on a batch of data, and perhaps writing a script that runs this model every night. However, it is often much more involved. Various parts of your infrastructure may need to use this model on live data, in which case you will probably want to wrap your model in a web service: this way, any part of your infrastructure can query the model at any time using a simple REST API (or some other protocol), as we discussed in Chapter 2. But as time passes, you'll need to regularly retrain your model on fresh data and push the updated version to production. You must handle model versioning, gracefully transition from one model to the next, possibly roll back to the previous model in case of problems, and perhaps run multiple different models in parallel to perform *A/B experiments*.[1] If your product becomes successful, your service may start to get a large number of of queries per second (QPS), and it must scale up to support the load. A great solution to scale up your service, as you will see in this chapter, is to use TF Serving, either on your own hardware infrastructure or via a cloud service such as Google Vertex AI.[2] It will take care of efficiently serving your model, handle graceful model transitions, and more. If you use the cloud platform you will also get many extra features, such as powerful monitoring tools.

Moreover, if you have a lot of training data and compute-intensive models, then training time may be prohibitively long. If your product needs to adapt to changes quickly, then a long training time can be a showstopper (e.g., think of a news recommendation system promoting news from last week). Perhaps even more importantly, a long training time will prevent you from experimenting with new ideas. In machine learning (as in many other fields), it is hard to know in advance which ideas will work, so you should try out as many as possible, as fast as possible. One way to speed up training is to use hardware accelerators such as GPUs or TPUs. To go even faster, you can train a model across multiple machines, each equipped with multiple hardware accelerators. TensorFlow's simple yet powerful distribution strategies API makes this easy, as you will see.

In this chapter we will look at how to deploy models, first using TF Serving, then using Vertex AI. We will also take a quick look at deploying models to mobile apps, embedded devices, and web apps. Then we will discuss how to speed up computations using GPUs and how to train models across multiple devices and servers using the distribution strategies API. Lastly, we will explore how to train models and fine-tune their hyperparameters at scale using Vertex AI. That's a lot of topics to discuss, so let's dive in!

Serving a TensorFlow Model

Once you have trained a TensorFlow model, you can easily use it in any Python code: if it's a Keras model, just call its `predict()` method! But as your infrastructure grows, there comes a point where it is preferable to wrap your model in a small service whose sole role is to make predictions and have the rest of the infrastructure query it (e.g., via a REST or gRPC API).[3] This decouples your model from the rest of the infrastructure, making it possible to easily switch model versions or scale the service up as needed (independently from the rest of your infrastructure), perform A/B experiments, and ensure that all your software components rely on the same model versions. It also simplifies testing and development, and more. You could create your own microservice using any technology you want (e.g., using the Flask library), but why reinvent the wheel when you can just use TF Serving?

Using TensorFlow Serving

TF Serving is a very efficient, battle-tested model server, written in C++. It can sustain a high load, serve multiple versions of your models and watch a model repository to automatically deploy the latest versions, and more (see Figure 19-1).

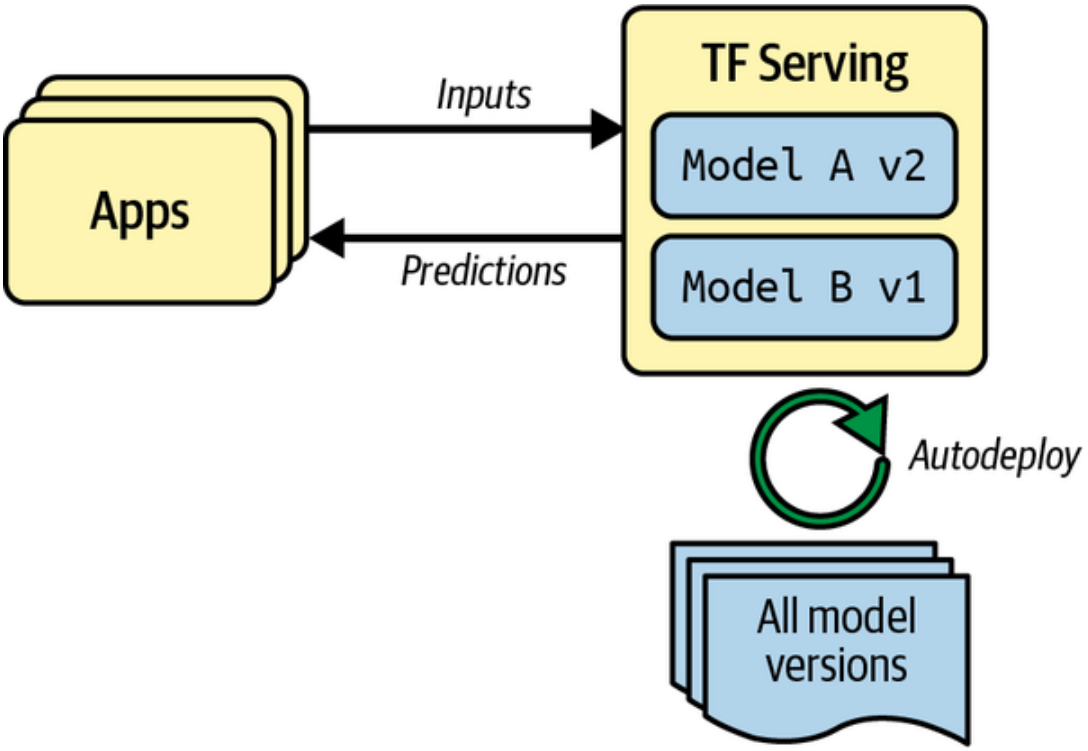

Figure 19-1. TF Serving can serve multiple models and automatically deploy the latest version of each model

So let's suppose you have trained an MNIST model using Keras, and you want to deploy it to TF Serving. The first thing you have to do is export this model to the SavedModel format, introduced in Chapter 10.

Exporting SavedModels

You already know how to save the model: just call `model.save()`. Now to version the model, you just need to create a subdirectory for each model version. Easy!

```
from pathlib import Path
import tensorflow as tf

X_train, X_valid, X_test = [...]  # load and split the MNIST dataset
model = [...]  # build & train an MNIST model (also handles image preprocessing

model_name = "my_mnist_model"
model_version = "0001"
model_path = Path(model_name) / model_version
model.save(model_path, save_format="tf")
```

It's usually a good idea to include all the preprocessing layers in the final model you export so that it can ingest data in its natural form once it is deployed to production. This avoids having to take care of preprocessing separately within the application that uses the model. Bundling the preprocessing steps within the model also makes it simpler to update them

later on and limits the risk of mismatch between a model and the preprocessing steps it requires.

WARNING

Since a SavedModel saves the computation graph, it can only be used with models that are based exclusively on TensorFlow operations, excluding the `tf.py_function()` operation, which wraps arbitrary Python code.

TensorFlow comes with a small `saved_model_cli` command-line interface to inspect SavedModels. Let use it to inspect our exported model:

```
$ saved_model_cli show --dir my_mnist_model/0001
The given SavedModel contains the following tag-sets:
'serve'
```

What does this output mean? Well, a SavedModel contains one or more *metagraphs*. A metagraph is a computation graph plus some function signature definitions, including their input and output names, types, and shapes. Each metagraph is identified by a set of tags. For example, you may want to have a metagraph containing the full computation graph, including the training operations: you would typically tag this one as `"train"`. And you might have another metagraph containing a pruned computation graph with only the prediction operations, including some GPU-specific operations: this one might be tagged as `"serve", "gpu"`. You might want to have other metagraphs as well. This can be done using TensorFlow's low-level SavedModel API. However, when you save a Keras model using its `save()` method, it saves a single metagraph tagged as `"serve"`. Let's inspect this `"serve"` tag set:

```
$ saved_model_cli show --dir 0001/my_mnist_model --tag_set serve
The given SavedModel MetaGraphDef contains SignatureDefs with these keys:
SignatureDef key: "__saved_model_init_op"
SignatureDef key: "serving_default"
```

This metagraph contains two signature definitions: an initialization function called `"__saved_model_init_op"`, which you do not need to worry about, and a default serving function called `"serving_default"`. When saving a Keras model, the default serving function is the model's `call()`

method, which makes predictions, as you already know. Let's get more details about this serving function:

```
$ saved_model_cli show --dir 0001/my_mnist_model --tag_set serve \
                       --signature_def serving_default
The given SavedModel SignatureDef contains the following input(s):
  inputs['flatten_input'] tensor_info:
      dtype: DT_UINT8
      shape: (-1, 28, 28)
      name: serving_default_flatten_input:0
The given SavedModel SignatureDef contains the following output(s):
  outputs['dense_1'] tensor_info:
      dtype: DT_FLOAT
      shape: (-1, 10)
      name: StatefulPartitionedCall:0
Method name is: tensorflow/serving/predict
```

Note that the function's input is named `"flatten_input"`, and the output is named `"dense_1"`. These correspond to the Keras model's input and output layer names. You can also see the type and shape of the input and output data. Looks good!

Now that you have a SavedModel, the next step is to install TF Serving.

Installing and starting TensorFlow Serving

There are many ways to install TF Serving: using the system's package manager, using a Docker image,[4] installing from source, and more. Since Colab runs on Ubuntu, we can use Ubuntu's `apt` package manager like this:

```
url = "https://storage.googleapis.com/tensorflow-serving-apt"
src = "stable tensorflow-model-server tensorflow-model-server-universal"
!echo 'deb {url} {src}' > /etc/apt/sources.list.d/tensorflow-serving.list
!curl '{url}/tensorflow-serving.release.pub.gpg' | apt-key add -
!apt update -q && apt-get install -y tensorflow-model-server
%pip install -q -U tensorflow-serving-api
```

This code starts by adding TensorFlow's package repository to Ubuntu's list of package sources. Then it downloads TensorFlow's public GPG key and adds it to the package manager's key list so it can verify TensorFlow's package signatures. Next, it uses `apt` to install the `tensorflow-model-`

`server` package. Lastly, it installs the `tensorflow-serving-api` library, which we will need to communicate with the server.

Now we want to start the server. The command will require the absolute path of the base model directory (i.e., the path to `my_mnist_model`, not `0001`), so let's save that to the `MODEL_DIR` environment variable:

```
import os

os.environ["MODEL_DIR"] = str(model_path.parent.absolute())
```

We can then start the server:

```
%%bash --bg
tensorflow_model_server \
     --port=8500 \
     --rest_api_port=8501 \
     --model_name=my_mnist_model \
     --model_base_path="${MODEL_DIR}" >my_server.log 2>&1
```

In Jupyter or Colab, the `%%bash --bg` magic command executes the cell as a bash script, running it in the background. The `>my_server.log 2>&1` part redirects the standard output and standard error to the *my_server.log* file. And that's it! TF Serving is now running in the background, and its logs are saved to *my_server.log*. It loaded our MNIST model (version 1), and it is now waiting for gRPC and REST requests, respectively, on ports 8500 and 8501.

RUNNING TF SERVING IN A DOCKER CONTAINER

If you are running the notebook on your own machine and you have installed Docker, you can run `docker pull tensorflow/serving` in a terminal to download the TF Serving image. The TensorFlow team highly recommends this installation method because it is simple, it will not mess with your system, and it offers high performance.[5] To start the server inside a Docker container, you can run the following command in a terminal:

```
$ docker run -it --rm -v "/path/to/my_mnist_model:/models/my_mnist_model" \
    -p 8500:8500 -p 8501:8501 -e MODEL_NAME=my_mnist_model tensorflow/serving
```

Here is what all these command-line options mean:

`-it`

Makes the container interactive (so you can press Ctrl-C to stop it) and displays the server's output.

`--rm`

Deletes the container when you stop it: no need to clutter your machine with interrupted containers. However, it does not delete the image.

`-v "/path/to/my_mnist_model:/models/my_mnist_model"`

Makes the host's *my_mnist_model* directory available to the container at the path */models/mnist_model*. You must replace */path/to/my_mnist_model* with the absolute path of this directory. On Windows, remember to use `\` instead of `/` in the host path, but not in the container path (since the container runs on Linux).

`-p 8500:8500`

Makes the Docker engine forward the host's TCP port 8500 to the container's TCP port 8500. By default, TF Serving uses this port to serve the gRPC API.

`-p 8501:8501`

Forwards the host's TCP port 8501 to the container's TCP port 8501. The Docker image is configured to use this port by default to serve the REST API.

`-e MODEL_NAME=my_mnist_model`

Sets the container's `MODEL_NAME` environment variable, so TF Serving knows which model to serve. By default, it will look for models in the */models* directory, and it will automatically serve the latest version it finds.

`tensorflow/serving`

This is the name of the image to run.

Now that the server is up and running, let's query it, first using the REST API, then the gRPC API.

Querying TF Serving through the REST API

Let's start by creating the query. It must contain the name of the function signature you want to call, and of course the input data. Since the request must use the JSON format, we have to convert the input images from a NumPy array to a Python list:

```
import json

X_new = X_test[:3]  # pretend we have 3 new digit images to classify
request_json = json.dumps({
    "signature_name": "serving_default",
    "instances": X_new.tolist(),
})
```

Note that the JSON format is 100% text-based. The request string looks like this:

```
>>> request_json
'{"signature_name": "serving_default", "instances": [[[0, 0, 0, 0, ... ]]]}'
```

Now let's send this request to TF Serving via an HTTP POST request. This can be done using the `requests` library (it is not part of Python's standard library, but it is preinstalled on Colab):

```
import requests

server_url = "http://localhost:8501/v1/models/my_mnist_model:predict"
response = requests.post(server_url, data=request_json)
response.raise_for_status()  # raise an exception in case of error
response = response.json()
```

If all goes well, the response should be a dictionary containing a single `"predictions"` key. The corresponding value is the list of predictions. This list is a Python list, so let's convert it to a NumPy array and round the floats it contains to the second decimal:

```
>>> import numpy as np
>>> y_proba = np.array(response["predictions"])
>>> y_proba.round(2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  ],
       [0.  , 0.  , 0.99, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.97, 0.01, 0.  , 0.  , 0.  , 0.01, 0.  , 0.  ]])
```

Hurray, we have the predictions! The model is close to 100% confident that the first image is a 7, 99% confident that the second image is a 2, and 97% confident that the third image is a 1. That's correct.

The REST API is nice and simple, and it works well when the input and output data are not too large. Moreover, just about any client application can make REST queries without additional dependencies, whereas other protocols are not always so readily available. However, it is based on JSON, which is text-based and fairly verbose. For example, we had to convert the NumPy array to a Python list, and every float ended up represented as a string. This is very inefficient, both in terms of serialization/deserialization time—we have to convert all the floats to strings and back—and in terms of payload size: many floats end up being represented using over 15 characters, which translates to over 120 bits for 32-bit floats! This will result in high latency and bandwidth usage when transferring large NumPy arrays.[6] So, let's see how to use gRPC instead.

TIP

When transferring large amounts of data, or when latency is important, it is much better to use the gRPC API, if the client supports it, as it uses a compact binary format and an efficient communication protocol based on HTTP/2 framing.

Querying TF Serving through the gRPC API

The gRPC API expects a serialized `PredictRequest` protocol buffer as input, and it outputs a serialized `PredictResponse` protocol buffer. These protobufs are part of the `tensorflow-serving-api` library, which we installed earlier. First, let's create the request:

```
from tensorflow_serving.apis.predict_pb2 import PredictRequest

request = PredictRequest()
```

```
request.model_spec.name = model_name
request.model_spec.signature_name = "serving_default"
input_name = model.input_names[0]  # == "flatten_input"
request.inputs[input_name].CopyFrom(tf.make_tensor_proto(X_new))
```

This code creates a `PredictRequest` protocol buffer and fills in the required fields, including the model name (defined earlier), the signature name of the function we want to call, and finally the input data, in the form of a `Tensor` protocol buffer. The `tf.make_tensor_proto()` function creates a `Tensor` protocol buffer based on the given tensor or NumPy array, in this case `X_new`.

Next, we'll send the request to the server and get its response. For this, we will need the `grpcio` library, which is preinstalled in Colab:

```
import grpc
from tensorflow_serving.apis import prediction_service_pb2_grpc

channel = grpc.insecure_channel('localhost:8500')
predict_service = prediction_service_pb2_grpc.PredictionServiceStub(channel)
response = predict_service.Predict(request, timeout=10.0)
```

The code is quite straightforward: after the imports, we create a gRPC communication channel to *localhost* on TCP port 8500, then we create a gRPC service over this channel and use it to send a request, with a 10-second timeout. Note that the call is synchronous: it will block until it receives the response or when the timeout period expires. In this example the channel is insecure (no encryption, no authentication), but gRPC and TF Serving also support secure channels over SSL/TLS.

Next, let's convert the `PredictResponse` protocol buffer to a tensor:

```
output_name = model.output_names[0]  # == "dense_1"
outputs_proto = response.outputs[output_name]
y_proba = tf.make_ndarray(outputs_proto)
```

If you run this code and print `y_proba.round(2)`, you will get the exact same estimated class probabilities as earlier. And that's all there is to it: in just a few lines of code, you can now access your TensorFlow model remotely, using either REST or gRPC.

Deploying a new model version

Now let's create a new model version and export a SavedModel, this time to the *my_mnist_model/0002* directory:

```
model = [...]  # build and train a new MNIST model version

model_version = "0002"
model_path = Path(model_name) / model_version
model.save(model_path, save_format="tf")
```

At regular intervals (the delay is configurable), TF Serving checks the model directory for new model versions. If it finds one, it automatically handles the transition gracefully: by default, it answers pending requests (if any) with the previous model version, while handling new requests with the new version. As soon as every pending request has been answered, the previous model version is unloaded. You can see this at work in the TF Serving logs (in *my_server.log*):

```
[...]
Reading SavedModel from: /models/my_mnist_model/0002
Reading meta graph with tags { serve }
[...]
Successfully loaded servable version {name: my_mnist_model version: 2}
Quiescing servable version {name: my_mnist_model version: 1}
Done quiescing servable version {name: my_mnist_model version: 1}
Unloading servable version {name: my_mnist_model version: 1}
```

TIP

If the SavedModel contains some example instances in the *assets/extra* directory, you can configure TF Serving to run the new model on these instances before starting to use it to serve requests. This is called *model warmup*: it will ensure that everything is properly loaded, avoiding long response times for the first requests.

This approach offers a smooth transition, but it may use too much RAM—especially GPU RAM, which is generally the most limited. In this case, you can configure TF Serving so that it handles all pending requests with the previous model version and unloads it before loading and using the new model version. This configuration will avoid having two model versions loaded at the same time, but the service will be unavailable for a short period.

As you can see, TF Serving makes it straightforward to deploy new models. Moreover, if you discover that version 2 does not work as well as you expected, then rolling back to version 1 is as simple as removing the *my_mnist_model/0002* directory.

TIP

Another great feature of TF Serving is its automatic batching capability, which you can activate using the `--enable_batching` option upon startup. When TF Serving receives multiple requests within a short period of time (the delay is configurable), it will automatically batch them together before using the model. This offers a significant performance boost by leveraging the power of the GPU. Once the model returns the predictions, TF Serving dispatches each prediction to the right client. You can trade a bit of latency for a greater throughput by increasing the batching delay (see the `--batching_parameters_file` option).

If you expect to get many queries per second, you will want to deploy TF Serving on multiple servers and load-balance the queries (see Figure 19-2). This will require deploying and managing many TF Serving containers across these servers. One way to handle that is to use a tool such as Kubernetes, which is an open source system for simplifying container orchestration across many servers. If you do not want to purchase, maintain, and upgrade all the hardware infrastructure, you will want to use virtual machines on a cloud platform such as Amazon AWS, Microsoft Azure, Google Cloud Platform, IBM Cloud, Alibaba Cloud, Oracle Cloud, or some other platform as a service (PaaS) offering. Managing all the virtual machines, handling container orchestration (even with the help of Kubernetes), taking care of TF Serving configuration, tuning and monitoring—all of this can be a full-time job. Fortunately, some service providers can take care of all this for you. In this chapter we will use Vertex AI: it's the only platform with TPUs today; it supports TensorFlow 2, Scikit-Learn, and XGBoost; and it offers a nice suite of AI services. There are several other providers in this space that are capable of serving TensorFlow models as well, though, such as Amazon AWS SageMaker and Microsoft AI Platform, so make sure to check them out too.

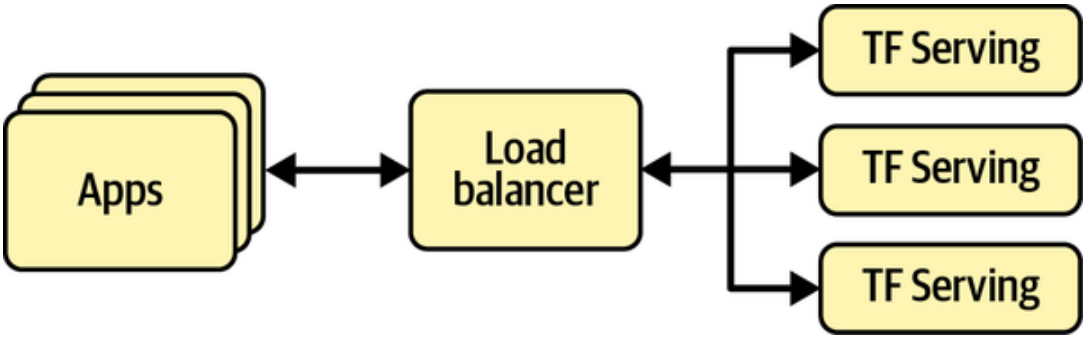

Figure 19-2. Scaling up TF Serving with load balancing

Now let's see how to serve our wonderful MNIST model on the cloud!

Creating a Prediction Service on Vertex AI

Vertex AI is a platform within Google Cloud Platform (GCP) that offers a wide range of AI-related tools and services. You can upload datasets, get humans to label them, store commonly used features in a feature store and use them for training or in production, and train models across many GPU or TPU servers with automatic hyperparameter tuning or model architecture search (AutoML). You can also manage your trained models, use them to make batch predictions on large amounts of data, schedule multiple jobs for your data workflows, serve your models via REST or gRPC at scale, and experiment with your data and models within a hosted Jupyter environment called the *Workbench*. There's even a *Matching Engine* service that lets you compare vectors very efficiently (i.e., approximate nearest neighbors). GCP also includes other AI services, such as APIs for computer vision, translation, speech-to-text, and more.

Before we start, there's a little bit of setup to take care of:

1. Log in to your Google account, and then go to the Google Cloud Platform console (see Figure 19-3). If you don't have a Google account, you'll have to create one.
2. If it's your first time using GCP, you'll have to read and accept the terms and conditions. New users are offered a free trial, including $300 worth of GCP credit that you can use over the course of 90 days (as of May 2022). You'll only need a small portion of that to pay for the services you'll use in this chapter. Upon signing up for a free trial, you'll still need to create a payment profile and enter your credit card number: it's used for verification purposes—probably to avoid people using the free trial multiple times—but you won't be billed for the first $300, and after that you'll only be charged if you opt in by upgrading to a paid account.

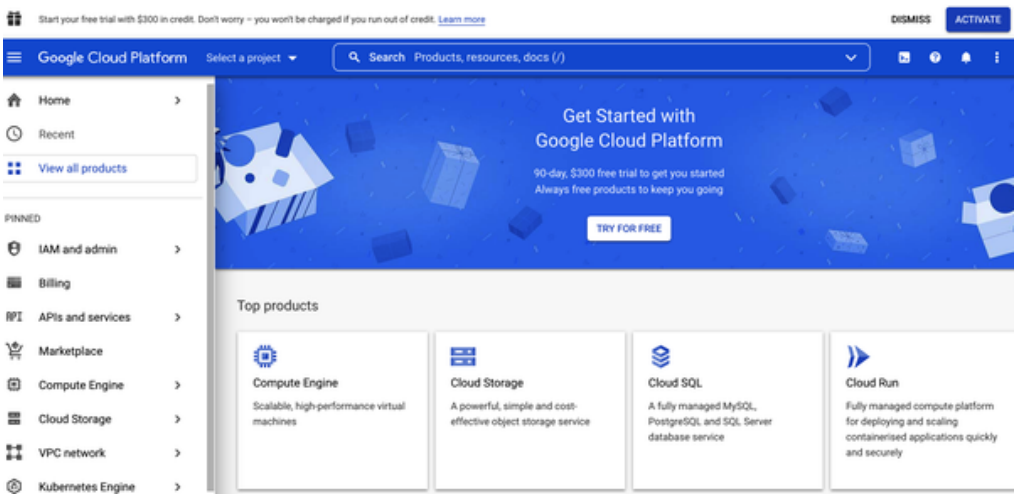

Figure 19-3. Google Cloud Platform console

3. If you have used GCP before and your free trial has expired, then the services you will use in this chapter will cost you some money. It shouldn't be too much, especially if you remember to turn off the services when you don't need them anymore. Make sure you understand and agree to the pricing conditions before you run any service. I hereby decline any responsibility if services end up costing more than you expected! Also make sure your billing account is active. To check, open the ☰ navigation menu at the top left and click Billing, then make sure you have set up a payment method and that the billing account is active.
4. Every resource in GCP belongs to a *project*. This includes all the virtual machines you may use, the files you store, and the training jobs you run. When you create an account, GCP automatically creates a project for you, called "My First Project". If you want, you can change its display name by going to the project settings: in the ☰ navigation menu, select "IAM and admin → Settings", change the project's display name, and click SAVE. Note that the project also has a unique ID and number. You can choose the project ID when you create a project, but you cannot change it later. The project number is automatically generated and cannot be changed. If you want to create a new project, click the project name at the top of the page, then click NEW PROJECT and enter the project name. You can also click EDIT to set the project ID. Make sure billing is active for this new project so that service fees can be billed (to your free credits, if any).

WARNING

Always set an alarm to remind yourself to turn services off when you know you will only need them for a few hours, or else you might leave them running for days or months, incurring potentially significant costs.

---

5. Now that you have a GCP account and a project, and billing is activated, you must activate the APIs you need. In the ☰ navigation menu, select "APIs and services", and make sure the Cloud Storage API is enabled. If needed, click + ENABLE APIS AND SERVICES, find Cloud Storage, and enable it. Also enable the Vertex AI API.

You could continue to do everything via the GCP console, but I recommend using Python instead: this way you can write scripts to automate just about anything you want with GCP, and it's often more convenient than clicking your way through menus and forms, especially for common tasks.

GOOGLE CLOUD CLI AND SHELL

Google Cloud's command-line interface (CLI) includes the `gcloud` command, which lets you control almost everything in GCP, and `gsutil`, which lets you interact with Google Cloud Storage. This CLI is preinstalled in Colab: all you need to do is authenticate using `google.auth.authenticate_user()`, and you're good to go. For example, `!gcloud config list` will display the configuration.

GCP also offers a preconfigured shell environment called the Google Cloud Shell, which you can use directly in your web browser; it runs on a free Linux VM (Debian) with the Google Cloud SDK already preinstalled and configured for you, so there's no need to authenticate. The Cloud Shell is available anywhere in GCP: just click the Activate Cloud Shell icon at the top right of the page (see Figure 19-4).

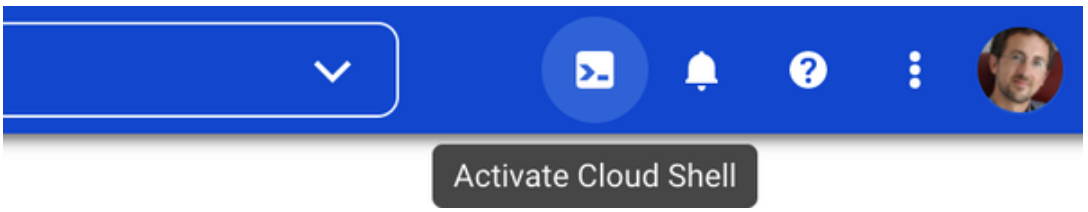

Figure 19-4. Activating the Google Cloud Shell

If you prefer to install the CLI on your machine, then after installation you need to initialize it by running `gcloud init`: follow the instructions to log in to GCP and grant access to your GCP resources, then select the default GCP project you want to use (if you have more than one) and the default region where you want your jobs to run.

The first thing you need to do before you can use any GCP service is to authenticate. The simplest solution when using Colab is to execute the following code:

```
from google.colab import auth

auth.authenticate_user()
```

The authentication process is based on *OAuth 2.0*: a pop-up window will ask you to confirm that you want the Colab notebook to access your Google credentials. If you accept, you must select the same Google account you used for GCP. Then you will be asked to confirm that you agree to give Colab full access to all your data on Google Drive and in GCP. If you allow access, only the current notebook will have access, and only until the Colab runtime expires. Obviously, you should only accept this if you trust the code in the notebook.

WARNING

If you are *not* working with the official notebooks from *https://github.com/ageron/handson-ml3*, then you should be extra careful: if the notebook's author is mischievous, they could include code to do whatever they want with your data.

AUTHENTICATION AND AUTHORIZATION ON GCP

In general, using OAuth 2.0 authentication is only recommended when an application must access the user's personal data or resources from another application, on the user's behalf. For example, some applications allow the user to save data to their Google Drive, but for that the application first needs the user to authenticate with Google and allow access to Google Drive. In general, the application will only ask for the level of access it needs; it won't be an unlimited access: for example, the application will only request access to Google Drive, not Gmail or any other Google service. Moreover, the authorization usually expires after a while, and it can always be revoked.

When an application needs to access a service on GCP on its own behalf, not on behalf of the user, then it should generally use a *service account*. For example, if you build a website that needs to send prediction requests to a Vertex AI endpoint, then the website will be accessing the service on

its own behalf. There's no data or resource that it needs to access in the user's Google account. In fact, many users of the website will not even have a Google account. For this scenario, you first need to create a service account. Select "IAM and admin → Service accounts" in the GCP console's ☰ navigation menu (or use the search box), then click + CREATE SERVICE ACCOUNT, fill in the first page of the form (service account name, ID, description), and click CREATE AND CONTINUE. Next, you must give this account some access rights. Select the "Vertex AI user" role: this will allow the service account to make predictions and use other Vertex AI services, but nothing else. Click CONTINUE. You can now optionally grant some users access to the service account: this is useful when your GCP user account is part of an organization and you wish to authorize other users in the organization to deploy applications that will be based on this service account, or to manage the service account itself. Next, click DONE.

Once you have created a service account, your application must authenticate as that service account. There are several ways to do that. If your application is hosted on GCP—for example, if you are coding a website hosted on Google Compute Engine—then the simplest and safest solution is to attach the service account to the GCP resource that hosts your website, such as a VM instance or a Google App Engine service. This can be done when creating the GCP resource, by selecting the service account in the "Identity and API access" section. Some resources, such as VM instances, also let you attach the service account after the VM instance is created: you must stop it and edit its settings. In any case, once a service account is attached to a VM instance, or any other GCP resource running your code, GCP's client libraries (discussed shortly) will automatically authenticate as the chosen service account, with no extra step needed.

If your application is hosted using Kubernetes, then you should use Google's Workload Identity service to map the right service account to each Kubernetes service account. If your application is not hosted on GCP—for example, if you are just running the Jupyter notebook on your own machine—then you can either use the Workload Identity Federation service (that's the safest but hardest option), or just generate an access key for your service account, save it to a JSON file, and point the `GOOGLE_APPLICATION_CREDENTIALS` environment variable to it so your client application can access it. You can manage access keys by clicking the service account you just created, and then opening the KEYS tab. Make sure to keep the key file secret: it's like a password for the service account.

For more details on setting up authentication and authorization so your application can access GCP services, check out the documentation.

Now let's create a Google Cloud Storage bucket to store our SavedModels (a GCS *bucket* is a container for your data). For this we will use the `google-cloud-storage` library, which is preinstalled in Colab. We first create a `Client` object, which will serve as the interface with GCS, then we use it to create the bucket:

```
from google.cloud import storage

project_id = "my_project"  # change this to your project ID
bucket_name = "my_bucket"  # change this to a unique bucket name
location = "us-central1"

storage_client = storage.Client(project=project_id)
bucket = storage_client.create_bucket(bucket_name, location=location)
```

TIP

If you want to reuse an existing bucket, replace the last line with `bucket = storage_client.bucket(bucket_name)`. Make sure `location` is set to the bucket's region.

GCS uses a single worldwide namespace for buckets, so simple names like "machine-learning" will most likely not be available. Make sure the bucket name conforms to DNS naming conventions, as it may be used in DNS records. Moreover, bucket names are public, so do not put anything private in the name. It is common to use your domain name, your company name, or your project ID as a prefix to ensure uniqueness, or simply use a random number as part of the name.

You can change the region if you want, but be sure to choose one that supports GPUs. Also, you may want to consider the fact that prices vary greatly between regions, some regions produce much more CO_2 than others, some regions do not support all services, and using a single-region bucket improves performance. See Google Cloud's list of regions and Vertex AI's documentation on locations for more details. If you are unsure, it might be best to stick with `"us-central1"`.

Next, let's upload the *my_mnist_model* directory to the new bucket. Files in GCS are called *blobs* (or *objects*), and under the hood they are all just placed in the bucket without any directory structure. Blob names can be arbitrary Unicode strings, and they can even contain forward slashes (/). The GCP console and other tools use these slashes to give the illusion that there are directories. So, when we upload the *my_mnist_model* directory, we only care about the files, not the directories:

```
def upload_directory(bucket, dirpath):
    dirpath = Path(dirpath)
    for filepath in dirpath.glob("**/*"):
        if filepath.is_file():
            blob = bucket.blob(filepath.relative_to(dirpath.parent).as_posix()
            blob.upload_from_filename(filepath)

upload_directory(bucket, "my_mnist_model")
```

This function works fine now, but it would be very slow if there were many files to upload. It's not too hard to speed it up tremendously by multithreading it (see the notebook for an implementation). Alternatively, if you have the Google Cloud CLI, then you can use following command instead:

```
!gsutil -m cp -r my_mnist_model gs://{bucket_name}/
```

Next, let's tell Vertex AI about our MNIST model. To communicate with Vertex AI, we can use the `google-cloud-aiplatform` library (it still uses the old AI Platform name instead of Vertex AI). It's not preinstalled in Colab, so we need to install it. After that, we can import the library and initialize it—just to specify some default values for the project ID and the location—then we can create a new Vertex AI model: we specify a display name, the GCS path to our model (in this case the version 0001), and the URL of the Docker container we want Vertex AI to use to run this model. If you visit that URL and navigate up one level, you will find other containers you can use. This one supports TensorFlow 2.8 with a GPU:

```
from google.cloud import aiplatform

server_image = "gcr.io/cloud-aiplatform/prediction/tf2-gpu.2-8:latest"

aiplatform.init(project=project_id, location=location)
mnist_model = aiplatform.Model.upload(
```

```
    display_name="mnist",
    artifact_uri=f"gs://{bucket_name}/my_mnist_model/0001",
    serving_container_image_uri=server_image,
)
```

Now let's deploy this model so we can query it via a gRPC or REST API to make predictions. For this we first need to create an *endpoint*. This is what client applications connect to when they want to access a service. Then we need to deploy our model to this endpoint:

```
endpoint = aiplatform.Endpoint.create(display_name="mnist-endpoint")

endpoint.deploy(
    mnist_model,
    min_replica_count=1,
    max_replica_count=5,
    machine_type="n1-standard-4",
    accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=1
)
```

This code may take a few minutes to run, because Vertex AI needs to set up a virtual machine. In this example, we use a fairly basic machine of type `n1-standard-4` (see *https://homl.info/machinetypes* for other types). We also use a basic GPU of type `NVIDIA_TESLA_K80` (see *https://homl.info/accelerators* for other types). If you selected another region than `"us-central1"`, then you may need to change the machine type or the accelerator type to values that are supported in that region (e.g., not all regions have Nvidia Tesla K80 GPUs).

NOTE

Google Cloud Platform enforces various GPU quotas, both worldwide and per region: you cannot create thousands of GPU nodes without prior authorization from Google. To check your quotas, open "IAM and admin → Quotas" in the GCP console. If some quotas are too low (e.g., if you need more GPUs in a particular region), you can ask for them to be increased; it often takes about 48 hours.

Vertex AI will initially spawn the minimum number of compute nodes (just one in this case), and whenever the number of queries per second becomes too high, it will spawn more nodes (up to the maximum number you defined, five in this case) and will load-balance the queries between

them. If the QPS rate goes down for a while, Vertex AI will stop the extra compute nodes automatically. The cost is therefore directly linked to the load, as well as the machine and accelerator types you selected and the amount of data you store on GCS. This pricing model is great for occasional users and for services with important usage spikes. It's also ideal for startups: the price remains low until the startup actually starts up.

Congratulations, you have deployed your first model to the cloud! Now let's query this prediction service:

```
response = endpoint.predict(instances=X_new.tolist())
```

We first need to convert the images we want to classify to a Python list, as we did earlier when we sent requests to TF Serving using the REST API. The response object contains the predictions, represented as a Python list of lists of floats. Let's round them to two decimal places and convert them to a NumPy array:

```
>>> import numpy as np
>>> np.round(response.predictions, 2)
array([[0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 1.  , 0.  , 0.  ],
       [0.  , 0.  , 0.99, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  ],
       [0.  , 0.97, 0.01, 0.  , 0.  , 0.  , 0.01, 0.  , 0.  ]])
```

Yes! We get the exact same predictions as earlier. We now have a nice prediction service running on the cloud that we can query from anywhere securely, and which can automatically scale up or down depending on the number of QPS. When you are done using the endpoint, don't forget to delete it, to avoid paying for nothing:

```
endpoint.undeploy_all()  # undeploy all models from the endpoint
endpoint.delete()
```

Now let's see how to run a job on Vertex AI to make predictions on a potentially very large batch of data.

Running Batch Prediction Jobs on Vertex AI

If we have a large number of predictions to make, then instead of calling our prediction service repeatedly, we can ask Vertex AI to run a predic-

tion job for us. This does not require an endpoint, only a model. For example, let's run a prediction job on the first 100 images of the test set, using our MNIST model. For this, we first need to prepare the batch and upload it to GCS. One way to do this is to create a file containing one instance per line, each formatted as a JSON value—this format is called *JSON Lines*—then pass this file to Vertex AI. So let's create a JSON Lines file in a new directory, then upload this directory to GCS:

```
batch_path = Path("my_mnist_batch")
batch_path.mkdir(exist_ok=True)
with open(batch_path / "my_mnist_batch.jsonl", "w") as jsonl_file:
    for image in X_test[:100].tolist():
        jsonl_file.write(json.dumps(image))
        jsonl_file.write("\n")

upload_directory(bucket, batch_path)
```

Now we're ready to launch the prediction job, specifying the job's name, the type and number of machines and accelerators to use, the GCS path to the JSON Lines file we just created, and the path to the GCS directory where Vertex AI will save the model's predictions:

```
batch_prediction_job = mnist_model.batch_predict(
    job_display_name="my_batch_prediction_job",
    machine_type="n1-standard-4",
    starting_replica_count=1,
    max_replica_count=5,
    accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=1,
    gcs_source=[f"gs://{bucket_name}/{batch_path.name}/my_mnist_batch.jsonl"],
    gcs_destination_prefix=f"gs://{bucket_name}/my_mnist_predictions/",
    sync=True  # set to False if you don't want to wait for completion
)
```

TIP

For large batches, you can split the inputs into multiple JSON Lines files and list them all via the `gcs_source` argument.

This will take a few minutes, mostly to spawn the compute nodes on Vertex AI. Once this command completes, the predictions will be available in a set of files named something like *prediction.results-00001-of-*

00002. These files use the JSON Lines format by default, and each value is a dictionary containing an instance and its corresponding prediction (i.e., 10 probabilities). The instances are listed in the same order as the inputs. The job also outputs *prediction-errors** files, which can be useful for debugging if something goes wrong. We can iterate through all these output files using `batch_prediction_job.iter_outputs()`, so let's go through all the predictions and store them in a `y_probas` array:

```
y_probas = []
for blob in batch_prediction_job.iter_outputs():
    if "prediction.results" in blob.name:
        for line in blob.download_as_text().splitlines():
            y_proba = json.loads(line)["prediction"]
            y_probas.append(y_proba)
```

Now let's see how good these predictions are:

```
>>> y_pred = np.argmax(y_probas, axis=1)
>>> accuracy = np.sum(y_pred == y_test[:100]) / 100
0.98
```

Nice, 98% accuracy!

The JSON Lines format is the default, but when dealing with large instances such as images, it is too verbose. Luckily, the `batch_predict()` method accepts an `instances_format` argument that lets you choose another format if you want. It defaults to `"jsonl"`, but you can change it to `"csv"`, `"tf-record"`, `"tf-record-gzip"`, `"bigquery"`, or `"file-list"`. If you set it to `"file-list"`, then the `gcs_source` argument should point to a text file containing one input filepath per line; for instance, pointing to PNG image files. Vertex AI will read these files as binary, encode them using Base64, and pass the resulting byte strings to the model. This means that you must add a preprocessing layer in your model to parse the Base64 strings, using `tf.io.decode_base64()`. If the files are images, you must then parse the result using a function like `tf.io.decode_image()` or `tf.io.decode_png()`, as discussed in Chapter 13.

When you're finished using the model, you can delete it if you want, by running `mnist_model.delete()`. You can also delete the directories you

created in your GCS bucket, optionally the bucket itself (if it's empty), and the batch prediction job:

```
for prefix in ["my_mnist_model/", "my_mnist_batch/", "my_mnist_predictions/"]:
    blobs = bucket.list_blobs(prefix=prefix)
    for blob in blobs:
        blob.delete()

bucket.delete()  # if the bucket is empty
batch_prediction_job.delete()
```

You now know how to deploy a model to Vertex AI, create a prediction service, and run batch prediction jobs. But what if you want to deploy your model to a mobile app instead? Or to an embedded device, such as a heating control system, a fitness tracker, or a self-driving car?

Deploying a Model to a Mobile or Embedded Device

Machine learning models are not limited to running on big centralized servers with multiple GPUs: they can run closer to the source of data (this is called *edge computing*), for example in the user's mobile device or in an embedded device. There are many benefits to decentralizing the computations and moving them toward the edge: it allows the device to be smart even when it's not connected to the internet, it reduces latency by not having to send data to a remote server and reduces the load on the servers, and it may improve privacy, since the user's data can stay on the device.

However, deploying models to the edge has its downsides too. The device's computing resources are generally tiny compared to a beefy multi-GPU server. A large model may not fit in the device, it may use too much RAM and CPU, and it may take too long to download. As a result, the application may become unresponsive, and the device may heat up and quickly run out of battery. To avoid all this, you need to make a lightweight and efficient model, without sacrificing too much of its accuracy. The TFLite library provides several tools[7] to help you deploy your models to the edge, with three main objectives:

- Reduce the model size, to shorten download time and reduce RAM usage.
- Reduce the amount of computations needed for each prediction, to reduce latency, battery usage, and heating.
- Adapt the model to device-specific constraints.

To reduce the model size, TFLite's model converter can take a SavedModel and compress it to a much lighter format based on FlatBuffers. This is an efficient cross-platform serialization library (a bit like protocol buffers) initially created by Google for gaming. It is designed so you can load FlatBuffers straight to RAM without any preprocessing: this reduces the loading time and memory footprint. Once the model is loaded into a mobile or embedded device, the TFLite interpreter will execute it to make predictions. Here is how you can convert a SavedModel to a FlatBuffer and save it to a *.tflite* file:

```
converter = tf.lite.TFLiteConverter.from_saved_model(str(model_path))
tflite_model = converter.convert()
with open("my_converted_savedmodel.tflite", "wb") as f:
    f.write(tflite_model)
```

TIP

You can also save a Keras model directly to a FlatBuffer using `tf.lite.TFLiteConverter.from_keras_model(model)`.

The converter also optimizes the model, both to shrink it and to reduce its latency. It prunes all the operations that are not needed to make predictions (such as training operations), and it optimizes computations whenever possible; for example, $3 \times a + 4 \times$_ a_ $+ 5 \times a$ will be converted to $12 \times a$. Addtionally, it tries to fuse operations whenever possible. For example, if possible, batch normalization layers end up folded into the previous layer's addition and multiplication operations. To get a good idea of how much TFLite can optimize a model, download one of the pretrained TFLite models, such as *Inception_V1_quant* (click *tflite&pb*), unzip the archive, then open the excellent Netron graph visualization tool and upload the *.pb* file to view the original model. It's a big, complex graph, right? Next, open the optimized *.tflite* model and marvel at its beauty!

Another way you can reduce the model size—other than simply using smaller neural network architectures—is by using smaller bit-widths: for example, if you use half-floats (16 bits) rather than regular floats (32 bits), the model size will shrink by a factor of 2, at the cost of a (generally small) accuracy drop. Moreover, training will be faster, and you will use roughly half the amount of GPU RAM.

TFLite's converter can go further than that, by quantizing the model weights down to fixed-point, 8-bit integers! This leads to a fourfold size reduction compared to using 32-bit floats. The simplest approach is called *post-training quantization*: it just quantizes the weights after training, using a fairly basic but efficient symmetrical quantization technique. It finds the maximum absolute weight value, *m*, then it maps the floating-point range –*m* to +*m* to the fixed-point (integer) range –127 to +127. For example, if the weights range from –1.5 to +0.8, then the bytes –127, 0, and +127 will correspond to the floats –1.5, 0.0, and +1.5, respectively (see Figure 19-5). Note that 0.0 always maps to 0 when using symmetrical quantization. Also note that the byte values +68 to +127 will not be used in this example, since they map to floats greater than +0.8.

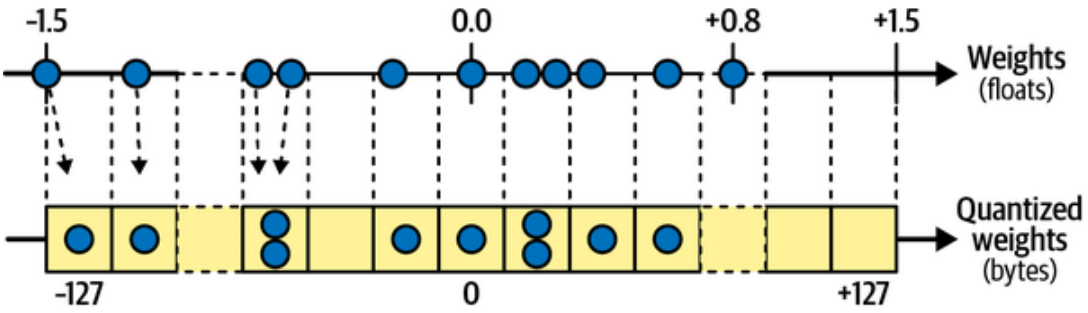

Figure 19-5. From 32-bit floats to 8-bit integers, using symmetrical quantization

To perform this post-training quantization, simply add `DEFAULT` to the list of converter optimizations before calling the `convert()` method:

```
converter.optimizations = [tf.lite.Optimize.DEFAULT]
```

This technique dramatically reduces the model's size, which makes it much faster to download, and uses less storage space. At runtime the quantized weights get converted back to floats before they are used. These recovered floats are not perfectly identical to the original floats, but they're not too far off, so the accuracy loss is usually acceptable. To avoid recomputing the float values all the time, which would severely slow down the model, TFLite caches them: unfortunately, this means that

this technique does not reduce RAM usage, and it doesn't speed up the model either. It's mostly useful to reduce the application's size.

The most effective way to reduce latency and power consumption is to also quantize the activations so that the computations can be done entirely with integers, without the need for any floating-point operations. Even when using the same bit-width (e.g., 32-bit integers instead of 32-bit floats), integer computations use less CPU cycles, consume less energy, and produce less heat. And if you also reduce the bit-width (e.g., down to 8-bit integers), you can get huge speedups. Moreover, some neural network accelerator devices—such as Google's Edge TPU—can only process integers, so full quantization of both weights and activations is compulsory. This can be done post-training; it requires a calibration step to find the maximum absolute value of the activations, so you need to provide a representative sample of training data to TFLite (it does not need to be huge), and it will process the data through the model and measure the activation statistics required for quantization. This step is typically fast.

The main problem with quantization is that it loses a bit of accuracy: it is similar to adding noise to the weights and activations. If the accuracy drop is too severe, then you may need to use *quantization-aware training*. This means adding fake quantization operations to the model so it can learn to ignore the quantization noise during training; the final weights will then be more robust to quantization. Moreover, the calibration step can be taken care of automatically during training, which simplifies the whole process.

I have explained the core concepts of TFLite, but going all the way to coding a mobile or embedded application woud require a dedicated book. Fortunately, some exist: if you want to learn more about building TensorFlow applications for mobile and embedded devices, check out the O'Reilly books *TinyML: Machine Learning with TensorFlow on Arduino and Ultra-Low Power Micro-Controllers*, by Pete Warden (former lead of the TFLite team) and Daniel Situnayake and *AI and Machine Learning for On-Device Development*, by Laurence Moroney.

Now what if you want to use your model in a website, running directly in the user's browser?

Running a Model in a Web Page

Running your machine learning model on the client side, in the user's browser, rather than on the server side can be useful in many scenarios, such as:

- When your web application is often used in situations where the user's connectivity is intermittent or slow (e.g., a website for hikers), so running the model directly on the client side is the only way to make your website reliable.
- When you need the model's responses to be as fast as possible (e.g., for an online game). Removing the need to query the server to make predictions will definitely reduce the latency and make the website much more responsive.
- When your web service makes predictions based on some private user data, and you want to protect the user's privacy by making the predictions on the client side so that the private data never has to leave the user's machine.

For all these scenarios, you can use the TensorFlow.js (TFJS) JavaScript library. This library can load a TFLite model and make predictions directly in the user's browser. For example, the following JavaScript module imports the TFJS library, downloads a pretrained MobileNet model, and uses this model to classify an image and log the predictions. You can play with the code at *https://homl.info/tfjscode*, using Glitch.com, a website that lets you build web apps in your browser for free; click the PREVIEW button in the lower-right corner of the page to see the code in action:

```
import "https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest";
import "https://cdn.jsdelivr.net/npm/@tensorflow-models/mobilenet@1.0.0";

const image = document.getElementById("image");

mobilenet.load().then(model => {
    model.classify(image).then(predictions => {
        for (var i = 0; i < predictions.length; i++) {
            let className = predictions[i].className
            let proba = (predictions[i].probability * 100).toFixed(1)
            console.log(className + " : " + proba + "%");
        }
    });
});
```

It's even possible to turn this website into a *progressive web app* (PWA): this is a website that respects a number of criteria[8] that allow it to be

viewed in any browser, and even installed as a standalone app on a mobile device. For example, try visiting *https://homl.info/tfjswpa* on a mobile device: most modern browsers will ask you whether you would like to add TFJS Demo to your home screen. If you accept, you will see a new icon in your list of applications. Clicking this icon will load the TFJS Demo website inside its own window, just like a regular mobile app. A PWA can even be configured to work offline, by using a *service worker*: this is a JavaScript module that runs in its own separate thread in the browser and intercepts network requests, allowing it to cache resources so the PWA can run faster, or even entirely offline. It can also deliver push messages, run tasks in the background, and more. PWAs allow you to manage a single code base for the web and for mobile devices. They also make it easier to ensure that all users run the same version of your application. You can play with this TFJS Demo's PWA code on Glitch.com at *https://homl.info/wpacode*.

TIP

Check out many more demos of machine learning models running in your browser at *https://tensorflow.org/js/demos*.

TFJS also supports training a model directly in your web browser! And it's actually pretty fast. If your computer has a GPU card, then TFJS can generally use it, even if it's not an Nvidia card. Indeed, TFJS will use WebGL when it's available, and since modern web browsers generally support a wide range of GPU cards, TFJS actually supports more GPU cards than regular TensorFlow (which only supports Nvidia cards).

Training a model in a user's web browser can be especially useful to guarantee that this user's data remains private. A model can be trained centrally, and then fine-tuned locally, in the browser, based on that user's data. If you're interested in this topic, check out *federated learning*.

Once again, doing justice to this topic would require a whole book. If you want to learn more about TensorFlow.js, check out the O'reilly books *Practical Deep Learning for Cloud, Mobile, and Edge*, by Anirudh Koul et al., or *Learning TensorFlow.js*, by Gant Laborde.

Now that you've seen how to deploy TensorFlow models to TF Serving, or to the cloud with Vertex AI, or to mobile and embedded devices using

TFLite, or to a web browser using TFJS, let's discuss how to use GPUs to speed up computations.

Using GPUs to Speed Up Computations

In Chapter 11 we looked at several techniques that can considerably speed up training: better weight initialization, sophisticated optimizers, and so on. But even with all of these techniques, training a large neural network on a single machine with a single CPU can take hours, days, or even weeks, depending on the task. Thanks to GPUs, this training time can be reduced down to minutes or hours. Not only does this save an enormous amount of time, but it also means that you can experiment with various models much more easily, and frequently retrain your models on fresh data.

In the previous chapters, we used GPU-enabled runtimes on Google Colab. All you have to do for this is select "Change runtime type" from the Runtime menu, and choose the GPU accelerator type; TensorFlow automatically detects the GPU and uses it to speed up computations, and the code is exactly the same as without a GPU. Then, in this chapter you saw how to deploy your models to Vertex AI on multiple GPU-enabled compute nodes: it's just a matter of selecting the right GPU-enabled Docker image when creating the Vertex AI model, and selecting the desired GPU type when calling `endpoint.deploy()`. But what if you want to buy your own GPU? And what if you want to distribute the computations across the CPU and multiple GPU devices on a single machine (see Figure 19-6)? This is what we will discuss now, then later in this chapter we will discuss how to distribute computations across multiple servers.

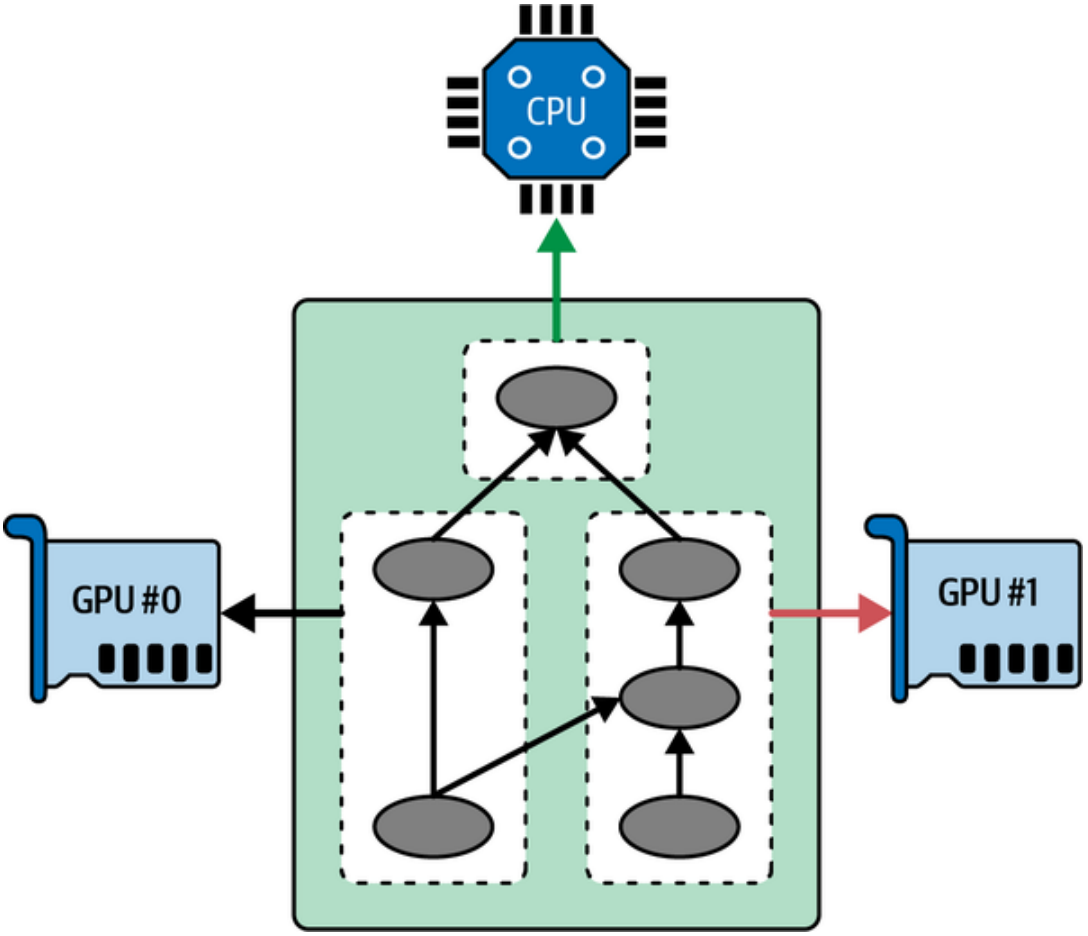

Figure 19-6. Executing a TensorFlow graph across multiple devices in parallel

Getting Your Own GPU

If you know that you'll be using a GPU heavily and for a long period of time, then buying your own can make financial sense. You may also want to train your models locally because you do not want to upload your data to the cloud. Or perhaps you just want to buy a GPU card for gaming, and you'd like to use it for deep learning as well.

If you decide to purchase a GPU card, then take some time to make the right choice. You will need to consider the amount of RAM you will need for your tasks (e.g., typically at least 10 GB for image processing or NLP), the bandwidth (i.e., how fast you can send data into and out of the GPU), the number of cores, the cooling system, etc. Tim Dettmers wrote an excellent blog post to help you choose: I encourage you to read it carefully. At the time of this writing, TensorFlow only supports Nvidia cards with CUDA Compute Capability 3.5+ (as well as Google's TPUs, of course), but it may extend its support to other manufacturers, so make sure to check TensorFlow's documentation to see what devices are supported today.

If you go for an Nvidia GPU card, you will need to install the appropriate Nvidia drivers and several Nvidia libraries.[9] These include the *Compute Unified Device Architecture* library (CUDA) Toolkit, which allows develop-

ers to use CUDA-enabled GPUs for all sorts of computations (not just graphics acceleration), and the *CUDA Deep Neural Network* library (cuDNN), a GPU-accelerated library of common DNN computations such as activation layers, normalization, forward and backward convolutions, and pooling (see Chapter 14). cuDNN is part of Nvidia's Deep Learning SDK. Note that you will need to create an Nvidia developer account in order to download it. TensorFlow uses CUDA and cuDNN to control the GPU cards and accelerate computations (see Figure 19-7).

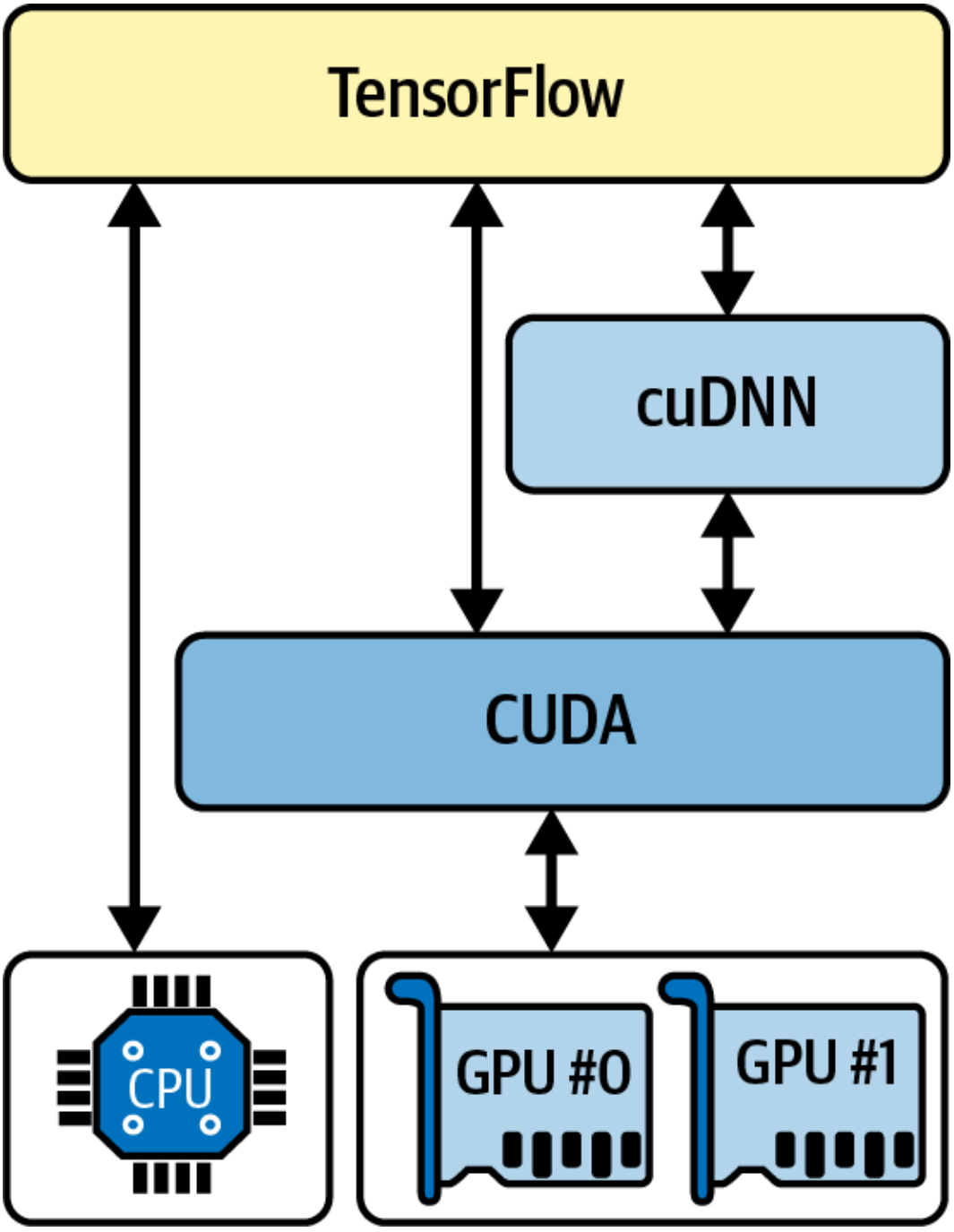

Figure 19-7. TensorFlow uses CUDA and cuDNN to control GPUs and boost DNNs

Once you have installed the GPU card(s) and all the required drivers and libraries, you can use the `nvidia-smi` command to check that everything is properly installed. This command lists the available GPU cards, as well as all the processes running on each card. In this example, it's an Nvidia

Tesla T4 GPU card with about 15 GB of available RAM, and there are no processes currently running on it:

```
$ nvidia-smi
Sun Apr 10 04:52:10 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 460.32.03    Driver Version: 460.32.03    CUDA Version: 11.2     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  Tesla T4            Off  | 00000000:00:04.0 Off |                    0 |
| N/A   34C    P8     9W /  70W |      3MiB / 15109MiB |      0%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|  No running processes found                                                 |
+-----------------------------------------------------------------------------+
```

To check that TensorFlow actually sees your GPU, run the following commands and make sure the result is not empty:

```
>>> physical_gpus = tf.config.list_physical_devices("GPU")
>>> physical_gpus
[PhysicalDevice(name='/physical_device:GPU:0', device_type='GPU')]
```

Managing the GPU RAM

By default TensorFlow automatically grabs almost all the RAM in all available GPUs the first time you run a computation. It does this to limit GPU RAM fragmentation. This means that if you try to start a second TensorFlow program (or any program that requires the GPU), it will quickly run out of RAM. This does not happen as often as you might think, as you will most often have a single TensorFlow program running on a machine: usually a training script, a TF Serving node, or a Jupyter notebook. If you need to run multiple programs for some reason (e.g., to train

two different models in parallel on the same machine), then you will need to split the GPU RAM between these processes more evenly.

If you have multiple GPU cards on your machine, a simple solution is to assign each of them to a single process. To do this, you can set the `CUDA_VISIBLE_DEVICES` environment variable so that each process only sees the appropriate GPU card(s). Also set the `CUDA_DEVICE_ORDER` environment variable to `PCI_BUS_ID` to ensure that each ID always refers to the same GPU card. For example, if you have four GPU cards, you could start two programs, assigning two GPUs to each of them, by executing commands like the following in two separate terminal windows:

```
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=0,1 python3 program_1.py
# and in another terminal:
$ CUDA_DEVICE_ORDER=PCI_BUS_ID CUDA_VISIBLE_DEVICES=3,2 python3 program_2.py
```

Program 1 will then only see GPU cards 0 and 1, named `"/gpu:0"` and `"/gpu:1"`, respectively, in TensorFlow, and program 2 will only see GPU cards 2 and 3, named `"/gpu:1"` and `"/gpu:0"`, respectively (note the order). Everything will work fine (see Figure 19-8). Of course, you can also define these environment variables in Python by setting `os.environ["CUDA_DEVICE_ORDER"]` and `os.environ["CUDA_VISIBLE_DEVICES"]`, as long as you do so before using TensorFlow.

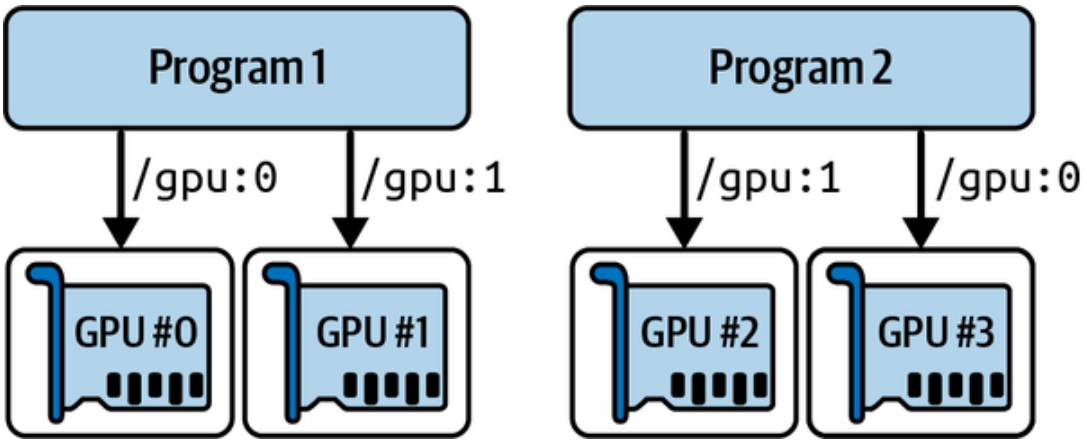

Figure 19-8. Each program gets two GPUs

Another option is to tell TensorFlow to grab only a specific amount of GPU RAM. This must be done immediately after importing TensorFlow. For example, to make TensorFlow grab only 2 GiB of RAM on each GPU, you must create a *logical GPU device* (sometimes called a *virtual GPU device*) for each physical GPU device and set its memory limit to 2 GiB (i.e., 2,048 MiB):

```
for gpu in physical_gpus:
    tf.config.set_logical_device_configuration(
        gpu,
        [tf.config.LogicalDeviceConfiguration(memory_limit=2048)]
    )
```

Let's suppose you have four GPUs, each with at least 4 GiB of RAM: in this case, two programs like this one can run in parallel, each using all four GPU cards (see Figure 19-9). If you run the `nvidia-smi` command while both programs are running, you should see that each process holds 2 GiB of RAM on each card.

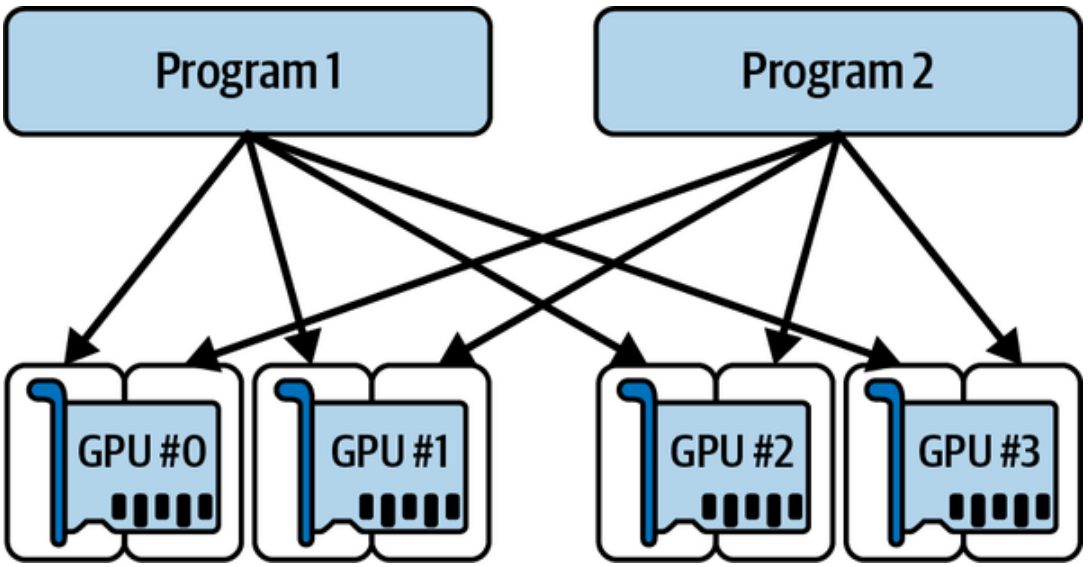

Figure 19-9. Each program gets all four GPUs, but with only 2 GiB of RAM on each GPU

Yet another option is to tell TensorFlow to grab memory only when it needs it. Again, this must be done immediately after importing TensorFlow:

```
for gpu in physical_gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
```

Another way to do this is to set the `TF_FORCE_GPU_ALLOW_GROWTH` environment variable to `true`. With this option, TensorFlow will never release memory once it has grabbed it (again, to avoid memory fragmentation), except of course when the program ends. It can be harder to guarantee deterministic behavior using this option (e.g., one program may crash because another program's memory usage went through the roof), so in production you'll probably want to stick with one of the previous options. However, there are some cases where it is very useful: for example, when you use a machine to run multiple Jupyter notebooks, several

of which use TensorFlow. The `TF_FORCE_GPU_ALLOW_GROWTH` environment variable is set to `true` in Colab runtimes.

Lastly, in some cases you may want to split a GPU into two or more *logical devices*. For example, this is useful if you only have one physical GPU—like in a Colab runtime—but you want to test a multi-GPU algorithm. The following code splits GPU #0 into two logical devices, with 2 GiB of RAM each (again, this must be done immediately after importing TensorFlow):

```
tf.config.set_logical_device_configuration(
    physical_gpus[0],
    [tf.config.LogicalDeviceConfiguration(memory_limit=2048),
     tf.config.LogicalDeviceConfiguration(memory_limit=2048)]
)
```

These two logical devices are called `"/gpu:0"` and `"/gpu:1"`, and you can use them as if they were two normal GPUs. You can list all logical devices like this:

```
>>> logical_gpus = tf.config.list_logical_devices("GPU")
>>> logical_gpus
[LogicalDevice(name='/device:GPU:0', device_type='GPU'),
 LogicalDevice(name='/device:GPU:1', device_type='GPU')]
```

Now let's see how TensorFlow decides which devices it should use to place variables and execute operations.

Placing Operations and Variables on Devices

Keras and tf.data generally do a good job of placing operations and variables where they belong, but you can also place operations and variables manually on each device, if you want more control:

- You generally want to place the data preprocessing operations on the CPU, and place the neural network operations on the GPUs.
- GPUs usually have a fairly limited communication bandwidth, so it is important to avoid unnecessary data transfers into and out of the GPUs.
- Adding more CPU RAM to a machine is simple and fairly cheap, so there's usually plenty of it, whereas the GPU RAM is baked into the GPU: it is an expensive and thus limited resource, so if a variable is

not needed in the next few training steps, it should probably be placed on the CPU (e.g., datasets generally belong on the CPU).

By default, all variables and all operations will be placed on the first GPU (the one named `"/gpu:0"`), except for variables and operations that don't have a GPU kernel:[10] these are placed on the CPU (always named `"/cpu:0"`). A tensor or variable's `device` attribute tells you which device it was placed on:[11]

```
>>> a = tf.Variable([1., 2., 3.])  # float32 variable goes to the GPU
>>> a.device
'/job:localhost/replica:0/task:0/device:GPU:0'
>>> b = tf.Variable([1, 2, 3])  # int32 variable goes to the CPU
>>> b.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

You can safely ignore the prefix `/job:localhost/replica:0/task:0` for now; we will discuss jobs, replicas, and tasks later in this chapter. As you can see, the first variable was placed on GPU #0, which is the default device. However, the second variable was placed on the CPU: this is because there are no GPU kernels for integer variables, or for operations involving integer tensors, so TensorFlow fell back to the CPU.

If you want to place an operation on a different device than the default one, use a `tf.device()` context:

```
>>> with tf.device("/cpu:0"):
...     c = tf.Variable([1., 2., 3.])
...
>>> c.device
'/job:localhost/replica:0/task:0/device:CPU:0'
```

NOTE

The CPU is always treated as a single device (`"/cpu:0"`), even if your machine has multiple CPU cores. Any operation placed on the CPU may run in parallel across multiple cores if it has a multithreaded kernel.

If you explicitly try to place an operation or variable on a device that does not exist or for which there is no kernel, then TensorFlow will silently fall back to the device it would have chosen by default. This is useful when

you want to be able to run the same code on different machines that don't have the same number of GPUs. However, you can run `tf.config.set_soft_device_placement(False)` if you prefer to get an exception.

Now, how exactly does TensorFlow execute operations across multiple devices?

Parallel Execution Across Multiple Devices

As we saw in Chapter 12, one of the benefits of using TF functions is parallelism. Let's look at this a bit more closely. When TensorFlow runs a TF function, it starts by analyzing its graph to find the list of operations that need to be evaluated, and it counts how many dependencies each of them has. TensorFlow then adds each operation with zero dependencies (i.e., each source operation) to the evaluation queue of this operation's device (see Figure 19-10). Once an operation has been evaluated, the dependency counter of each operation that depends on it is decremented. Once an operation's dependency counter reaches zero, it is pushed to the evaluation queue of its device. And once all the outputs have been computed, they are returned.

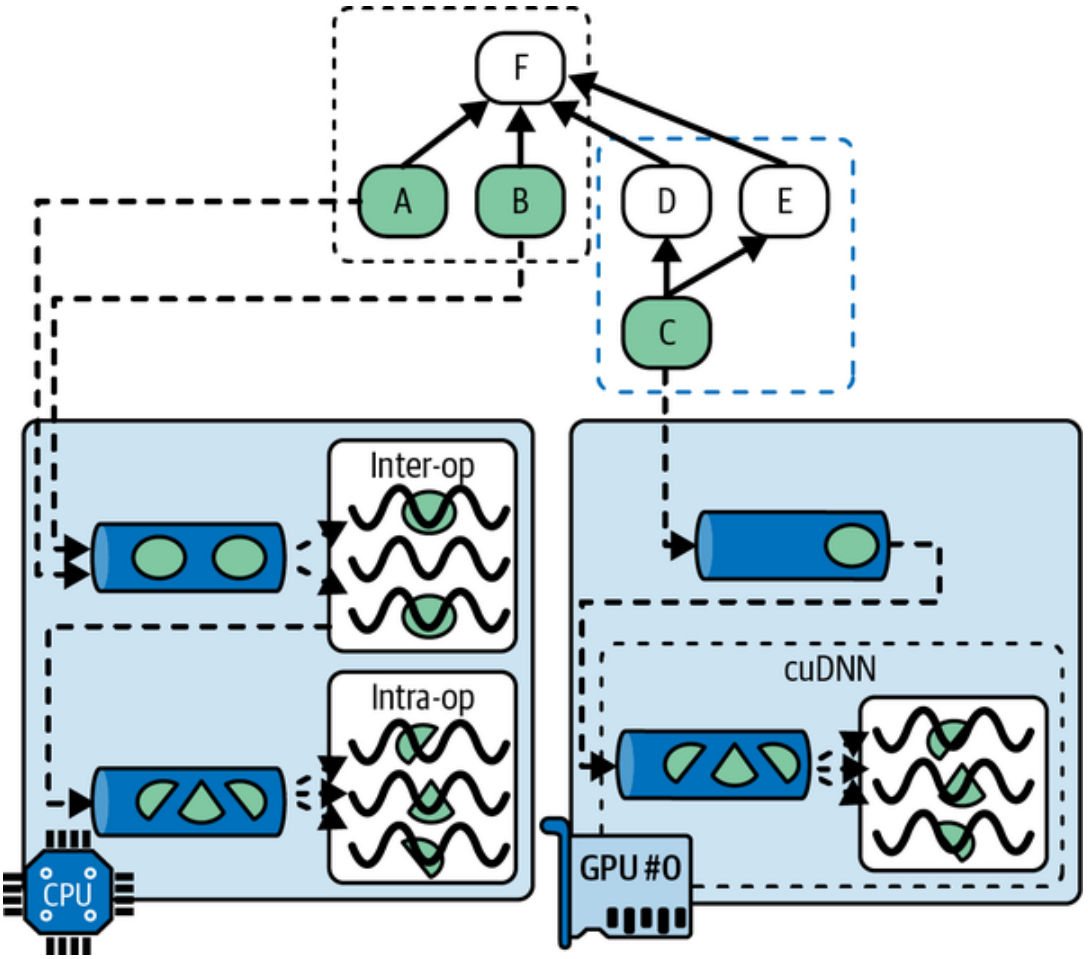

Figure 19-10. Parallelized execution of a TensorFlow graph

Operations in the CPU's evaluation queue are dispatched to a thread pool called the *inter-op thread pool*. If the CPU has multiple cores, then these operations will effectively be evaluated in parallel. Some operations have multithreaded CPU kernels: these kernels split their tasks into multiple suboperations, which are placed in another evaluation queue and dispatched to a second thread pool called the *intra-op thread pool* (shared by all multithreaded CPU kernels). In short, multiple operations and suboperations may be evaluated in parallel on different CPU cores.

For the GPU, things are a bit simpler. Operations in a GPU's evaluation queue are evaluated sequentially. However, most operations have multithreaded GPU kernels, typically implemented by libraries that TensorFlow depends on, such as CUDA and cuDNN. These implementations have their own thread pools, and they typically exploit as many GPU threads as they can (which is the reason why there is no need for an inter-op thread pool in GPUs: each operation already floods most GPU threads).

For example, in Figure 19-10, operations A, B, and C are source ops, so they can immediately be evaluated. Operations A and B are placed on the CPU, so they are sent to the CPU's evaluation queue, then they are dispatched to the inter-op thread pool and immediately evaluated in parallel. Operation A happens to have a multithreaded kernel; its computations are split into three parts, which are executed in parallel by the intra-op thread pool. Operation C goes to GPU #0's evaluation queue, and in this example its GPU kernel happens to use cuDNN, which manages its own intra-op thread pool and runs the operation across many GPU threads in parallel. Suppose C finishes first. The dependency counters of D and E are decremented and they reach 0, so both operations are pushed to GPU #0's evaluation queue, and they are executed sequentially. Note that C only gets evaluated once, even though both D and E depend on it. Suppose B finishes next. Then F's dependency counter is decremented from 4 to 3, and since that's not 0, it does not run yet. Once A, D, and E are finished, then F's dependency counter reaches 0, and it is pushed to the CPU's evaluation queue and evaluated. Finally, TensorFlow returns the requested outputs.

An extra bit of magic that TensorFlow performs is when the TF function modifies a stateful resource, such as a variable: it ensures that the order of execution matches the order in the code, even if there is no explicit dependency between the statements. For example, if your TF function con-

tains `v.assign_add(1)` followed by `v.assign(v * 2)`, TensorFlow will ensure that these operations are executed in that order.

TIP

You can control the number of threads in the inter-op thread pool by calling `tf.config.threading.set_inter_op_parallelism_threads()`. To set the number of intra-op threads, use `tf.config.threading.set_intra_op_parallelism_threads()`. This is useful if you do not want TensorFlow to use all the CPU cores or if you want it to be single-threaded.[12]

With that, you have all you need to run any operation on any device, and exploit the power of your GPUs! Here are some of the things you could do:

- You could train several models in parallel, each on its own GPU: just write a training script for each model and run them in parallel, setting `CUDA_DEVICE_ORDER` and `CUDA_VISIBLE_DEVICES` so that each script only sees a single GPU device. This is great for hyperparameter tuning, as you can train in parallel multiple models with different hyperparameters. If you have a single machine with two GPUs, and it takes one hour to train one model on one GPU, then training two models in parallel, each on its own dedicated GPU, will take just one hour. Simple!
- You could train a model on a single GPU and perform all the preprocessing in parallel on the CPU, using the dataset's `prefetch()` method[13] to prepare the next few batches in advance so that they are ready when the GPU needs them (see Chapter 13).
- If your model takes two images as input and processes them using two CNNs before joining their outputs,[14] then it will probably run much faster if you place each CNN on a different GPU.
- You can create an efficient ensemble: just place a different trained model on each GPU so that you can get all the predictions much faster to produce the ensemble's final prediction.

But what if you want to speed up training by using multiple GPUs?

Training Models Across Multiple Devices

There are two main approaches to training a single model across multiple devices: *model parallelism*, where the model is split across the devices, and *data parallelism*, where the model is replicated across every device, and each replica is trained on a different subset of the data. Let's look at these two options.

Model Parallelism

So far we have trained each neural network on a single device. What if we want to train a single neural network across multiple devices? This requires chopping the model into separate chunks and running each chunk on a different device. Unfortunately, such model parallelism turns out to be pretty tricky, and its effectiveness really depends on the architecture of your neural network. For fully connected networks, there is generally not much to be gained from this approach (see Figure 19-11). Intuitively, it may seem that an easy way to split the model is to place each layer on a different device, but this does not work because each layer needs to wait for the output of the previous layer before it can do anything. So perhaps you can slice it vertically—for example, with the left half of each layer on one device, and the right part on another device? This is slightly better, since both halves of each layer can indeed work in parallel, but the problem is that each half of the next layer requires the output of both halves, so there will be a lot of cross-device communication (represented by the dashed arrows). This is likely to completely cancel out the benefit of the parallel computation, since cross-device communication is slow (and even more so when the devices are located on different machines).

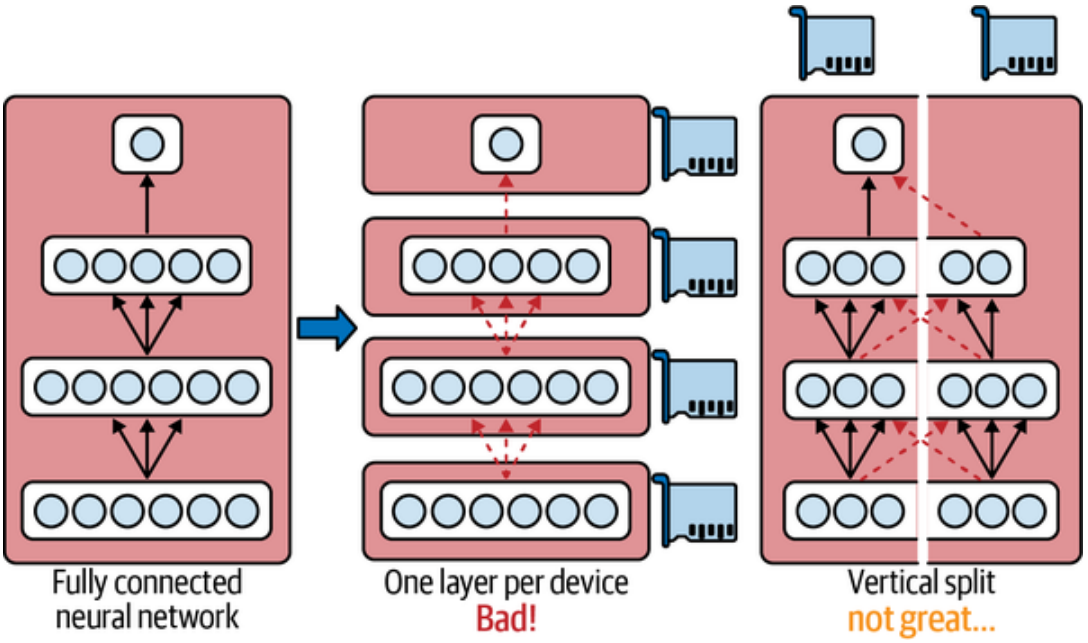

Figure 19-11. Splitting a fully connected neural network

Some neural network architectures, such as convolutional neural networks (see Chapter 14), contain layers that are only partially connected to the lower layers, so it is much easier to distribute chunks across devices in an efficient way (Figure 19-12).

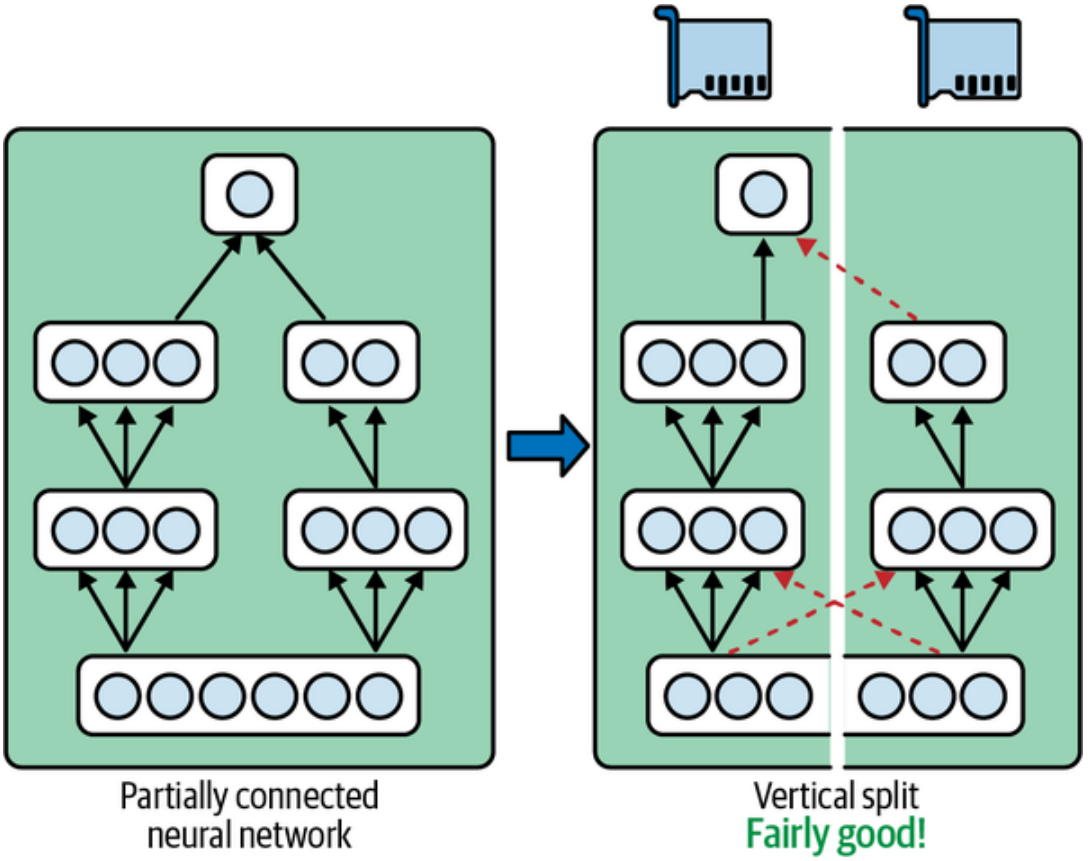

Figure 19-12. Splitting a partially connected neural network

Deep recurrent neural networks (see Chapter 15) can be split a bit more efficiently across multiple GPUs. If you split the network horizontally by placing each layer on a different device, and feed the network with an input sequence to process, then at the first time step only one device will be active (working on the sequence's first value), at the second step two will be active (the second layer will be handling the output of the first layer for the first value, while the first layer will be handling the second value), and by the time the signal propagates to the output layer, all devices will be active simultaneously (Figure 19-13). There is still a lot of cross-device communication going on, but since each cell may be fairly complex, the benefit of running multiple cells in parallel may (in theory) outweigh the communication penalty. However, in practice a regular stack of `LSTM` layers running on a single GPU actually runs much faster.

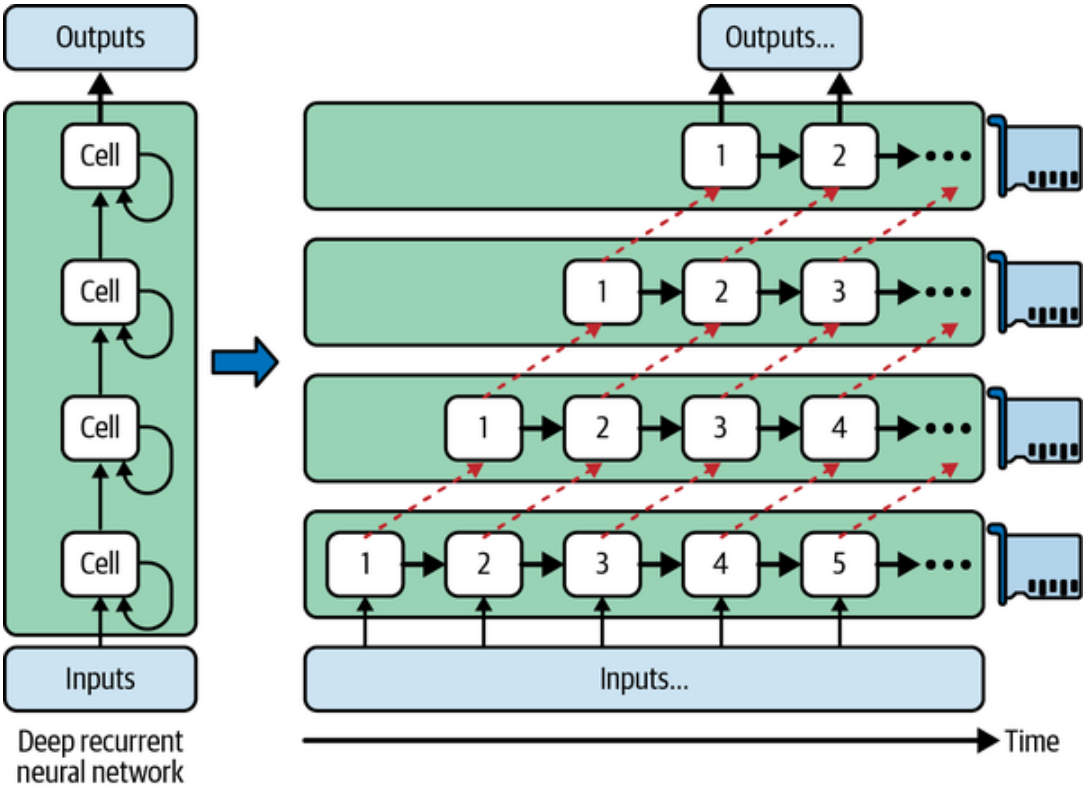

Figure 19-13. Splitting a deep recurrent neural network

In short, model parallelism may speed up running or training some types of neural networks, but not all, and it requires special care and tuning, such as making sure that devices that need to communicate the most run on the same machine.[15] Next we'll look at a much simpler and generally more efficient option: data parallelism.

Data Parallelism

Another way to parallelize the training of a neural network is to replicate it on every device and run each training step simultaneously on all replicas, using a different mini-batch for each. The gradients computed by each replica are then averaged, and the result is used to update the model parameters. This is called *data parallelism*, or sometimes *single program, multiple data* (SPMD). There are many variants of this idea, so let's look at the most important ones.

Data parallelism using the mirrored strategy

Arguably the simplest approach is to completely mirror all the model parameters across all the GPUs and always apply the exact same parameter updates on every GPU. This way, all replicas always remain perfectly identical. This is called the *mirrored strategy*, and it turns out to be quite efficient, especially when using a single machine (see Figure 19-14).

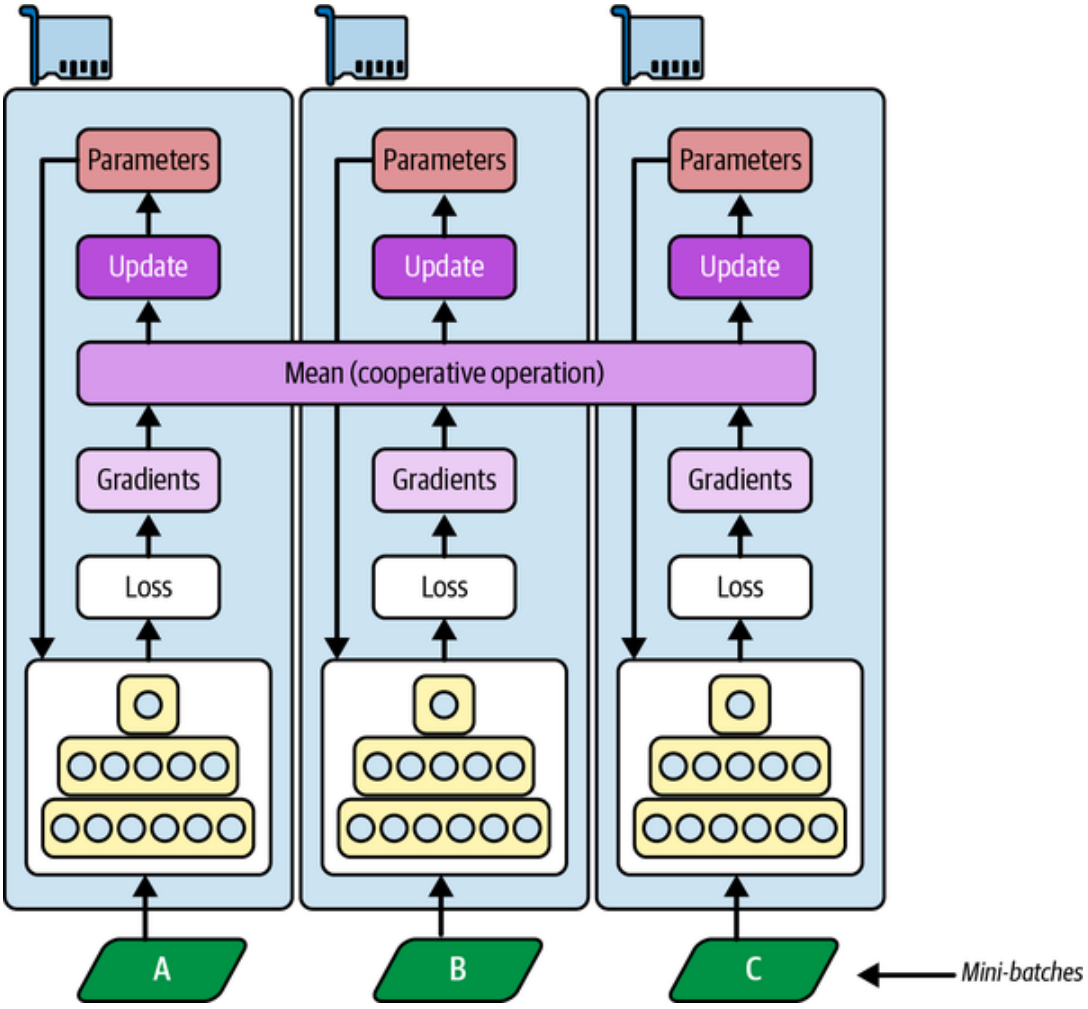

Figure 19-14. Data parallelism using the mirrored strategy

The tricky part when using this approach is to efficiently compute the mean of all the gradients from all the GPUs and distribute the result across all the GPUs. This can be done using an *AllReduce* algorithm, a class of algorithms where multiple nodes collaborate to efficiently perform a *reduce operation* (such as computing the mean, sum, and max), while ensuring that all nodes obtain the same final result. Fortunately, there are off-the-shelf implementations of such algorithms, as you will see.

Data parallelism with centralized parameters

Another approach is to store the model parameters outside of the GPU devices performing the computations (called *workers*); for example, on the CPU (see Figure 19-15). In a distributed setup, you may place all the parameters on one or more CPU-only servers called *parameter servers*, whose only role is to host and update the parameters.

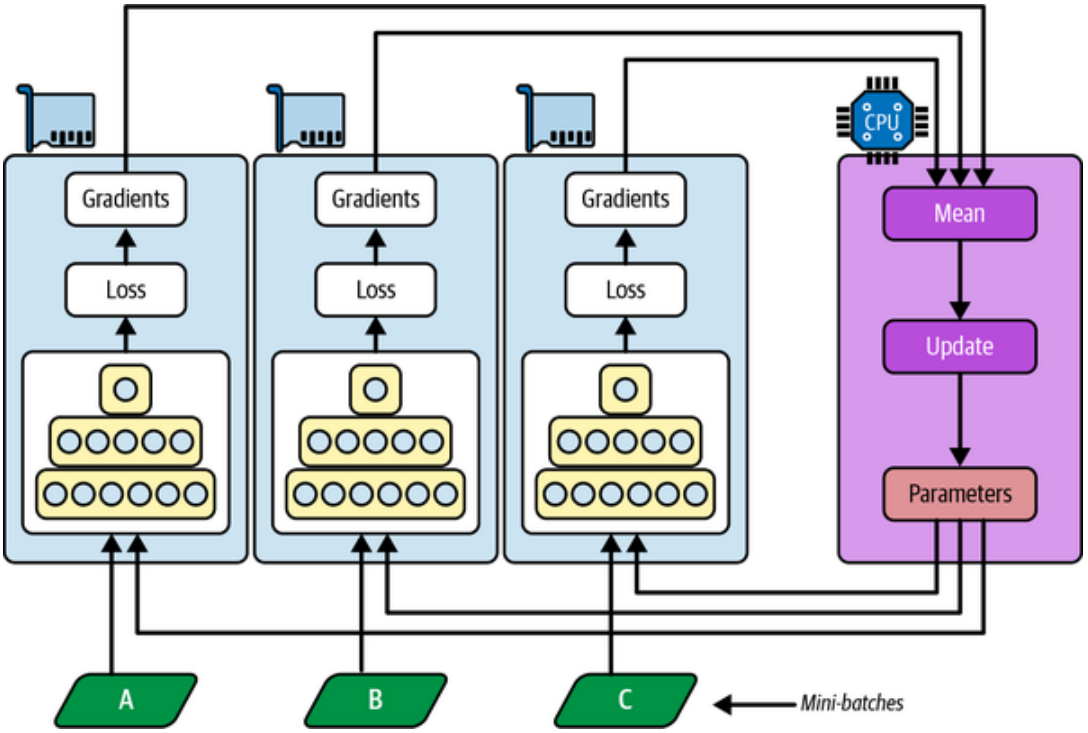

Figure 19-15. Data parallelism with centralized parameters

Whereas the mirrored strategy imposes synchronous weight updates across all GPUs, this centralized approach allows either synchronous or asynchronous updates. Let's take a look at the pros and cons of both options.

Synchronous updates

With *synchronous updates*, the aggregator waits until all gradients are available before it computes the average gradients and passes them to the optimizer, which will update the model parameters. Once a replica has finished computing its gradients, it must wait for the parameters to be updated before it can proceed to the next mini-batch. The downside is that some devices may be slower than others, so the fast devices will have to wait for the slow ones at every step, making the whole process as slow as the slowest device. Moreover, the parameters will be copied to every device almost at the same time (immediately after the gradients are applied), which may saturate the parameter servers' bandwidth.

TIP

To reduce the waiting time at each step, you could ignore the gradients from the slowest few replicas (typically ~10%). For example, you could run 20 replicas, but only aggregate the gradients from the fastest 18 replicas at each step, and just ignore the gradients from the last 2. As soon as the parameters are updated, the first 18 replicas can start working again immediately, without having to wait for the 2 slowest replicas. This setup is generally described as having 18 replicas plus 2 *spare replicas*.[16]

Asynchronous updates

With asynchronous updates, whenever a replica has finished computing the gradients, the gradients are immediately used to update the model parameters. There is no aggregation (it removes the "mean" step in Figure 19-15) and no synchronization. Replicas work independently of the other replicas. Since there is no waiting for the other replicas, this approach runs more training steps per minute. Moreover, although the parameters still need to be copied to every device at every step, this happens at different times for each replica, so the risk of bandwidth saturation is reduced.

Data parallelism with asynchronous updates is an attractive choice because of its simplicity, the absence of synchronization delay, and its better use of the bandwidth. However, although it works reasonably well in practice, it is almost surprising that it works at all! Indeed, by the time a replica has finished computing the gradients based on some parameter values, these parameters will have been updated several times by other replicas (on average $N - 1$ times, if there are N replicas), and there is no guarantee that the computed gradients will still be pointing in the right direction (see Figure 19-16). When gradients are severely out of date, they are called *stale gradients*: they can slow down convergence, introducing noise and wobble effects (the learning curve may contain temporary oscillations), or they can even make the training algorithm diverge.

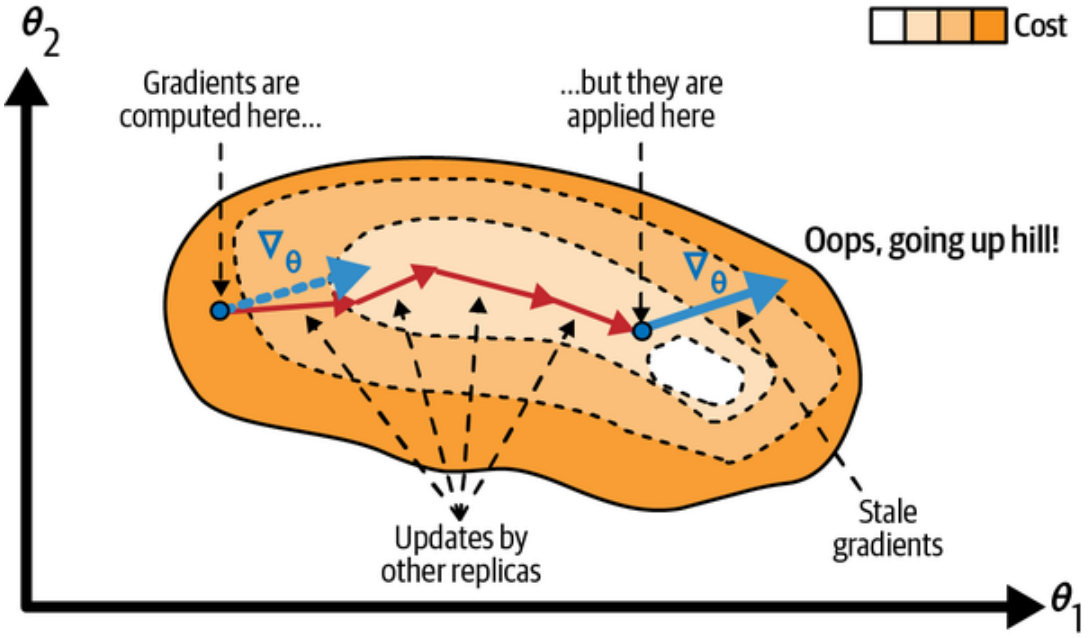

Figure 19-16. Stale gradients when using asynchronous updates

There are a few ways you can reduce the effect of stale gradients:

- Reduce the learning rate.
- Drop stale gradients or scale them down.
- Adjust the mini-batch size.
- Start the first few epochs using just one replica (this is called the *warmup phase*). Stale gradients tend to be more damaging at the beginning of training, when gradients are typically large and the parameters have not settled into a valley of the cost function yet, so different replicas may push the parameters in quite different directions.

A paper published by the Google Brain team in 2016[17] benchmarked various approaches and found that using synchronous updates with a few spare replicas was more efficient than using asynchronous updates, not only converging faster but also producing a better model. However, this is still an active area of research, so you should not rule out asynchronous updates just yet.

Bandwidth saturation

Whether you use synchronous or asynchronous updates, data parallelism with centralized parameters still requires communicating the model parameters from the parameter servers to every replica at the beginning of each training step, and the gradients in the other direction at the end of each training step. Similarly, when using the mirrored strategy, the gradients produced by each GPU will need to be shared with every other GPU.

Unfortunately, there often comes a point where adding an extra GPU will not improve performance at all because the time spent moving the data into and out of GPU RAM (and across the network in a distributed setup) will outweigh the speedup obtained by splitting the computation load. At that point, adding more GPUs will just worsen the bandwidth saturation and actually slow down training.

Saturation is more severe for large dense models, since they have a lot of parameters and gradients to transfer. It is less severe for small models (but the parallelization gain is limited) and for large sparse models, where the gradients are typically mostly zeros and so can be communicated efficiently. Jeff Dean, initiator and lead of the Google Brain project, reported typical speedups of 25–40× when distributing computations across 50 GPUs for dense models, and a 300× speedup for sparser models trained across 500 GPUs. As you can see, sparse models really do scale better. Here are a few concrete examples:

- Neural machine translation: 6× speedup on 8 GPUs
- Inception/ImageNet: 32× speedup on 50 GPUs
- RankBrain: 300× speedup on 500 GPUs

There is plenty of research going on to alleviate the bandwidth saturation issue, with the goal of allowing training to scale linearly with the number of GPUs available. For example, a 2018 paper[18] by a team of researchers from Carnegie Mellon University, Stanford University, and Microsoft Research proposed a system called *PipeDream* that managed to reduce network communications by over 90%, making it possible to train large models across many machines. They achieved this using a new technique called *pipeline parallelism*, which combines model parallelism and data parallelism: the model is chopped into consecutive parts, called *stages*, each of which is trained on a different machine. This results in an asynchronous pipeline in which all machines work in parallel with very little idle time. During training, each stage alternates one round of forward propagation and one round of backpropagation (see Figure 19-17): it pulls a mini-batch from its input queue, processes it, and sends the outputs to the next stage's input queue, then it pulls one mini-batch of gradients from its gradient queue, backpropagates these gradients and updates its own model parameters, and pushes the backpropagated gradients to the previous stage's gradient queue. It then repeats the whole process again and again. Each stage can also use regular data parallelism (e.g., using the mirrored strategy), independently from the other stages.

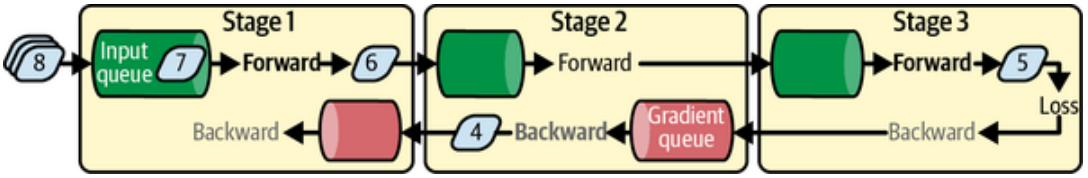

Figure 19-17. PipeDream's pipeline parallelism

However, as it's presented here, PipeDream would not work so well. To understand why, consider mini-batch #5 in Figure 19-17: when it went through stage 1 during the forward pass, the gradients from mini-batch #4 had not yet been backpropagated through that stage, but by the time #5's gradients flow back to stage 1, #4's gradients will have been used to update the model parameters, so #5's gradients will be a bit stale. As we have seen, this can degrade training speed and accuracy, and even make it diverge: the more stages there are, the worse this problem becomes. The paper's authors proposed methods to mitigate this issue, though: for example, each stage saves weights during forward propagation and restores them during backpropagation, to ensure that the same weights are used for both the forward pass and the backward pass. This is called *weight stashing*. Thanks to this, PipeDream demonstrates impressive scaling capability, well beyond simple data parallelism.

The latest breakthrough in this field of research was published in a 2022 paper[19] by Google researchers: they developed a system called *Pathways* that uses automated model parallelism, asynchronous gang scheduling, and other techniques to reach close to 100% hardware utilization across thousands of TPUs! *Scheduling* means organizing when and where each task must run, and *gang scheduling* means running related tasks at the same time in parallel and close to each other to reduce the time tasks have to wait for the others' outputs. As we saw in Chapter 16, this system was used to train a massive language model across over 6,000 TPUs, with close to 100% hardware utilization: that's a mindblowing engineering feat.

At the time of writing, Pathways is not public yet, but it's likely that in the near future you will be able to train huge models on Vertex AI using Pathways or a similar system. In the meantime, to reduce the saturation problem, you'll probably want to use a few powerful GPUs rather than plenty of weak GPUs, and if you need to train a model across multiple servers, you should group your GPUs on few and very well interconnected servers. You can also try dropping the float precision from 32 bits (`tf.float32`) to 16 bits (`tf.bfloat16`). This will cut in half the amount of data to transfer, often without much impact on the convergence rate or

the model's performance. Lastly, if you are using centralized parameters, you can shard (split) the parameters across multiple parameter servers: adding more parameter servers will reduce the network load on each server and limit the risk of bandwidth saturation.

OK, now that we've gone through all the theory, let's actually train a model across multiple GPUs!

Training at Scale Using the Distribution Strategies API

Luckily, TensorFlow comes with a very nice API that takes care of all the complexity of distributing your model across multiple devices and machines: the *distribution strategies API*. To train a Keras model across all available GPUs (on a single machine, for now) using data parallelism with the mirrored strategy, just create a `MirroredStrategy` object, call its `scope()` method to get a distribution context, and wrap the creation and compilation of your model inside that context. Then call the model's `fit()` method normally:

```
strategy = tf.distribute.MirroredStrategy()

with strategy.scope():
    model = tf.keras.Sequential([...])  # create a Keras model normally
    model.compile([...])  # compile the model normally

batch_size = 100  # preferably divisible by the number of replicas
model.fit(X_train, y_train, epochs=10,
          validation_data=(X_valid, y_valid), batch_size=batch_size)
```

Under the hood, Keras is distribution-aware, so in this `MirroredStrategy` context it knows that it must replicate all variables and operations across all available GPU devices. If you look at the model's weights, they are of type `MirroredVariable`:

```
>>> type(model.weights[0])
tensorflow.python.distribute.values.MirroredVariable
```

Note that the `fit()` method will automatically split each training batch across all the replicas, so it's preferable to ensure that the batch size is divisible by the number of replicas (i.e., the number of available GPUs) so

that all replicas get batches of the same size. And that's all! Training will generally be significantly faster than using a single device, and the code change was really minimal.

Once you have finished training your model, you can use it to make predictions efficiently: call the `predict()` method, and it will automatically split the batch across all replicas, making predictions in parallel. Again, the batch size must be divisible by the number of replicas. If you call the model's `save()` method, it will be saved as a regular model, *not* as a mirrored model with multiple replicas. So when you load it, it will run like a regular model, on a single device: by default on GPU #0, or on the CPU if there are no GPUs. If you want to load a model and run it on all available devices, you must call `tf.keras.models.load_model()` within a distribution context:

```
with strategy.scope():
    model = tf.keras.models.load_model("my_mirrored_model")
```

If you only want to use a subset of all the available GPU devices, you can pass the list to the `MirroredStrategy` 's constructor:

```
strategy = tf.distribute.MirroredStrategy(devices=["/gpu:0", "/gpu:1"])
```

By default, the `MirroredStrategy` class uses the *NVIDIA Collective Communications Library* (NCCL) for the AllReduce mean operation, but you can change it by setting the `cross_device_ops` argument to an instance of the `tf.distribute.HierarchicalCopyAllReduce` class, or an instance of the `tf.distribute.ReductionToOneDevice` class. The default NCCL option is based on the `tf.distribute.NcclAllReduce` class, which is usually faster, but this depends on the number and types of GPUs, so you may want to give the alternatives a try.[20]

If you want to try using data parallelism with centralized parameters, replace the `MirroredStrategy` with the `CentralStorageStrategy` :

```
strategy = tf.distribute.experimental.CentralStorageStrategy()
```

You can optionally set the `compute_devices` argument to specify the list of devices you want to use as workers—by default it will use all available GPUs—and you can optionally set the `parameter_device` argument to

specify the device you want to store the parameters on. By default it will use the CPU, or the GPU if there is just one.

Now let's see how to train a model across a cluster of TensorFlow servers!

Training a Model on a TensorFlow Cluster

A *TensorFlow cluster* is a group of TensorFlow processes running in parallel, usually on different machines, and talking to each other to complete some work—for example, training or executing a neural network model. Each TF process in the cluster is called a *task*, or a *TF server*. It has an IP address, a port, and a type (also called its *role* or its *job*). The type can be either `"worker"`, `"chief"`, `"ps"` (parameter server), or `"evaluator"`:

- Each *worker* performs computations, usually on a machine with one or more GPUs.
- The *chief* performs computations as well (it is a worker), but it also handles extra work such as writing TensorBoard logs or saving checkpoints. There is a single chief in a cluster. If no chief is specified explicitly, then by convention the first worker is the chief.
- A *parameter server* only keeps track of variable values, and it is usually on a CPU-only machine. This type of task is only used with the `ParameterServerStrategy`.
- An *evaluator* obviously takes care of evaluation. This type is not used often, and when it's used, there's usually just one evaluator.

To start a TensorFlow cluster, you must first define its specification. This means defining each task's IP address, TCP port, and type. For example, the following *cluster specification* defines a cluster with three tasks (two workers and one parameter server; see Figure 19-18). The cluster spec is a dictionary with one key per job, and the values are lists of task addresses (*IP*:*port*):

```
cluster_spec = {
    "worker": [
        "machine-a.example.com:2222",     # /job:worker/task:0
        "machine-b.example.com:2222"      # /job:worker/task:1
    ],
    "ps": ["machine-a.example.com:2221"]  # /job:ps/task:0
}
```

In general there will be a single task per machine, but as this example shows, you can configure multiple tasks on the same machine if you want. In this case, if they share the same GPUs, make sure the RAM is split appropriately, as discussed earlier.

WARNING

By default, every task in the cluster may communicate with every other task, so make sure to configure your firewall to authorize all communications between these machines on these ports (it's usually simpler if you use the same port on every machine).

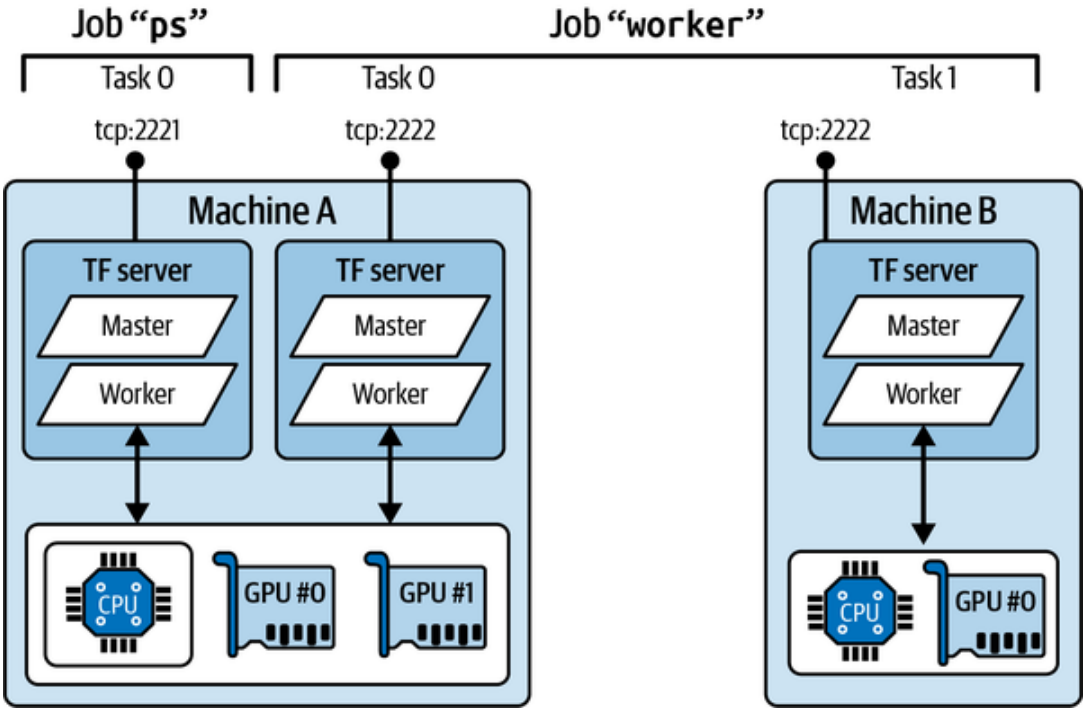

Figure 19-18. An example TensorFlow cluster

When you start a task, you must give it the cluster spec, and you must also tell it what its type and index are (e.g., worker #0). The simplest way to specify everything at once (both the cluster spec and the current task's type and index) is to set the `TF_CONFIG` environment variable before starting TensorFlow. It must be a JSON-encoded dictionary containing a cluster specification (under the `"cluster"` key) and the type and index of the current task (under the `"task"` key). For example, the following `TF_CONFIG` environment variable uses the cluster we just defined and specifies that the task to start is worker #0:

```
os.environ["TF_CONFIG"] = json.dumps({
    "cluster": cluster_spec,
```

```
    "task": {"type": "worker", "index": 0}
})
```

TIP

In general you want to define the `TF_CONFIG` environment variable outside of Python, so the code does not need to include the current task's type and index (this makes it possible to use the same code across all workers).

Now let's train a model on a cluster! We will start with the mirrored strategy. First, you need to set the `TF_CONFIG` environment variable appropriately for each task. There should be no parameter server (remove the `"ps"` key in the cluster spec), and in general you will want a single worker per machine. Make extra sure you set a different task index for each task. Finally, run the following script on every worker:

```
import tempfile
import tensorflow as tf

strategy = tf.distribute.MultiWorkerMirroredStrategy()  # at the start!
resolver = tf.distribute.cluster_resolver.TFConfigClusterResolver()
print(f"Starting task {resolver.task_type} #{resolver.task_id}")
[...] # load and split the MNIST dataset

with strategy.scope():
    model = tf.keras.Sequential([...])  # build the Keras model
    model.compile([...])  # compile the model

model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10)

if resolver.task_id == 0:  # the chief saves the model to the right location
    model.save("my_mnist_multiworker_model", save_format="tf")
else:
    tmpdir = tempfile.mkdtemp()  # other workers save to a temporary directory
    model.save(tmpdir, save_format="tf")
    tf.io.gfile.rmtree(tmpdir)  # and we can delete this directory at the end!
```

That's almost the same code you used earlier, except this time you are using the `MultiWorkerMirroredStrategy` . When you start this script on the first workers, they will remain blocked at the AllReduce step, but training will begin as soon as the last worker starts up, and you will see them all advancing at exactly the same rate since they synchronize at each step.

WARNING

When using the `MultiWorkerMirroredStrategy`, it's important to ensure that all workers do the same thing, including saving model checkpoints or writing TensorBoard logs, even though you will only keep what the chief writes. This is because these operations may need to run the AllReduce operations, so all workers must be in sync.

There are two AllReduce implementations for this distribution strategy: a ring AllReduce algorithm based on gRPC for the network communications, and NCCL's implementation. The best algorithm to use depends on the number of workers, the number and types of GPUs, and the network. By default, TensorFlow will apply some heuristics to select the right algorithm for you, but you can force NCCL (or RING) like this:

```
strategy = tf.distribute.MultiWorkerMirroredStrategy(
    communication_options=tf.distribute.experimental.CommunicationOptions(
        implementation=tf.distribute.experimental.CollectiveCommunication.NCCL
```

If you prefer to implement asynchronous data parallelism with parameter servers, change the strategy to `ParameterServerStrategy`, add one or more parameter servers, and configure `TF_CONFIG` appropriately for each task. Note that although the workers will work asynchronously, the replicas on each worker will work synchronously.

Lastly, if you have access to TPUs on Google Cloud—for example, if you use Colab and you set the accelerator type to TPU—then you can create a `TPUStrategy` like this:

```
resolver = tf.distribute.cluster_resolver.TPUClusterResolver()
tf.tpu.experimental.initialize_tpu_system(resolver)
strategy = tf.distribute.experimental.TPUStrategy(resolver)
```

This needs to be run right after importing TensorFlow. You can then use this strategy normally.

TIP

If you are a researcher, you may be eligible to use TPUs for free; see *https://tensorflow.org/tfrc* for more details.

You can now train models across multiple GPUs and multiple servers: give yourself a pat on the back! If you want to train a very large model, however, you will need many GPUs, across many servers, which will require either buying a lot of hardware or managing a lot of cloud virtual machines. In many cases, it's less hassle and less expensive to use a cloud service that takes care of provisioning and managing all this infrastructure for you, just when you need it. Let's see how to do that using Vertex AI.

Running Large Training Jobs on Vertex AI

Vertex AI allows you to create custom training jobs with your own training code. In fact, you can use almost the same training code as you would use on your own TF cluster. The main thing you must change is where the chief should save the model, the checkpoints, and the TensorBoard logs. Instead of saving the model to a local directory, the chief must save it to GCS, using the path provided by Vertex AI in the `AIP_MODEL_DIR` environment variable. For the model checkpoints and TensorBoard logs, you should use the paths contained in the `AIP_CHECKPOINT_DIR` and `AIP_TENSORBOARD_LOG_DIR` environment variables, respectively. Of course, you must also make sure that the training data can be accessed from the virtual machines, such as on GCS, or another GCP service like BigQuery, or directly from the web. Lastly, Vertex AI sets the `"chief"` task type explicitly, so you should identify the chief using `resolved.task_type == "chief"` instead of `resolved.task_id == 0`:

```
import os
[...]  # other imports, create MultiWorkerMirroredStrategy, and resolver

if resolver.task_type == "chief":
    model_dir = os.getenv("AIP_MODEL_DIR")  # paths provided by Vertex AI
    tensorboard_log_dir = os.getenv("AIP_TENSORBOARD_LOG_DIR")
    checkpoint_dir = os.getenv("AIP_CHECKPOINT_DIR")
else:
    tmp_dir = Path(tempfile.mkdtemp())  # other workers use temporary dirs
    model_dir = tmp_dir / "model"
    tensorboard_log_dir = tmp_dir / "logs"
    checkpoint_dir = tmp_dir / "ckpt"

callbacks = [tf.keras.callbacks.TensorBoard(tensorboard_log_dir),
             tf.keras.callbacks.ModelCheckpoint(checkpoint_dir)]
[...]  # build and  compile using the strategy scope, just like earlier
model.fit(X_train, y_train, validation_data=(X_valid, y_valid), epochs=10,
```

```
            callbacks=callbacks)
model.save(model_dir, save_format="tf")
```

TIP

If you place the training data on GCS, you can create a `tf.data.TextLineDataset` or `tf.data.TFRecordDataset` to access it: just use the GCS paths as the filenames (e.g., *gs://my_bucket/data/001.csv*). These datasets rely on the `tf.io.gfile` package to access files: it supports both local files and GCS files.

Now you can create a custom training job on Vertex AI, based on this script. You'll need to specify the job name, the path to your training script, the Docker image to use for training, the one to use for predictions (after training), any additional Python libraries you may need, and lastly the bucket that Vertex AI should use as a staging directory to store the training script. By default, that's also where the training script will save the trained model, as well as the TensorBoard logs and model checkpoints (if any). Let's create the job:

```
custom_training_job = aiplatform.CustomTrainingJob(
    display_name="my_custom_training_job",
    script_path="my_vertex_ai_training_task.py",
    container_uri="gcr.io/cloud-aiplatform/training/tf-gpu.2-4:latest",
    model_serving_container_image_uri=server_image,
    requirements=["gcsfs==2022.3.0"],  # not needed, this is just an example
    staging_bucket=f"gs://{bucket_name}/staging"
)
```

And now let's run it on two workers, each with two GPUs:

```
mnist_model2 = custom_training_job.run(
    machine_type="n1-standard-4",
    replica_count=2,
    accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=2,
)
```

And that's it: Vertex AI will provision the compute nodes you requested (within your quotas), and it will run your training script across them. Once the job is complete, the `run()` method will return a trained model that you can use exactly like the one you created earlier: you can deploy it

to an endpoint, or use it to make batch predictions. If anything goes wrong during training, you can view the logs in the GCP console: in the ☰ navigation menu, select Vertex AI → Training, click on your training job, and click VIEW LOGS. Alternatively, you can click the CUSTOM JOBS tab and copy the job's ID (e.g., 1234), then select Logging from the ☰ navigation menu and query `resource.labels.job_id=1234`.

TIP

To visualize the training progress, just start TensorBoard and point its `--logdir` to the GCS path of the logs. It will use *application default credentials*, which you can set up using `gcloud auth application-default login`. Vertex AI also offers hosted TensorBoard servers if you prefer.

If you want to try out a few hyperparameter values, one option is to run multiple jobs. You can pass the hyperparameter values to your script as command-line arguments by setting the `args` parameter when calling the `run()` method, or you can pass them as environment variables using the `environment_variables` parameter.

However, if you want to run a large hyperparameter tuning job on the cloud, a much better option is to use Vertex AI's hyperparameter tuning service. Let's see how.

Hyperparameter Tuning on Vertex AI

Vertex AI's hyperparameter tuning service is based on a Bayesian optimization algorithm, capable of quickly finding optimal combinations of hyperparameters. To use it, you first need to create a training script that accepts hyperparameter values as command-line arguments. For example, your script could use the `argparse` standard library like this:

```
import argparse

parser = argparse.ArgumentParser()
parser.add_argument("--n_hidden", type=int, default=2)
parser.add_argument("--n_neurons", type=int, default=256)
parser.add_argument("--learning_rate", type=float, default=1e-2)
parser.add_argument("--optimizer", default="adam")
args = parser.parse_args()
```

The hyperparameter tuning service will call your script multiple times, each time with different hyperparameter values: each run is called a *trial*, and the set of trials is called a *study*. Your training script must then use the given hyperparameter values to build and compile a model. You can use a mirrored distribution strategy if you want, in case each trial runs on a multi-GPU machine. Then the script can load the dataset and train the model. For example:

```
import tensorflow as tf

def build_model(args):
    with tf.distribute.MirroredStrategy().scope():
        model = tf.keras.Sequential()
        model.add(tf.keras.layers.Flatten(input_shape=[28, 28], dtype=tf.uint8
        for _ in range(args.n_hidden):
            model.add(tf.keras.layers.Dense(args.n_neurons, activation="relu")
        model.add(tf.keras.layers.Dense(10, activation="softmax"))
        opt = tf.keras.optimizers.get(args.optimizer)
        opt.learning_rate = args.learning_rate
        model.compile(loss="sparse_categorical_crossentropy", optimizer=opt,
                      metrics=["accuracy"])
        return model

[...]  # load the dataset
model = build_model(args)
history = model.fit([...])
```

TIP

You can use the `AIP_*` environment variables we mentioned earlier to determine where to save the checkpoints, the TensorBoard logs, and the final model.

Lastly, the script must report the model's performance back to Vertex AI's hyperparameter tuning service, so it can decide which hyperparameters to try next. For this, you must use the `hypertune` library, which is automatically installed on Vertex AI training VMs:

```
import hypertune

hypertune = hypertune.HyperTune()
hypertune.report_hyperparameter_tuning_metric(
    hyperparameter_metric_tag="accuracy",  # name of the reported metric
    metric_value=max(history.history["val_accuracy"]),  # metric value
```

```
        global_step=model.optimizer.iterations.numpy(),
    )
```

Now that your training script is ready, you need to define the type of machine you would like to run it on. For this, you must define a custom job, which Vertex AI will use as a template for each trial:

```
trial_job = aiplatform.CustomJob.from_local_script(
    display_name="my_search_trial_job",
    script_path="my_vertex_ai_trial.py",  # path to your training script
    container_uri="gcr.io/cloud-aiplatform/training/tf-gpu.2-4:latest",
    staging_bucket=f"gs://{bucket_name}/staging",
    accelerator_type="NVIDIA_TESLA_K80",
    accelerator_count=2,  # in this example, each trial will have 2 GPUs
)
```

Finally, you're ready to create and run the hyperparameter tuning job:

```
from google.cloud.aiplatform import hyperparameter_tuning as hpt

hp_job = aiplatform.HyperparameterTuningJob(
    display_name="my_hp_search_job",
    custom_job=trial_job,
    metric_spec={"accuracy": "maximize"},
    parameter_spec={
        "learning_rate": hpt.DoubleParameterSpec(min=1e-3, max=10, scale="log"
        "n_neurons": hpt.IntegerParameterSpec(min=1, max=300, scale="linear"),
        "n_hidden": hpt.IntegerParameterSpec(min=1, max=10, scale="linear"),
        "optimizer": hpt.CategoricalParameterSpec(["sgd", "adam"]),
    },
    max_trial_count=100,
    parallel_trial_count=20,
)
hp_job.run()
```

Here, we tell Vertex AI to maximize the metric named `"accuracy"`: this name must match the name of the metric reported by the training script. We also define the search space, using a log scale for the learning rate and a linear (i.e., uniform) scale for the other hyperparameters. The hyperparameter names must match the command-line arguments of the training script. Then we set the maximum number of trials to 100, and the maximum number of trials running in parallel to 20. If you increase the number of parallel trials to (say) 60, the total search time will be re-

duced significantly, by a factor of up to 3. But the first 60 trials will be started in parallel, so they will not benefit from the other trials' feedback. Therefore, you should increase the max number of trials to compensate—for example, up to about 140.

This will take quite a while. Once the job is completed, you can fetch the trial results using `hp_job.trials`. Each trial result is represented as a protobuf object, containing the hyperparameter values and the resulting metrics. Let's find the best trial:

```
def get_final_metric(trial, metric_id):
    for metric in trial.final_measurement.metrics:
        if metric.metric_id == metric_id:
            return metric.value

trials = hp_job.trials
trial_accuracies = [get_final_metric(trial, "accuracy") for trial in trials]
best_trial = trials[np.argmax(trial_accuracies)]
```

Now let's look at this trial's accuracy, and its hyperparameter values:

```
>>> max(trial_accuracies)
0.977400004863739
>>> best_trial.id
'98'
>>> best_trial.parameters
[parameter_id: "learning_rate" value { number_value: 0.001 },
 parameter_id: "n_hidden" value { number_value: 8.0 },
 parameter_id: "n_neurons" value { number_value: 216.0 },
 parameter_id: "optimizer" value { string_value: "adam" }
]
```

That's it! Now you can get this trial's SavedModel, optionally train it a bit more, and deploy it to production.

TIP

Vertex AI also includes an AutoML service, which completely takes care of finding the right model architecture and training it for you. All you need to do is upload your dataset to Vertex AI using a special format that depends on the type of dataset (images, text, tabular, video, etc.), then create an AutoML training job, pointing to the dataset and specifying the maximum number of compute hours you're willing to spend. See the notebook for an example.

HYPERPARAMETER TUNING USING KERAS TUNER ON VERTEX AI

Instead of using Vertex AI's hyperparameter tuning service, you can use Keras Tuner (introduced in Chapter 10) and run it on Vertex AI VMs. Keras Tuner provides a simple way to scale hyperparameter search by distributing it across multiple machines: it only requires setting three environment variables on each machine, then running your regular Keras Tuner code on each machine. You can use the exact same script on all machines. One of the machines acts as the chief (i.e., the oracle), and the others act as workers. Each worker asks the chief which hyperparameter values to try, then the worker trains the model using these hyperparameter values, and finally it reports the model's performance back to the chief, which can then decide which hyperparameter values the worker should try next.

The three environment variables you need to set on each machine are:

KERASTUNER_TUNER_ID

: This is equal to `"chief"` on the chief machine, or a unique identifier on each worker machine, such as `"worker0"`, `"worker1"`, etc.

KERASTUNER_ORACLE_IP

: This is the IP address or hostname of the chief machine. The chief itself should generally use `"0.0.0.0"` to listen on every IP address on the machine.

KERASTUNER_ORACLE_PORT

: This is the TCP port that the chief will be listening on.

You can distribute Keras Tuner across any set of machines. If you want to run it on Vertex AI machines, then you can spawn a regular training job, and just modify the training script to set the environment variables properly before using Keras Tuner. See the notebook for an example.

Now you have all the tools and knowledge you need to create state-of-the-art neural net architectures and train them at scale using various distribution strategies, on your own infrastructure or on the cloud, and then deploy them anywhere. In other words, you now have superpowers: use them well!

Exercises

1. What does a SavedModel contain? How do you inspect its content?
2. When should you use TF Serving? What are its main features? What are some tools you can use to deploy it?
3. How do you deploy a model across multiple TF Serving instances?
4. When should you use the gRPC API rather than the REST API to query a model served by TF Serving?
5. What are the different ways TFLite reduces a model's size to make it run on a mobile or embedded device?
6. What is quantization-aware training, and why would you need it?
7. What are model parallelism and data parallelism? Why is the latter generally recommended?
8. When training a model across multiple servers, what distribution strategies can you use? How do you choose which one to use?
9. Train a model (any model you like) and deploy it to TF Serving or Google Vertex AI. Write the client code to query it using the REST API or the gRPC API. Update the model and deploy the new version. Your client code will now query the new version. Roll back to the first version.
10. Train any model across multiple GPUs on the same machine using the `MirroredStrategy` (if you do not have access to GPUs, you can use Google Colab with a GPU runtime and create two logical GPUs). Train the model again using the `CentralStorageStrategy` and compare the training time.
11. Fine-tune a model of your choice on Vertex AI, using either Keras Tuner or Vertex AI's hyperparameter tuning service.

Solutions to these exercises are available at the end of this chapter's notebook, at *https://homl.info/colab3*.

Thank You!

Before we close the last chapter of this book, I would like to thank you for reading it up to the last paragraph. I truly hope that you had as much fun reading this book as I had writing it, and that it will be useful for your projects, big or small.

If you find errors, please send feedback. More generally, I would love to know what you think, so please don't hesitate to contact me via O'Reilly,

through the *ageron/handson-ml3* GitHub project, or on Twitter at @aureliengeron.

Going forward, my best advice to you is to practice and practice: try going through all the exercises (if you have not done so already), play with the notebooks, join Kaggle or some other ML community, watch ML courses, read papers, attend conferences, and meet experts. Things move fast, so try to keep up to date. Several YouTube channels regularly present deep learning papers in great detail, in a very approachable way. I particularly recommend the channels by Yannic Kilcher, Letitia Parcalabescu, and Xander Steenbrugge. For fascinating ML discussions and higher-level insights, make sure to check out ML Street Talk, and Lex Fridman's channel. It also helps tremendously to have a concrete project to work on, whether it is for work or for fun (ideally for both), so if there's anything you have always dreamed of building, give it a shot! Work incrementally; don't shoot for the moon right away, but stay focused on your project and build it piece by piece. It will require patience and perseverance, but when you have a walking robot, or a working chatbot, or whatever else you fancy building, it will be immensely rewarding!

My greatest hope is that this book will inspire you to build a wonderful ML application that will benefit all of us. What will it be?

—*Aurélien Géron*

1 An A/B experiment consists in testing two different versions of your product on different subsets of users in order to check which version works best and get other insights.

2 Google AI Platform (formerly known as Google ML Engine) and Google AutoML merged in 2021 to form Google Vertex AI.

3 A REST (or RESTful) API is an API that uses standard HTTP verbs, such as GET, POST, PUT, and DELETE, and uses JSON inputs and outputs. The gRPC protocol is more complex but more efficient; data is exchanged using protocol buffers (see Chapter 13).

4 If you are not familiar with Docker, it allows you to easily download a set of applications packaged in a *Docker image* (including all their dependencies and usually some good default configuration) and then run them on your system using a *Docker engine*. When you run an image, the engine creates a *Docker container* that keeps the applications well isolated from your own system—but you can give it

some limited access if you want. It is similar to a virtual machine, but much faster and lighter, as the container relies directly on the host's kernel. This means that the image does not need to include or run its own kernel.

5 There are also GPU images available, and other installation options. For more details, please check out the official installation instructions.

6 To be fair, this can be mitigated by serializing the data first and encoding it to Base64 before creating the REST request. Moreover, REST requests can be compressed using gzip, which reduces the payload size significantly.

7 Also check out TensorFlow's Graph Transform Tool for modifying and optimizing computational graphs.

8 For example, a PWA must include icons of various sizes for different mobile devices, it must be served via HTTPS, it must include a manifest file containing metadata such as the name of the app and the background color.

9 Please check the TensorFlow docs for detailed and up-to-date installation instructions, as they change quite often.

10 As we saw in Chapter 12, a kernel is an operation's implementation for a specific data type and device type. For example, there is a GPU kernel for the `float32` `tf.matmul()` operation, but there is no GPU kernel for `int32` `tf.matmul()`, only a CPU kernel.

11 You can also use `tf.debugging.set_log_device_placement(True)` to log all device placements.

12 This can be useful if you want to guarantee perfect reproducibility, as I explain in this video, based on TF 1.

13 At the time of writing, it only prefetches the data to the CPU RAM, but use `tf.data.experimental.prefetch_to_device()` to make it prefetch the data and push it to the device of your choice so that the GPU does not waste time waiting for the data to be transferred.

14 If the two CNNs are identical, then it is called a *Siamese neural network*.

15 If you are interested in going further with model parallelism, check out Mesh TensorFlow.

16 This name is slightly confusing because it sounds like some replicas are special, doing nothing. In reality, all replicas are equivalent: they all work hard to be among the fastest at each training step, and the losers vary at every step (unless

some devices are really slower than others). However, it does mean that if one or two servers crash, training will continue just fine.

17 Jianmin Chen et al., "Revisiting Distributed Synchronous SGD", arXiv preprint arXiv:1604.00981 (2016).

18 Aaron Harlap et al., "PipeDream: Fast and Efficient Pipeline Parallel DNN Training", arXiv preprint arXiv:1806.03377 (2018).

19 Paul Barham et al., "Pathways: Asynchronous Distributed Dataflow for ML", arXiv preprint arXiv:2203.12533 (2022).

20 For more details on AllReduce algorithms, read Yuichiro Ueno's post on the technologies behind deep learning and Sylvain Jeaugey's post on massively scaling deep learning training with NCCL.