So far in this book, we have covered how to train and test different kinds of machine learning models using PyTorch. We started by reviewing the basic elements of PyTorch that enable us to work on deep learning tasks efficiently. Then, we explored a wide range of deep learning model architectures and applications that can be written using PyTorch.

In this chapter, we will be focusing on taking these models into production. But what does that mean? Basically, we will be discussing the different ways of taking a trained and tested model (object) into a separate environment where it can be used to make predictions or inferences on incoming data. This is what is referred to as the **productionization** of a model, as the model is being deployed into a production system.

We will begin by discussing some common approaches you can take to serve PyTorch models in production environments, starting from defining a simple model inference function and going all the way to using model microservices. We will then take a look at TorchServe, which is a scalable PyTorch model-serving framework that has been jointly developed by AWS and Facebook.

We will then dive into the world of exporting PyTorch models using **TorchScript**, which, through **serialization**, makes our models independent of the Python ecosystem so that they can be, for instance, loaded in a **C++** based environment . We will also look beyond the Torch framework and the Python ecosystem as we explore **ONNX** – an open source universal format for machine learning models – which will help us export PyTorch trained models to non-PyTorch and non-Pythonic environments.

Finally, we will briefly discuss how to use PyTorch for model serving with some of the well-known cloud platforms such as **Amazon Web Services** (**AWS**), **Google Cloud**, and **Microsoft Azure**.

Throughout this chapter, we will use the handwritten digits image classification **convolutional neural network** (**CNN**) model that we trained in *Chapter 1, Overview of Deep Learning Using PyTorch*, as our reference. We will demonstrate how that trained model can be deployed and exported using the different approaches discussed in this chapter.

This chapter is broken down into the following sections:

- Model serving in PyTorch
- Serving a PyTorch model using TorchServe
- Exporting universal PyTorch models using TorchScript and ONNX
- Serving PyTorch model in the cloud

Model serving in PyTorch

In this section, we will begin with building a simple PyTorch inference pipeline that can make predictions given some input data and the location of a previously trained and saved PyTorch model. We will proceed thereafter to place this inference pipeline on a model server that can listen to incoming data requests and return predictions. Finally, we will advance from developing a model server to creating a model microservice using Docker.

Creating a PyTorch model inference pipeline

We will be working on the handwritten digits image classification CNN model that we built in *Chapter 1*, *Overview of Deep Learning Using PyTorch*, on the `MNIST` dataset. Using this trained model, we will build an inference pipeline that shall be able to predict a digit between 0 to 9 for a given handwritten-digit input image.

For the process of building and training the model, please refer to the *Training a neural network using PyTorch* section of *Chapter 1, Overview of Deep Learning Using PyTorch*. For the full code of this exercise, you can refer to our github repository [13.1 .

Saving and loading a trained model

In this section, we will demonstrate how to efficiently load a saved pre-trained PyTorch model, which will later be used for serving requests.

So, using the notebook code from *Chapter 1, Overview of Deep Learning Using PyTorch*, we have trained a model and evaluated it against test data samples. But what next? In real life, we would like to close this notebook and, later on, still be able to use this model that we worked hard on training to make inferences on handwritten-digit images. This is where the concept of serving a model comes in.

From here, we will get into a position where we can use the preceding trained model in a separate Jupyter notebook without having to do any (re)training. The crucial next step is to save the model object into a file that can later be restored/de-serialized. PyTorch provides two main ways of doing this:

The less recommended way is to save the entire model object as follows:

```
torch.save(model, PATH_TO_MODEL)
```

And then, the saved model can be later read as follows:

```
model = torch.load(PATH_TO_MODEL)
```

Although this approach looks the most straightforward, this can be problematic in some cases. This is because we are not only saving the model parameters, but also the model classes and directory structure used in our source code. If our class signatures or directory structures change later, loading the model will fail in potentially unfixable ways.

The second and more recommended way is to only save the model parameters as follows:

```
torch.save(model.state_dict(), PATH_TO_MODEL)
```

Later, when we need to restore the model, first we instantiate an empty model object and then load the model parameters into that model object as follows:

```
model = ConvNet()
model.load_state_dict(torch.load(PATH_TO_MODEL))
```

We will use the more recommended way to save the model as shown in the following code:

```
PATH_TO_MODEL = "./convnet.pth"
torch.save(model.state_dict(), PATH_TO_MODEL)
```

The `convnet.pth` file is essentially a pickle file containing model parameters.

At this point, we can safely close the notebook we were working on and open another one, which is available at our github repository [13.2] :

1. As a first step, we will once again need to import libraries:

```
import torch
```

1. Next, we need to instantiate an empty CNN model once again. Ideally, the model definition done in *step 1* would be written in a Python script (say, `cnn_model.py`), and then we would simply need to write this:

```
from cnn_model import ConvNet
model = ConvNet()
```

However, since we are operating in Jupyter notebooks in this exercise, we shall rewrite the model definition and then instantiate it as follows:

```
class ConvNet(nn.Module):
    def __init__(self):
       …
    def forward(self, x):
        …
model = ConvNet()
```

1. We can now restore the saved model parameters into this instantiated model object as follows:

```
PATH_TO_MODEL = "./convnet.pth"
model.load_state_dict(torch.load(PATH_TO_MODEL, map_location="cpu"))
```

You shall see the following output:

```
<All keys matched successfully>
```

Figure 13 .1 – Model parameter loading

This essentially means that the parameter loading is successful. That is, the model that we have instantiated has the same structure as the model whose parameters were saved and are now being restored. We specify that we are loading the model on a CPU device as opposed to GPU (CUDA).

1. Finally, we want to specify that we do not wish to update or change the parameter values of the loaded model, and we will do so with the following line of code:

```
model.eval()
```

This should give the following output:

```
ConvNet(
  (cn1): Conv2d(1, 16, kernel_size=(3, 3), stride=(1, 1))
  (cn2): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1))
  (dp1): Dropout2d(p=0.1, inplace=False)
  (dp2): Dropout2d(p=0.25, inplace=False)
  (fc1): Linear(in_features=4608, out_features=64, bias=True)
  (fc2): Linear(in_features=64, out_features=10, bias=True)
)
```

Figure 13 .2 – Loaded model in evaluation mode

This again verifies that we are indeed working with the same model (architecture) that we trained.

Building the inference pipeline

Having successfully loaded a pre-trained model in a new environment (notebook) in the previous section, we shall now build our model inference pipeline and use it to run model predictions:

1. At this point, we have the previously trained model object fully restored to us. We shall now load an image that we can run the model prediction on using the following code:

```
image = Image.open("./digit_image.jpg")
```

The image file should be available in the exercise folder and is as follows:

Figure 13 .3 – Model inference input image

It is not necessary to use this particular image in the exercise. You may use any image you want, to check how the model reacts to it.

1. In any inference pipeline, there are three main components at the core of it: (a) the data preprocessing component, (b) the model inference (forward pass in the case of neural networks), and (c) the post-processing step.

We will begin with the first part by defining a function that takes in an image and transforms it into the tensor that shall be fed to the model as input as follows:

Copy Explain

```
def image_to_tensor(image):
    gray_image = transforms.functional.to_grayscale(image)
    resized_image = transforms.functional.resize(gray_image, (28, 28))
    input_image_tensor = transforms.functional.to_tensor(resized_image)
    input_image_tensor_norm = transforms.functional.normalize(input_image_tensor,
(0.1302,), (0.3069,))
    return input_image_tensor_norm
```

This can be seen as a series of steps as follows:

1. First, the RGB image is converted to a grayscale image.
2. The image is then resized to a 28x28 pixels image because this is the image size the model is trained with.
3. Then, the image array is converted to a PyTorch tensor.
4. And finally, the pixel values in the tensor are normalized with the same mean and standard deviation values as those used during model training time.

Having defined this function, we call it to convert our loaded image into a tensor:

Copy Explain

```
input_tensor = image_to_tensor(image)
```

1. Next, we define the **model inference functionality**. This is where the model takes in a tensor as input and outputs the predictions. In this case, the prediction will be any digit between 0 to 9 and the input tensor will be the tensorized form of the input image:

Copy Explain

```
def run_model(input_tensor):
    model_input = input_tensor.unsqueeze(0)
    with torch.no_grad():
        model_output = model(model_input)[0]
    model_prediction = model_output.detach().numpy().argmax()
    return model_prediction
```

`model_output` contains the raw predictions of the model, which contains a list of predictions for each image. Because we have only one image in the input, this list of predictions will just have one entry at index `0`. The raw prediction at index `0` is essentially a tensor with 10 probability values for digits 0,1,2...9, in that order. This tensor is converted to a `numpy` array, and finally, we choose the digit that has the highest probability.

1. We can now use this function to generate our model prediction. The following code uses the `run_model` model inference function from *step 3* to generate the model prediction for the given input data, `input_tensor`:

```
output = run_model(input_tensor)
print(output)
print(type(output))
```

This should output the following:

```
output = run_model(input_tensor)
print(output)
print(type(output))

2
<class 'numpy.int64'>
```

Figure 13 .4 – Model inference output

As we can see from the preceding screenshot, the model outputs a `numpy` integer. And based on the image shown in *Figure 13 .3*, the model output seems rather correct.

1. Besides just outputting the model prediction, we can also write a debug function to dig deeper into metrics such as raw prediction probabilities, as shown in the following code snippet:

```
def debug_model(input_tensor):
    model_input = input_tensor.unsqueeze(0)
    with torch.no_grad():
        model_output = model(model_input)[0]
    model_prediction = model_output.detach().numpy()
    return np.exp(model_prediction)
```

This function is exactly the same as the `run_model` function except that it returns the raw list of probabilities for each digit. The model originally returns the logarithm of softmax outputs because of the `log_softmax` layer being used as the final layer in the model (refer to *step 2* of this exercise).

Hence, we need to exponentiate those numbers to return the softmax outputs, which are equivalent to model prediction probabilities. Using this debug function, we can look at how the model is performing in more detail, such as whether the probability distribution is flat or has clear peaks:

```
print(debug_model(input_tensor))
```

This should produce an output similar to the following:

```
[8.69212745e-05 5.61913612e-06 9.97763395e-01 1.33050999e-04
 5.43686365e-05 1.59305739e-06 1.17863165e-04 5.08185963e-07
 1.83202932e-03 4.63086781e-06]
```

Figure 13 .5 – Model inference debug output

We can see that the third probability in the list is the highest by far, which corresponds to digit 2.

1. Finally, we shall post-process the model prediction so that it can be used by other applications. In our case, we are just going to transform the digit predicted by the model from the integer type to the string type.

The post-processing step can be more complex in other scenarios, such as speech recognition, where we might want to process the output waveform by smoothening, removing outliers, and so on:

```
def post_process(output):
    return str(output)
```

Because string is a serializable format, this enables the model predictions to be communicated easily across servers and applications. We can check whether our final post-processed data is as expected:

```
final_output = post_process(output)
print(final_output)
print(type(final_output))
```

This should provide you with the following output:

```
final_output = post_process(output)
print(final_output)
print(type(final_output))

2
<class 'str'>
```

Figure 13 .6 – Post-processed model prediction

As expected, the output is now of the `type` string.

This concludes our exercise of loading a saved model architecture, restoring its trained weights, and using the loaded model to generate predictions for sample input data (an image). We loaded a sample image, pre-processed it to transform it into a PyTorch tensor, passed it to the model as input to obtain the model prediction, and post-processed the prediction to generate the final output.

This is a step forward in the direction of serving trained models with a clearly defined input and output interface. In this exercise, the input was an externally provided image file and the output was a generated string containing a digit between 0 to 9. Such a system can be embedded by copying and pasting the provided code into any application that requires the functionality of digitizing hand-written digits.

In the next section, we will go a level deeper into model serving, where we aim to build a system that can be interacted with by any application to use the digitizing functionality without copying and pasting any code.

Building a basic model server

We have so far built a model inference pipeline that has all the code necessary to independently perform predictions from a pre-trained model. Here, we will work on building our first model server, which is essentially a machine that hosts the model inference pipeline, actively listens to any incoming input data via an interface, and outputs model predictions on any input data through the interface.

Writing a basic app using Flask

To develop our server, we will use a popular Python library – Flask [13.3]. **Flask** will enable us to build our model server in a few lines of code . A good example of how this library works is shown with the following code:

Copy Explain

```
from flask import Flask
app = Flask(__name__)
@app.route('/')
def hello_world():
    return 'Hello, World!'
if __name__ == '__main__':
    app.run(host='localhost', port=8890)
```

Say we saved this Python script as `example.py` and ran it from the terminal:

Copy Explain

```
python example.py
```

It would show the following output in the terminal:

```
* Serving Flask app "example" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://localhost:8890/ (Press CTRL+C to quit)
```

Figure 13 .7 – Flask example app launch

Basically, it will launch a Flask server that will serve an app called **example**. Let's open a browser and go to the following URL:

Copy Explain

```
http://localhost:8890/
```

It will result in the following output in the browser:

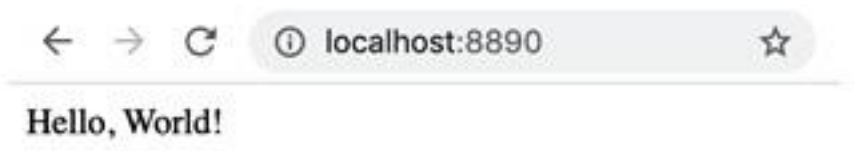

Figure 13 .8 – Flask example app testing

Essentially, the Flask server is listening to port number `8890` on the IP address `0.0.0.0 (localhost)` at the endpoint `/`. As soon as we input `localhost:8890/` in a browser search bar and press *Enter*, a request is received by this server. The server then runs the `hello_world` function, which in turn returns the string `Hello, World!` as per the function definition provided in `example.py`.

Using Flask to build our model server

Using the principles of running a Flask server demonstrated in the preceding section, we will now use the model inference pipeline built in the previous section to create our first model server. At the end of the exercise, we will launch the server that will be listening to incoming requests (image data input).

We will furthermore write another Python script that will make a request to this server by sending the sample image shown in *Figure 13 .3*. The Flask server shall run the model inference on this image and output the post-processed predictions.

The full code for this exercise is available on GitHub incuding the Flask server code [13.4] and the client (request-maker) code [13.5].

Setting up model inference for Flask serving

In this section, we will load a pre-trained model and write the model inference pipeline code:

1. First, we will build the Flask server. And for that, we once again start by importing the necessary libraries:

```
from flask import Flask, request
import torch
```

Both `flask` and `torch` are vital necessities for this task, besides other basic libraries such as `numpy` and `json`.

1. Next, we will need to define the model class (architecture):

```
class ConvNet(nn.Module):
    def __init__(self):
    def forward(self, x):
```

1. Now that we have the empty model class defined, we can instantiate a model object and load the pre-trained model parameters into this model object as follows:

```
model = ConvNet()
PATH_TO_MODEL = "./convnet.pth"
model.load_state_dict(torch.load(PATH_TO_MODEL, map_location="cpu"))
model.eval()
```

1. We will reuse the exact `run_model` function defined in *step 3* of the *Building the inference pipeline* section:

```
def run_model(input_tensor):
    …
    return model_prediction
```

As a reminder, this function takes in the tensorized input image and outputs the model prediction, which is any digit between 0 to 9.

1. Next, we will reuse the exact `post_process` function defined in *step 6* of the *Building the inference pipeline* section:

```
def post_process(output):
    return str(output)
```

This will essentially convert the integer output from the `run_model` function to a string.

Building a Flask app to serve model

Having established the inference pipeline in the previous section, we will now build our own Flask app and use it to serve the loaded model:

1. We will instantiate our Flask app as shown in the following line of code:

```
app = Flask(__name__)
```

This creates a Flask app with the same name as the Python script, which in our case is `server(.py)`.

1. This is the critical step, where we will be defining an endpoint functionality of the Flask server. We will expose a `/test` endpoint and define what happens when a `POST` request is made to that endpoint on the server as follows:

```
@app.route("/test", methods=["POST"])
def test():
    data = request.files['data'].read()
    md = json.load(request.files['metadata'])
    input_array = np.frombuffer(data, dtype=np.float32)
    input_image_tensor = torch.from_numpy(input_array).view(md["dims"])
    output = run_model(input_image_tensor)
    final_output = post_process(output)
    return final_output
```

Let's go through the steps one by one:

1. First, we add a decorator to the function – `test` – defined underneath. This decorator tells the Flask app to run this function whenever someone makes a `POST` request to the `/test` endpoint.
2. Next, we get to defining what exactly happens inside the `test` function. First, we read the data and metadata from the `POST` request. Because the data is in serialized form, we need to convert it into a numerical format – we convert it to a `numpy` array. And from a `numpy` array, we swiftly cast it as a PyTorch tensor.
3. Next, we use the image dimensions provided in the metadata to reshape the tensor.
4. Finally, we run a forward pass of the model loaded earlier with this tensor. This gives us the model prediction, which is then post-processed and returned by our test function.

1. We have all the necessary ingredients to launch our Flask app. We will add these last two lines to our `server.py` Python script:

```
if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8890)
```

This indicates that the Flask server will be hosted at IP address `0.0.0.0` (also known as `localhost`) and port number `8890`. We may now save the Python script and in a new terminal window simply execute the following:

```
python server.py
```

This will run the entire script written in the previous steps and you shall see the following output:

```
* Serving Flask app "server" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:8890/ (Press CTRL+C to quit)
```

Figure 13 .9 – Flask server launch

This looks similar to the example demonstrated in *Figure 13 .7*. The only difference is the app name.

Using a Flask server to run predictions

We have successfully launched our model server, which is actively listening to requests. Let's now work on making a request:

1. We will write a separate Python script in the next few steps to do this job. We begin with importing libraries:

```
import requests
from PIL import Image
from torchvision import transforms
```

The `requests` library will help us make the actual `POST` request to the Flask server. `Image` helps us to read a sample input image file, and `transforms` will help us to preprocess the input image array.

1. Next, we read an image file:

```
image = Image.open("./digit_image.jpg")
```

The image read here is an RGB image and may have any dimensions (not necessarily 28x28 as expected by the model as input).

1. We now define a preprocessing function that converts the read image into a format that is readable by the model:

```
def image_to_tensor(image):
    gray_image = transforms.functional.to_grayscale(image)
    resized_image = transforms.functional.resize(gray_image, (28, 28))
    input_image_tensor = transforms.functional.to_tensor(resized_image)
    input_image_tensor_norm = transforms.functional.normalize(input_image_tensor,
(0.1302,), (0.3069,))
    return input_image_tensor_norm
```

Having defined the function, we can execute it:

```
image_tensor = image_to_tensor(image)
```

`image_tensor` is what we need to send as input data to the Flask server.

1. Let's now get into packaging our data together to send it over. We want to send both the pixel values of the image as well as the shape of the image (28x28) so that the Flask server at the receiving end knows how to reconstruct the stream of pixel values as an image:

```
dimensions = io.StringIO(json.dumps({'dims': list(image_tensor.shape)}))
data = io.BytesIO(bytearray(image_tensor.numpy()))
```

We stringify the shape of our tensor and convert the image array into bytes to make it all serializable.

1. This is the most critical step in this client code . This is where we actually make the `POST` request:

```
r = requests.post('http://localhost:8890/test',
                  files={'metadata': dimensions,                    'data' :
data})
```

Using the `requests` library, we make the `POST` request at the URL `localhost:8890/test`. This is where the Flask server is listening for requests. We send both the actual image data (as bytes) and the metadata (as string) in the form of a dictionary

1. The `r` variable in the preceding code will receive the response of the request from the Flask server. This response should contain the post-processed model prediction. We will now read that output:

```
response = json.loads(r.content)
```

The `response` variable will essentially contain what the Flask server outputs, which is a digit between 0 and 9 as a string.

1. We can print the response just to be sure:

```
print("Predicted digit :", response)
```

At this point, we can save this Python script as `make_request.py` and execute the following command in the terminal:

```
python make_request.py
```

This should output the following:

```
Predicted digit : 2
```

Figure 13 .10 – Flask server response

Based on the input image (see *Figure 13 .3*), the response seems rather correct. This concludes our current exercise.

Thus, we have successfully built a standalone model server that can render predictions for handwritten digit images. The same set of steps can easily be extended to any other machine learning model, and so this opens up endless possibilities with regards to creating machine learning applications using PyTorch and Flask.

So far, we have moved from simply writing inference functions to creating model servers that can be hosted remotely and render predictions over the network. In our next and final model serving venture, we will go a level further. You might have noticed that in order to follow the steps in the previous two exercises, there were inherent dependencies to be considered. We are required to install certain libraries, save and load the models at particular locations, read image data, and so on. All of these manual steps slow down the development of a model server.

Up next, we will work on creating a model microservice that can be spun up with one command and replicated across several machines .

Creating a model microservice

Imagine you know nothing about training machine learning models but want to use an already-trained model without having to get your hands dirty with any PyTorch code. This is where a paradigm such as the machine learning model microservice [13.6] comes into play.

A machine learning model microservice can be thought of as a black box to which you send input data and it sends back predictions to you. Moreover, it is easy to spin up this black box on a given machine with just a few lines of code. The best part is that it scales effortlessly. You can scale a microservice vertically by using a bigger machine (more memory, more processing power) as well as horizontally, by replicating the microservice across multiple machines.

How do we go about deploying a machine learning model as a microservice? Thanks to the work done using Flask and PyTorch in the previous exercise, we are already a few steps ahead. We have already built a standalone model server using Flask.

In this section, we will take that idea forward and build a standalone model-serving environment using **Docker**. Docker helps containerize software, which essentially means that it helps virtualize the entire **operating system** (**OS**), including software libraries, configuration files, and even data files.

Note
Docker is a huge topic of discussion in itself. However, because the book is focused on PyTorch, we will only cover the basic concepts and usage of Docker for our limited purposes. If you are interested in reading about Docker further, their own documentation is a great place to start [13.7] .

In our case, we have so far used the following libraries in building our model server:

- Python
- PyTorch
- Pillow (for image I/O)
- Flask

And, we have used the following data file:

- Pre-trained model checkpoint file (`convnet.pth`)

We have had to manually arrange for these dependencies by installing the libraries and placing the file in the current working directory. What if we have to redo all of this in a new machine? We would have to manually install the libraries and copy and paste the file once again. This way of working is neither efficient nor failproof, as we might end up installing different library versions across different machines, for example.

To solve this problem, we would like to create an OS-level blueprint that can be consistently repeated across machines. This is where Docker comes in handy. Docker lets us create that blueprint in the form of a Docker image. This image can then be built on any empty machine with no assumptions regarding pre-installed Python libraries or an already-available model.

Let's actually create such a blueprint using Docker for our digits classification model. In the form of an exercise, we will go from a Flask-based standalone model server to a Docker-based model microservice. Before delving into the exercise, you will need to install Docker [13.8] :

1. First, we need to list the Python library requirements for our Flask model server. The requirements (with their versions) are as follows:

```
torch==1.5.0
torchvision==0.5.0
Pillow==6.2.2
Flask==1.1.1
```

As a general practice, we will save this list as a text file – `requirements.txt`. This file is also available in our github repository [13.9] . This list will come in handy for installing the libraries consistently in any given environment.

1. Next, we get straight to the blueprint, which, in Docker terms, will be the `Dockerfile`. A `Dockerfile` is a script that is essentially a list of instructions. The machine where this `Dockerfile` is run needs to execute the listed instructions in the file. This results in a Docker image, and the process is called *building an image*.

An **image** here is a system snapshot that can be effectuated on any machine, provided that the machine has the minimum necessary hardware resources (for example, installing PyTorch alone requires multiple GBs of disk space).

Let's look at our `Dockerfile` and try to understand what it does step by step. The full code for the `Dockerfile` is available in our guthub repository [13.10]

.

1. The `FROM` keyword instructs Docker to fetch a standard Linux OS with `python 3.8` baked in:

```
FROM python:3.8-slim
```

This ensures that we will have Python installed.

1. Next, install `wget`, which is a Unix command useful for downloading resources from the internet via the command line:

```
RUN apt-get -q update && apt-get -q install -y wget
```

The && symbol indicates the sequential execution of commands written before and after the symbol.

1. Here, we are copying two files from our local development environment into this virtual environment:

```
COPY ./server.py ./
COPY ./requirements.txt ./
```

We copy the requirements file as discussed in *step 1* as well as the Flask model server code that we worked on in the previous exercise.

1. Next, we download the pre-trained PyTorch model checkpoint file:

```
RUN wget -q
https://github.com/arj7192/MasteringPyTorchV2/raw/main/Chapter13/convnet.pth
```

This is the same model checkpoint file that we had saved in the *Saving and loading a trained model* section of this chapter.

1. Here, we are installing all the relevant libraries listed under `requirements.txt`:

```
RUN pip install -r requirements.txt
```

This `txt` file is the one we wrote under *step 1*.

1. Next, we give `root` access to the Docker client:

```
USER root
```

This step is important in this exercise as it ensures that the client has the credentials to perform all necessary operations on our behalf, such as saving model inference logs on the disk.

Note

In general, though, it is advised not to give root privileges to the client as per the principle of least privilege in data security [13.11] .

1. Finally, we specify that after performing all the previous steps, Docker should execute the `python server.py` command:

```
ENTRYPOINT ["python", "server.py"]
```

This will ensure the launch of a Flask model server in the virtual machine.

1. Let's now run this Dockerfile. In other words, let's build a Docker image using the Dockerfile from *step 2*. In the current working directory, on the command line, simply run this:

```
docker build -t digit_recognizer .
```

We are allocating a tag with the name `digit_recognizer` to our Docker image. This should output the following:

```
Sending build context to Docker daemon  7.283MB
Step 1/9 : FROM python:3.8-slim
 ---> 62297c9f4e5c
Step 2/9 : RUN apt-get -q update && apt-get -q install -y wget
 ---> Using cache
 ---> e6142d540652
Step 3/9 : COPY ./server.py ./
 ---> Using cache
 ---> cb82fb5cb2e5
Step 4/9 : COPY ./requirements.txt ./
 ---> Using cache
 ---> faa39c98f044
Step 5/9 : RUN wget -q https://github.com/PacktPublishing/Mastering-PyTorch/raw/master/Chapter10/convnet.pth
 ---> Running in 198679553bac
Removing intermediate container 198679553bac
 ---> 8bddc82ccf1e
Step 6/9 : RUN wget -q https://github.com/PacktPublishing/Mastering-PyTorch/raw/master/Chapter10/digit_image.jpg
 ---> Running in 34836205c03d
Removing intermediate container 34836205c03d
 ---> 331c8447524a
Step 7/9 : RUN pip install -r requirements.txt
 ---> Running in e4bd0692812b
Collecting torch==1.5.0
  Downloading torch-1.5.0-cp38-cp38-manylinux1_x86_64.whl (752.0 MB)
Collecting torchvision==0.6.0
  Downloading torchvision-0.6.0-cp38-cp38-manylinux1_x86_64.whl (6.6 MB)
Collecting Pillow==6.2.2
  Downloading Pillow-6.2.2-cp38-cp38-manylinux1_x86_64.whl (2.1 MB)
Collecting Flask==1.1.1
  Downloading Flask-1.1.1-py2.py3-none-any.whl (94 kB)
Collecting numpy
  Downloading numpy-1.19.5-cp38-cp38-manylinux2010_x86_64.whl (14.9 MB)
Collecting future
  Downloading future-0.18.2.tar.gz (829 kB)
Collecting Werkzeug>=0.15
  Downloading Werkzeug-1.0.1-py2.py3-none-any.whl (298 kB)
Collecting itsdangerous>=0.24
  Downloading itsdangerous-1.1.0-py2.py3-none-any.whl (16 kB)
Collecting Jinja2>=2.10.1
  Downloading Jinja2-2.11.2-py2.py3-none-any.whl (125 kB)
Collecting click>=5.1
  Downloading click-7.1.2-py2.py3-none-any.whl (82 kB)
Collecting MarkupSafe>=0.23
  Downloading MarkupSafe-1.1.1-cp38-cp38-manylinux1_x86_64.whl (32 kB)
Building wheels for collected packages: future
  Building wheel for future (setup.py): started
  Building wheel for future (setup.py): finished with status 'done'
  Created wheel for future: filename=future-0.18.2-py3-none-any.whl size=491059 sha256=726027831b4159f39c497dee6280cf48539056b1b80d80fbf113330e7ea5af0d
  Stored in directory: /root/.cache/pip/wheels/8e/70/28/3d6ccd6e315f65f245da085482a2e1c7d14b90b30f239e2cf4
Successfully built future
Installing collected packages: numpy, future, torch, Pillow, torchvision, Werkzeug, itsdangerous, MarkupSafe, Jinja2, click, Flask
Successfully installed Flask-1.1.1 Jinja2-2.11.2 MarkupSafe-1.1.1 Pillow-6.2.2 Werkzeug-1.0.1 click-7.1.2 future-0.18.2 itsdangerous-1.1.0 numpy-1.19.5 torch-1.5.0 torchvision-0.6.0
WARNING: You are using pip version 20.2.3; however, version 20.3.3 is available.
You should consider upgrading via the '/usr/local/bin/python -m pip install --upgrade pip' command.
Removing intermediate container e4bd0692812b
 ---> d7ecf4681868
Step 8/9 : USER root
 ---> Running in 90dac164d1bc
Removing intermediate container 90dac164d1bc
 ---> e4488fab0902
Step 9/9 : ENTRYPOINT ["python", "server.py"]
 ---> Running in a1d55f33a99f
Removing intermediate container a1d55f33a99f
 ---> 2de7d75aae1b
Successfully built 2de7d75aae1b
Successfully tagged digit_recognizer:latest
```

Figure 13 .11 – Building a Docker image

Figure 13 .11 shows the sequential execution of the steps mentioned in *step 2*. Running this step might take a while, depending on your internet connection, as it downloads the entire PyTorch library among others to build the image.

1. At this stage, we already have a Docker image with the name `digit_recognizer`. We are all set to deploy this image on any machine. In order to deploy the image on your own machine for now, just run the following command:

```
docker run -p 8890:8890 digit_recognizer
```

Copy Explain

With this command, we are essentially starting a virtual machine inside our machine using the `digit_recognizer` Docker image. Because our original Flask model server was designed to listen to port `8890`, we have forwarded our actual machine's port `8890` to the virtual machine's port `8890` using the `-p` argument. Running this command should output this:

```
* Serving Flask app "server" (lazy loading)
* Environment: production
  WARNING: This is a development server. Do not use it in a production deployment.
  Use a production WSGI server instead.
* Debug mode: off
* Running on http://0.0.0.0:8890/ (Press CTRL+C to quit)
```

Figure 13 .12 – Running a Docker instance

The preceding screenshot is remarkably similar to *Figure 13 .9* from the previous exercise, which is no surprise because the Docker instance is running the same Flask model server that we were manually running in our previous exercise.

1. We can now test whether our Dockerized Flask model server (model microservice) works as expected by using it to make model predictions. We will once again use the `make_request.py` file used in the previous exercise to send a prediction request to our model. From the current local working directory, simply execute this:

```
python make_request.py
```

This should output the following:

```
Predicted digit : 2
```

Figure 13 .13 – Microservice model prediction

The microservice seems to be doing the job, and thus we have successfully built and tested our own machine learning model microservice using Python, PyTorch, Flask, and Docker.

1. Upon successful completion of the preceding steps, you can close the launched Docker instance from *step 4* by pressing *Ctrl+C* as indicated in *Figure 13 .12*. And once the running Docker instance is stopped, you can delete the instance by running the following command:

```
docker rm $(docker ps -a -q | head -1)
```

This command basically removes the most recent inactive Docker instance, which in our case is the Docker instance that we just stopped.

1. Finally, you can also delete the Docker image that we had built under *step 3*, by running the following command:

```
docker rmi $(docker images -q "digit_recognizer")
```

Copy Explain

This will basically remove the image that has been tagged with the `digit_recognizer` tag.

This concludes our section for serving models written in PyTorch. We started off by designing a local model inference system. We took this inference system and wrapped a Flask-based model server around it to create a standalone model serving system.

Finally, we used the Flask-based model server inside a Docker container to essentially create a model serving microservice. Using both the theory as well as the exercises discussed in this section, you should be able to get started with hosting/serving your trained models across different use cases, system configurations, and environments.

In the next section, we will stay with the model-serving theme but will discuss a particular tool that has been developed precisely to serve PyTorch models: **TorchServe**. We will also do a quick exercise to demonstrate how to use this tool.

Serving a PyTorch model using TorchServe

TorchServe, released in April 2020, is a dedicated PyTorch model-serving framework. Using the functionalities offered by TorchServe, we can serve multiple models at the same time with low prediction latency and without having to write much custom code. Furthermore, TorchServe offers features such as model versioning, metrics monitoring, and data preprocessing and post-processing.

This clearly makes TorchServe a more advanced model-serving alternative than the model microservice we developed in the previous section. However, making custom model microservices still proves to be a powerful solution for complicated machine learning pipelines (which is more common than we might think).

In this section, we will continue working with our handwritten digits classification model and demonstrate how to serve it using TorchServe. After reading this section, you should be able to get started with TorchServe and go further in utilizing its full set of features.

Installing TorchServe

Before starting with the exercise, we will need to install Java 11 SDK as a requirement.

For Linux OS, run the following:

```
sudo apt-get install openjdk-11-jdk
```

And for macOS, we need to run the following command on the command line:

```
brew tap AdoptOpenJDK/openjdk
brew      install --cask adoptopenjdk11
```

And thereafter, we need to install `torchserve` by running this:

```
pip install torchserve==0.6.0 torch-model-archiver==0.6.0
```

For detailed installation instructions, refer to torchserve documentation [13.12] .

Notice that we also install a library called `torch-model-archiver` [13.13]. This archiver aims at creating one model file that will contain both the model parameters as well as the model architecture definition in an independent serialized format as a `.mar` file.

Launching and using a TorchServe server

Now that we have installed all that we need, we can start putting together our existing code from the previous exercises to serve our model using TorchServe. We will hereon go through a number of steps in the form of an exercise:

1. First, we will place the existing model architecture code in a model file saved as `convnet.py`:

```
==========convnet.py==========
import torch
import torch.nn as nn
import torch.nn.functional as F
class ConvNet(nn.Module):
    def __init__(self):
        …
    def forward(self, x):
        …
```

We will need this model file as one of the inputs to `torch-model-archiver` to produce a unified `.mar` file. You can find the full model file in our github repository [13.14] .

Remember we had discussed the three parts of any model inference pipeline: data pre-processing, model prediction, and post-processing. TorchServe provides *handlers*, which handle the pre-processing and post-processing parts of popular kinds of machine learning tasks: `image_classifier`, `image_segmenter`, `object_detector`, and `text_classifier`.

This list might grow in the future as TorchServe is actively being developed at the time of writing this book.

1. For our task, we will create a custom image handler that is inherited from the default `Image_classifier` handler. We choose to create a custom handler because as opposed to the usual image classification models that deal with color (RGB) images, our model deals with grayscale images of a specific size (28x28 pixels). The following is the code for our custom handler, which you can also find in our github repository [13.15] :

```
==========convnet_handler.py==========
from torchvision import transforms
from ts.torch_handler.image_classifier import ImageClassifier
class ConvNetClassifier(ImageClassifier):
    image_processing = transforms.Compose([
        transforms.Grayscale(), transforms.Resize((28, 28)),
        transforms.ToTensor(),  transforms.Normalize((0.1302,), (0.3069,))])
    def postprocess(self, output):
        return output.argmax(1).tolist()
```

First, we imported the `image_classifer` default handler, which will provide most of the basic image classification inference pipeline handling capabilities. Next, we inherit the `ImageClassifer` handler class to define our custom `ConvNetClassifier` handler class.

There are two blocks of custom code:

1. The data pre-processing step, where we apply a sequence of transformations to the data exactly as we did in *step 3* of the *Building the inference pipeline* section.
2. The postprocessing step, defined under the `postprocess` method, where we extract the predicted class label from the list of prediction probabilities of all classes

1. We already produced a `convnet.pth` file in *the Saving and loading a trained model section* of this chapter while creating the model inference pipeline. Using `convnet.py`, `convnet_handler.py`, and `convnet.pth`, we can finally create the `.mar` file using `torch-model-archiver` by running the following command:

```
torch-model-archiver --model-name convnet --version 1.0 --model-file ./convnet.py --
serialized-file ./convnet.pth --handler  ./convnet_handler.py
```

This command should result in a `convnet.mar` file written to the current working directory. We have specified a `model_name` argument, which names the `.mar` file. We have specified a `version` argument, which will be helpful in model versioning while working with multiple variations of a model at the same time.

We have located where our `convnet.py` (for model architecture), `convnet.pth` (for model weights) and `convnet_handler.py` (for pre- and post-processing) files are, using the `model_file`, `serialzed_file`, and `handler` arguments, respectively.

1. Next, we need to create a new directory in the current working directory and move the `convnet.mar` file created in *step 3* to that directory, by running the following on the command line:

```
mkdir model_store
mv convnet.mar model_store/
```

We have to do so to follow the design requirements of the TorchServe framework.

1. Finally, we may launch our model server using TorchServe. On the command line, simply run the following:

```
torchserve --start --ncs --model-store model_store --models convnet.mar
```

This will silently start the model inference server and you will see some logs on the screen, including the following:

```
Number of GPUs: 0
Number of CPUs: 8
Max heap size: 4096 M
Python executable: /Users/ashish.jha/opt/anaconda3/bin/python
Config file: N/A
Inference address: http://127.0.0.1:8080
Management address: http://127.0.0.1:8081
Metrics address: http://127.0.0.1:8082
```

Figure 13 .14 – TorchServe launch output

As you can see, TorchServe investigates the available devices on the machine among other details. It allocates three separate URLs for *inference*, *management*, and *metrics*. To check whether the launched server is indeed serving our model, we can ping the management server with the following command:

```
curl http://localhost:8081/models
```

This should output the following:

```
{
  "models": [
    {
      "modelName": "convnet",
      "modelUrl": "convnet.mar"
    }
  ]
}
```

Figure 13 .15 – TorchServe-served models

This verifies that the TorchServe server is indeed hosting the model.

1. Finally, we can test our TorchServe model server by making an inference request. This time, we won't need to write a Python script, because the handler will already take care of processing any input image file. So, we can directly make a request using the `digit_image.jpg` sample image file by running this:

```
curl http://127.0.0.1:8080/predictions/convnet -T ./digit_image.jpg
```

This should output 2 in the terminal, which is indeed the correct prediction as evident from *Figure 13 .3*.

1. Finally, once we are done with using the model server, it can be stopped by running the following on the command line:

```
torchserve --stop
```

This concludes our exercise on how to use TorchServe to spin up our own PyTorch model server and use it to make predictions. There is a lot more to unpack here, such as model monitoring (metrics), logging, versioning, benchmarking, and so on [13.16]. TorchServe's website is a great place to pursue these advanced topics in detail.

After finishing this section, you should be able to use TorchServe to serve your own models. I encourage you to write custom handlers for your own use cases, explore the various TorchServe configuration settings [13.17] , and try out other advanced features of TorchServe [13.18] .

Note
TorchServe is constantlt evolving , with a lot of promise. My advice would be to keep an eye on the rapid updates in this territory of PyTorch.

In the next section, we will take a look at exporting PyTorch models so that they can be used in different environments, programming languages, and deep learning libraries.

Exporting universal PyTorch models using TorchScript and ONNX

We have discussed serving PyTorch models extensively in the previous sections of this chapter, which is perhaps the most critical aspect of operationalizing PyTorch models in production systems. In this section, we will look at another important aspect – exporting PyTorch models. We have already learned how to save PyTorch models and load them back from disk in the classic Python scripting environment. But we need more ways of exporting PyTorch models. Why?

Well, for starters, the Python interpreter allows only one thread to run at a time using the **global interpreter lock** (**GIL**). This keeps us from parallelizing operations. Secondly, Python might not be supported in every system or device that we might want to run our models on. To address these problems, PyTorch offers support for exporting its models in an efficient format and in a platform- or language-agnostic manner such that a model can be run in environments different from the one it was trained in.

We will first explore TorchScript, which enables us to export serialized and optimized PyTorch models into an intermediate representation that can then be run in a Python-independent program (say, a C++ program).

And then , we will look at ONNX and how it lets us save PyTorch models into a universal format that can then be loaded into other deep learning frameworks and different programming languages.

Understanding the utility of TorchScript

There are two key reasons why TorchScript is a vital tool when it comes to putting PyTorch models into production:

- PyTorch works on an eager execution basis, as discussed in *Chapter 1, Overview of Deep Learning Using PyTorch*, of this book. This has its advantages, such as easier debugging. However, executing steps/operations one by one by writing and reading intermediate results to and from memory may lead to high inference latency as well as limiting us from overall operational optimizations. To tackle this problem, PyTorch provides its own **just-in-time** (**JIT**) compiler, which is based on the PyTorch-centered parts of Python.

The JIT compiler compiles PyTorch models instead of interpreting, which is equivalent to creating one composite graph for the entire model by looking at all of its operations at once. The JIT-compiled code is TorchScript code, which is basically a

statically typed subset of Python. This compilation leads to several performance improvements and optimizations, such as getting rid of the GIL and thereby enabling multithreading.

> PyTorch is essentially built to be used with the Python programming language. Remember, we have used Python in almost the entirety of this book too. However, when it comes to productionizing models, there are more performant (that is, quicker) languages than Python, such as C++. And also, we might want to deploy our trained models on systems or devices that do not work with Python.

This is where TorchScript kicks in. As soon as we compile our PyTorch code into TorchScript code, which is an intermediate representation of our PyTorch model, we can serialize this representation into a C++-friendly format using the TorchScript compiler. Thereafter, this serialized file can be read in a C++ model inference program using LibTorch – the PyTorch C++ API.

We have mentioned JIT compilation of PyTorch models several times in this section. Let's now look at two of the possible options of compiling our PyTorch models into TorchScript format.

Model tracing with TorchScript

One way of translating PyTorch code to TorchScript is tracing the PyTorch model. Tracing requires the PyTorch model object along with a dummy example input to the model. As the name suggests, the tracing mechanism traces the flow of this dummy input through the model (neural network), records the various operations, and renders a TorchScript **Intermediate Representation** (IR), which can be visualized both as a graph as well as TorchScript code.

We will now walk through the steps involved in tracing a PyTorch model using our handwritten digits classification model. The full code for this exercise is available in our github repository [13.19] .

The first five steps of this exercise are the same as the steps of the *Saving and loading a trained model* and *Building the inference pipeline* sections, where we built the model inference pipeline:

1. We will start with importing libraries by running the following code:

Copy Explain

```
import torch
...
```

1. Next, we will define and instantiate the `model` object:

Copy Explain

```
class ConvNet(nn.Module):
    def __init__(self):
       …
    def forward(self, x):
        …
model = ConvNet()
```

1. Next, we will restore the model weights using the following lines of code:

Copy Explain

```
PATH_TO_MODEL = "./convnet.pth"
model.load_state_dict(torch.load(PATH_TO_MODEL, map_location="cpu"))
model.eval()
```

1. We then load a sample image:

Copy Explain

```
image = Image.open("./digit_image.jpg")
```

1. Next, we define the data pre-processing function:

Copy Explain

```
def image_to_tensor(image):
    gray_image = transforms.functional.to_grayscale(image)
    resized_image = transforms.functional.resize(gray_image, (28, 28))
    input_image_tensor = transforms.functional.to_tensor(resized_image)
    input_image_tensor_norm = transforms.functional.normalize(input_image_tensor,
(0.1302,), (0.3069,))
    return input_image_tensor_norm
```

And we then apply the pre-processing function to the sample image:

```
input_tensor = image_to_tensor(image)
```

1. In addition to the code under *step 3*, we also execute the following lines of code:

```
for p in model.parameters():
    p.requires_grad_(False)
```

If we do not do this, the traced model will have all parameters requiring gradients and we will have to load the model within the `torch.no_grad()` context.

1. We already have the loaded PyTorch model object with pre-trained weights. We are ready to trace the model with a dummy input as shown next:

```
demo_input = torch.ones(1, 1, 28, 28)
traced_model = torch.jit.trace(model, demo_input)
```

The dummy input is an image with all pixel values set to `1`.

1. We can now look at the traced model graph by running this:

```
print(traced_model.graph)
```

This should output the following:

```
graph(%self.1 : __torch__.torch.nn.modules.module.___torch_mangle_6.Module,
      %input.1 : Float(1, 1, 28, 28)):
  %113 : __torch__.torch.nn.modules.module.___torch_mangle_5.Module = prim::GetAttr[name="fc2"](%self.1)
  %110 : __torch__.torch.nn.modules.module.___torch_mangle_3.Module = prim::GetAttr[name="dp2"](%self.1)
  %109 : __torch__.torch.nn.modules.module.___torch_mangle_4.Module = prim::GetAttr[name="fc1"](%self.1)
  %106 : __torch__.torch.nn.modules.module.___torch_mangle_2.Module = prim::GetAttr[name="dp1"](%self.1)
  %105 : __torch__.torch.nn.modules.module.___torch_mangle_1.Module = prim::GetAttr[name="cn2"](%self.1)
  %102 : __torch__.torch.nn.modules.module.Module = prim::GetAttr[name="cn1"](%self.1)
  %120 : Tensor = prim::CallMethod[name="forward"](%102, %input.1)
  %input.3 : Float(1, 16, 26, 26) = aten::relu(%120) # /Users/ashish.jha/opt/anaconda3/lib/python3.7/site-packages/to
rch/nn/functional.py:914:0
  %121 : Tensor = prim::CallMethod[name="forward"](%105, %input.3)
  %input.5 : Float(1, 32, 24, 24) = aten::relu(%121) # /Users/ashish.jha/opt/anaconda3/lib/python3.7/site-packages/to
rch/nn/functional.py:914:0
```



```
  %input.9 : Float(1, 64) = aten::relu(%123) # /Users/ashish.jha/opt/anaconda3/lib/python3.7/site-packages/torch/nn/f
unctional.py:914:0
  %124 : Tensor = prim::CallMethod[name="forward"](%110, %input.9)
  %125 : Tensor = prim::CallMethod[name="forward"](%113, %124)
  %91 : int = prim::Constant[value=1]() # /Users/ashish.jha/opt/anaconda3/lib/python3.7/site-packages/torch/nn/functi
onal.py:1317:0
  %92 : None = prim::Constant()
  %93 : Float(1, 10) = aten::log_softmax(%125, %91, %92) # /Users/ashish.jha/opt/anaconda3/lib/python3.7/site-package
s/torch/nn/functional.py:1317:0
  return (%93)
```

Figure 13 .16 – Traced model graph

Intuitively, the first few lines in the graph show the initialization of layers of this model, such as `cn1`, `cn2`, and so on. Toward the end, we see the last layer, that is, the softmax layer. Evidently, the graph is written in a lower-level language with statically typed variables and closely resembles the TorchScript language.

1. Besides the graph, we can also look at the exact TorchScript code behind the traced model by running this:

Copy Explain

```
print(traced_model.code)
```

This should output the following lines of Python-like code that define the forward pass method for the model:

```
def forward(self,
    input: Tensor) -> Tensor:
  _0 = self.fc2
  _1 = self.dp2
  _2 = self.fc1
  _3 = self.dp1
  _4 = self.cn2
  input0 = torch.relu((self.cn1).forward(input, ))
  input1 = torch.relu((_4).forward(input0, ))
  input2 = torch.max_pool2d(input1, [2, 2], annotate(List[int], []), [0, 0], [1, 1], False)
  input3 = torch.flatten((_3).forward(input2, ), 1, -1)
  input4 = torch.relu((_2).forward(input3, ))
  _5 = (_0).forward((_1).forward(input4, ), )
  return torch.log_softmax(_5, 1, None)
```

Figure 13 .17 – Traced model code

This precisely is the TorchScript equivalent for the code that we wrote using PyTorch in *step 2*.

1. Next, we will export or save the traced model:

```
torch.jit.save(traced_model, 'traced_convnet.pt')
```

1. Now we load the saved model:

```
loaded_traced_model = torch.jit.load('traced_convnet.pt')
```

Note that we didn't need to load the model architecture and parameters separately.

1. Finally, we can use this model for inference:

```
loaded_traced_model(input_tensor.unsqueeze(0))
```

The output is as follows:

This should output the following:

```
tensor([[-9.3505e+00, -1.2089e+01, -2.2391e-03, -8.9248e+00, -9.8197e+00,
         -1.3350e+01, -9.0460e+00, -1.4492e+01, -6.3023e+00, -1.2283e+01]])
```

Figure 13 .18 – Traced model inference

1. We can check these results by re-running model inference on the original model:

```
model(input_tensor.unsqueeze(0))
```

This should produce the same output as in *Figure 13 .18*, which verifies that our traced model is working properly.

You can use the traced model instead of the original PyTorch model object to build more efficient Flask model servers and Dockerized model microservices, thanks to the GIL-free nature of TorchScript. While tracing is a viable option for JIT compiling

PyTorch models, it has some drawbacks.

For instance, if the forward pass of the model consists of control flows such as `if` and `for` statements, then the tracing will only render one of the multiple possible paths in the flow. In order to accurately translate PyTorch code to TorchScript code for such scenarios, we will use the other compilation mechanism called scripting.

Model scripting with TorchScript

Please follow *steps 1 to 6* from the previous exercise and then follow up with the steps given in this exercise. The full code is available in our github repository [13.20] :

1. For scripting, we need not provide any dummy input to the model, and the following line of code transforms PyTorch code to TorchScript code directly:

```
scripted_model = torch.jit.script(model)
```

1. Let's look at the scripted model graph by running the following line of code:

```
print(scripted_model.graph)
```

This should output the scripted model graph in a similar fashion as the traced model graph, as shown in the following figure:

```
graph(%self : __torch__.ConvNet,
      %x.1 : Tensor):
  %51 : Function = prim::Constant[name="log_softmax"]()
  %49 : int = prim::Constant[value=3]()
  %33 : int = prim::Constant[value=-1]()
  %26 : Function = prim::Constant[name="_max_pool2d"]()
  %20 : int = prim::Constant[value=0]()
  %19 : None = prim::Constant()
  %7 : Function = prim::Constant[name="relu"]()
  %6 : bool = prim::Constant[value=0]()
                                                    :
                                                    :
                                                    :
  %x.19 : Tensor = prim::CallFunction(%7, %x.17, %6) # <ipython-input-3-936a1c5cab85>:20:12
  %42 : __torch__.torch.nn.modules.dropout.___torch_mangle_1.Dropout2d = prim::GetAttr[name
="dp2"](%self)
  %x.21 : Tensor = prim::CallMethod[name="forward"](%42, %x.19) # <ipython-input-3-936a1c5cab
85>:21:12
  %45 : __torch__.torch.nn.modules.linear.___torch_mangle_2.Linear = prim::GetAttr[name="fc
2"](%self)
  %x.23 : Tensor = prim::CallMethod[name="forward"](%45, %x.21) # <ipython-input-3-936a1c5cab
85>:22:12
  %op.1 : Tensor = prim::CallFunction(%51, %x.23, %32, %49, %19) # <ipython-input-3-936a1c5ca
b85>:23:13
  return (%op.1)
```

Figure 13 .19 – Scripted model graph

Once again, we can see similar, verbose, low-level script that lists the various edges of the graph per line. Notice that the graph here is not the same as in *Figure 13 .16*, which indicates differences in code compilation strategy in using tracing rather than scripting.

1. We can also look at the equivalent TorchScript code by running this:

```
print(scripted_model.code)
```

Copy Explain

This should output the following:

```
def forward(self,
    x: Tensor) -> Tensor:
  _0 = __torch__.torch.nn.functional.___torch_mangle_12.relu
  _1 = __torch__.torch.nn.functional._max_pool2d
  _2 = __torch__.torch.nn.functional.___torch_mangle_13.relu
  _3 = __torch__.torch.nn.functional.log_softmax
  x0 = (self.cn1).forward(x, )
  x1 = __torch__.torch.nn.functional.relu(x0, False, )
  x2 = (self.cn2).forward(x1, )
  x3 = _0(x2, False, )
  x4 = _1(x3, [2, 2], None, [0, 0], [1, 1], False, False, )
  x5 = (self.dp1).forward(x4, )
  x6 = torch.flatten(x5, 1, -1)
  x7 = (self.fc1).forward(x6, )
  x8 = _2(x7, False, )
  x9 = (self.dp2).forward(x8, )
  x10 = (self.fc2).forward(x9, )
  return _3(x10, 1, 3, None, )
```

Figure 13 .20 – Scripted model code

In essence, the flow is similar to that in *Figure 13 .17*; however, there are subtle differences in the code signature resulting from differences in compilation strategy.

1. Once again, the scripted model can be exported and loaded back in the following way:

```
torch.jit.save(scripted_model, 'scripted_convnet.pt')
loaded_scripted_model = torch.jit.load('scripted_convnet.pt')
```

1. Finally, we use the scripted model for inference using this:

```
loaded_scripted_model(input_tensor.unsqueeze(0))
```

This should produce the exact same results as in *Figure 13 .18,* which verifies that the scripted model is working as expected.

Similar to tracing, a scripted PyTorch model is GIL-free and hence can improve model serving performance when used with Flask or Docker. *Table 13 .1* shows a quick comparison between the model tracing and scripting approaches:

Tracing	Scripting
• Dummy input is needed. • Records a fixed sequence of mathematical operations by passing the dummy input to the model. • Cannot handle multiple control flows (if-else) within the model forward pass. • Works even if the model has PyTorch functionalities that are not supported by TorchScript (`https://pytorch.org/docs/stable/jit_unsupported.html`).	• No need for dummy input. • Generates TorchScript code/graph by inspecting the `nn.Module` contents within the PyTorch code. • Useful in handling all types of control flows. • Scripting can work only if the PyTorch model does not contain any functionalities which are not supported by TorchScript.

Table 13 .1 – Tracing versus scripting

We have so far demonstrated how PyTorch models can be translated and serialized as TorchScript models. In the next section, we will completely get rid of Python for a moment and demonstrate how to load the TorchScript serialized model using C++.

Running a PyTorch model in C++

Python can sometimes be limiting or we might be unable to run machine learning models trained using PyTorch and Python. In this section, we will use the serialized TorchScript model objects (using tracing and scripting) that we exported in the previous section to run model inferences inside C++ code.

Note
Basic working knowledge of C++ is assumed for this section [13.21] . This section specifically talks a lot about C++ code compilation [13.22]

For this exercise, we need to install CMake following the steps mentioned in [13.23] to be able to build the C++ code. After that, we will create a folder named `cpp_convnet` in the current working directory and work from that directory:

1. Let's get straight into writing the C++ file that will run the model inference pipeline. The full C++ code is available here in our github respository [13.24]:

Copy Explain

```
#include <torch/script.h>
...
int main(int argc, char **argv) {
    Mat img = imread(argv[2], IMREAD_GRAYSCALE);
```

First the `.jpg` image file is read as a grayscale image using the OpenCV library. You will need to install the OpenCV library for C++ as per your OS requirements - Mac [13.25], Linux [13.26] or Windows [13.27].

1. The grayscale image is then resized to `28x28` pixels as that is the requirement for our CNN model:

Copy Explain

```
resize(img, img, Size(28, 28));
```

1. The image array is then converted to a PyTorch tensor:

Copy Explain

```
auto input_ = torch::from_blob(img.data, { img.rows, img.cols, img.channels() },
at::kByte);
```

For all `torch`-related operations as in this step, we use the `libtorch` library, which is the home for all `torch` C++-related APIs. If you have PyTorch installed, you need not install LibTorch separately.

1. Because OpenCV reads the grayscale in (28, 28, 1) dimension, we need to turn it around as (1, 28, 28) to suit the PyTorch requirements. The tensor is then reshaped to shape (1,1,28,28), where the first `1` is `batch_size` for inference and the second `1` is the number of channels, which is `1` for grayscale:

```
    auto input = input_.permute({2,0,1}).unsqueeze_(0).reshape({1, 1, img.rows,
img.cols}).toType(c10::kFloat).div(255);
    input = (input - 0.1302) / 0.3069;
```

Because OpenCV read images have pixel values ranging from `0` to `255`, we normalize these values to the range of `0` to `1`. Thereafter, we standardize the image with mean `0.1302` and std `0.3069`, as we did in a previous section (see *step 2* of the *Building the inference pipeline* section).

1. In this step, we load the JIT-ed TorchScript model object that we exported in the previous exercise:

```
    auto module = torch::jit::load(argv[1]);
    std::vector<torch::jit::IValue> inputs;
    inputs.push_back(input);
```

1. Finally, we come to the model prediction, where we use the loaded model object to make a forward pass with the supplied input data (an image, in this case):

```
auto output_ = module.forward(inputs).toTensor();
```

The `output_` variable contains a list of probabilities for each class. Let's extract the class label with the highest probability and print it:

```
auto output = output_.argmax(1);
cout << output << '\n';
```

Finally, we successfully exit the C++ routine:

```
    return 0;
}
```

1. While *steps 1-6* concern the various parts of our C++, we also need to write a `CMakeLists.txt` file in the same working directory. The full code for this file is available in our github repository [13.28] :

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(cpp_convnet)
find_package(Torch REQUIRED)
find_package(OpenCV REQUIRED)
add_executable(cpp_convnet cpp_convnet.cpp)
...
```

This file is basically the library installation and building script similar to `setup.py` in a Python project. In addition to this code, the `OpenCV_DIR` environment variable needs to be set to the path where the OpenCV build artifacts are created, shown in the following code block:

```
export OpenCV_DIR=/Users/ashish.jha/code/personal/MasteringPyTorchV2     /
Chapter13     /cpp_convnet/build_opencv/
```

1. Next, we need to actually run the `CMakeLists` file to create build artifacts. We do so by creating a new directory in the current working directory and run the build process from there. In the command line, we simply need to run the following:

```
mkdir build
cd build
cmake -
DCMAKE_PREFIX_PATH=/Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch/lib/python3.9/si
packages/torch     /share/cmake/ ..
cmake --build . --config Release
```

In the third line, you shall provide the path to LibTorch. To find your own, open Python and execute this:

```
import torch; torch.__path__
```

For me, it outputs this:

```
['/Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch/lib/python3.9/site-
packages/torch     ']_
```

Executing the third line shall output the following:

```
-- The C compiler identification is AppleClang 13.1.6.13160021
-- The CXX compiler identification is AppleClang 13.1.6.13160021
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /Library/Developer/CommandLineTools/usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /Library/Developer/CommandLineTools/usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD
-- Performing Test CMAKE_HAVE_LIBC_PTHREAD - Success
-- Found Threads: TRUE
-- MKL_ARCH: None, set to ` intel64` by default
-- MKL_ROOT /Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch_7_chaps
-- MKL_LINK: None, set to ` dynamic` by default
-- MKL_INTERFACE_FULL: None, set to ` intel_ilp64` by default
-- MKL_THREADING: None, set to ` intel_thread` by default
-- MKL_MPI: None, set to ` mpich` by default
CMake Warning at /Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch_7_chaps/lib/python3.9/site-packages/torch/share/cmake/Torch/TorchConfig.cmake:22 (message):
  static library kineto_LIBRARY-NOTFOUND not found.
Call Stack (most recent call first):
  /Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch_7_chaps/lib/python3.9/site-packages/torch/share/cmake/Torch/TorchConfig.cmake:127 (append_torchlib_if_found)
  CMakeLists.txt:3 (find_package)


-- Found Torch: /Users/ashish.jha/opt/anaconda3/envs/mastering_pytorch_7_chaps/lib/python3.9/site-packages/torch/lib/libtorch.dylib
-- Found OpenCV: /Users/ashish.jha/code/personal/Mastering-PyTorch/Chapter13/cpp_convnet/build_opencv (found version "4.6.0")
-- Configuring done
-- Generating done
-- Build files have been written to: /Users/ashish.jha/code/personal/Mastering-PyTorch/Chapter13/cpp_convnet/build
```

Figure 13 .21 – The C++ CMake output

And the fourth line should result in this:

```
Scanning dependencies of target cpp_convnet
[ 50%] Building CXX object CMakeFiles/cpp_convnet.dir/cpp_convnet.cpp.o
[100%] Linking CXX executable cpp_convnet
[100%] Built target cpp_convnet
```

Figure 13 .22 – C++ model building

1. Upon successful execution of the previous step, we will have produced a C++ compiled binary with the name `cpp_convnet`. It is now time to execute this binary program. In other words, we can now supply a sample image to our C++ model for inference. We may use the scripted model as input:

```
./cpp_convnet ../../scripted_convnet.pt ../../digit_image.jpg
```

Alternatively, we may use the traced model as input:

```
./cpp_convnet ../../traced_convnet.pt ../../digit_image.jpg
```

Either of these should result in the following output:

```
 2
[ CPULongType{1} ]
```

Figure 13 .23 – C++ model prediction

According to *Figure 13 .3*, the C++ model seems to be working correctly. Because we have used a different image handling library in C++ (that is, OpenCV) as compared to in Python (PIL), the pixel values are slightly differently encoded, which will result in slightly different prediction probabilities, but the final model prediction in the two languages should not differ significantly if correct normalizations are applied.

This concludes our exploration of PyTorch model inference using C++. This exercise shall help you get started with transporting your favorite deep learning models written and trained using PyTorch into a C++ environment, which should make predictions more efficient as well as opening up the possibility of hosting models in Python-less environments (for example, certain embedded systems, drones, and so on).

In the next section, we will move away from TorchScript and discuss a universal neural network modeling format – ONNX – that has enabled model usage across deep learning frameworks, programming languages, and OSes. We will work on loading a PyTorch trained model for inference in TensorFlow.

Using ONNX to export PyTorch models

There are scenarios in production systems where most of the already-deployed machine learning models are written in a certain deep learning library, say, TensorFlow, with its own sophisticated model-serving infrastructure. However, if a certain model is written using PyTorch, we would like it to be runnable using TensorFlow to conform to the serving strategy. This is one among various other use cases where a framework such as ONNX is useful.

ONNX is a universal format where the essential operations of a deep learning model such as matrix multiplications and activations, written differently in different deep learning libraries, are standardized. It enables us to interchangeably use different deep learning libraries, programming languages, and even operating environments to run the same deep learning model.

Here, we will demonstrate how to run a model, trained using PyTorch, in TensorFlow. We will first export the PyTorch model into ONNX format and then load the ONNX model inside TensorFlow code.

ONNX works with restricted versions of TensorFlow and hence we will work with `tensorflow==1.15.0`. And because of this we will work with python 3.7, as tensorflow==1.15.0 is not available in the newer versions of python. You can create and activate a new conda environment with python 3.7 with the following command:

```
conda create -n <env_name> python=3.7
source activate <env_name>
```

We will also need to install the `onnx==1.7.0` and `onnx-tf==1.5.0` libraries for the exercise. The full code for this exercise is available in our github repository [13.29] . Please follow *steps 1 to 11* from the *Model tracing with TorchScript* section, and then follow up with the steps given in this exercise:

1. Similar to model tracing, we again pass a dummy input through our loaded model:

```
demo_input = torch.ones(1, 1, 28, 28)
torch.onnx.export(model, demo_input, "convnet.onnx")
```

This should save a model `onnx` file. Under the hood, the same mechanism is used for serializing the model as was used in model tracing.

1. Next, we load the saved `onnx` model and convert it into a TensorFlow model:

```
import onnx
from onnx_tf.backend import prepare
model_onnx = onnx.load("./convnet.onnx")
tf_rep = prepare(model_onnx)
tf_rep.export_graph("./convnet.pb")
```

1. Next, we load the serialized `tensorflow` model to parse the model graph. This will help us in verifying that we have loaded the model architecture correctly as well as in identifying the input and output nodes of the graph:

Copy Explain

```
with tf.gfile.GFile("./convnet.pb", "rb") as f:
    graph_definition = tf.GraphDef()
    graph_definition.ParseFromString(f.read())
with tf.Graph().as_default() as model_graph:
    tf.import_graph_def(graph_definition, name="")
for op in model_graph.get_operations():
    print(op.values())
```

This should output the following:

```
(<tf.Tensor 'Const:0' shape=(16,) dtype=float32>,)
(<tf.Tensor 'Const_1:0' shape=(16, 1, 3, 3) dtype=float32>,)
(<tf.Tensor 'Const_2:0' shape=(32,) dtype=float32>,)
(<tf.Tensor 'Const_3:0' shape=(32, 16, 3, 3) dtype=float32>,)
(<tf.Tensor 'Const_4:0' shape=(64,) dtype=float32>,)
(<tf.Tensor 'Const_5:0' shape=(64, 4608) dtype=float32>,)
(<tf.Tensor 'Const_6:0' shape=(10,) dtype=float32>,)
(<tf.Tensor 'Const_7:0' shape=(10, 64) dtype=float32>,)
(<tf.Tensor 'input.1:0' shape=(1, 1, 28, 28) dtype=float32>,)
(<tf.Tensor 'transpose/perm:0' shape=(4,) dtype=int32>,)
(<tf.Tensor 'transpose:0' shape=(3, 3, 1, 16) dtype=float32>,)
                                .
                                .
                                .
(<tf.Tensor 'mul_2/x:0' shape=() dtype=float32>,)
(<tf.Tensor 'mul_2:0' shape=(1, 10) dtype=float32>,)
(<tf.Tensor 'mul_3/x:0' shape=() dtype=float32>,)
(<tf.Tensor 'mul_3:0' shape=(10,) dtype=float32>,)
(<tf.Tensor 'add_3:0' shape=(1, 10) dtype=float32>,)
(<tf.Tensor '18:0' shape=(1, 10) dtype=float32>,)
```

Figure 13 .24 – TensorFlow model graph

From the graph, we are able to identify the input and output nodes, as marked.

1. Finally, we can assign variables to the input and output nodes of the neural network model, instantiate a TensorFlow session, and run the graph to generate predictions for our sample image:

Copy Explain

```
model_output = model_graph.get_tensor_by_name('18:0')
model_input = model_graph.get_tensor_by_name('input.1:0')
sess = tf.Session(graph=model_graph)
output = sess.run(model_output, feed_dict={model_input: input_tensor.unsqueeze(0)})
print(output)
```

This should output the following:

```
[[-9.35050774e+00 -1.20893326e+01 -2.23922171e-03 -8.92477798e+00
  -9.81972313e+00 -1.33498535e+01 -9.04598618e+00 -1.44924192e+01
  -6.30233145e+00 -1.22827682e+01]]
```

Figure 13 .25 – TensorFlow model prediction

As you can see, in comparison with *Figure 13 .18*, the predictions are exactly the same for the TensorFlow and PyTorch versions of our model. This validates the successful functioning of the ONNX framework. I encourage you to dissect the TensorFlow model further and understand how ONNX helps regenerate the exact same model in a different deep learning library by utilizing the underlying mathematical operations in the model graph.

This concludes our discussion of the different ways of exporting PyTorch models. The techniques covered here will be useful in deploying PyTorch models in production systems as well as in working across various platforms. As new versions of deep learning libraries, programming languages, and even OSes keep coming, this is an area that will rapidly evolve accordingly.

Hence, it is highly advisable to keep an eye on the developments and make sure to use the latest and most efficient ways of exporting models as well as operationalizing them into production.

So far, we have been working on our local machines for serving and exporting our PyTorch models. In the next and final section of this chapter, we will briefly look at serving PyTorch models on some of the well-known cloud platforms, such as AWS, Google Cloud, and Microsoft Azure.

Serving PyTorch models in the cloud

Deep learning is computationally expensive and therefore demands powerful and sophisticated computational hardware. Not everyone might have access to a local machine that has enough CPUs and GPUs to train gigantic deep learning models in a reasonable time. Furthermore, we cannot guarantee 100 percent availability for a local machine that is serving a trained model for inference. For reasons such as these, cloud computing platforms are a vital alternative for both training and serving deep learning models.

In this section, we will discuss how to use PyTorch with some of the most popular cloud platforms – **AWS**, **Google Cloud**, and **Microsoft Azure**. We will explore the different ways of serving a trained PyTorch model in each of these platforms. The model-serving exercises we discussed in the earlier sections of this chapter were executed on a local machine. The goal of this section is to enable you to perform similar exercises using **virtual machines** (**VMs**) on the cloud.

Using PyTorch with AWS

AWS is the oldest and one of the most popular cloud computing platforms. It has deep integrations with PyTorch. We have already seen an example of it in the form of TorchServe, which is jointly developed by AWS and Facebook.

In this section, we will look at some of the common ways of serving PyTorch models using AWS. First, we will simply learn how to use an AWS instance as a replacement for our local machine (laptop) to serve PyTorch models. Then, we will briefly discuss Amazon SageMaker, which is a fully dedicated cloud machine learning platform. We will briefly discuss how TorchServe can be used together with SageMaker for model serving.

Note
This section assumes basic familiarity with AWS. Therefore, we will not be elaborating on topics such as what an AWS EC2 instance is, what AMIs are, how to create an instance, and so on [13.30]. . We will instead focus on the components of AWS that are related to PyTorch.

Serving a PyTorch model using an AWS instance

In this section, we will demonstrate how we can use PyTorch within a VM – an AWS instance, in this case. After reading this section, you will be able to execute the exercises discussed in the *Model serving in PyTorch* section inside an AWS instance.

First, you will need to create an AWS account if you haven't done so already. Creating an account requires an email address and a payment method (credit card) [13.31]. .

Once you have an AWS account, you may log in to enter the AWS console [13.32] . From here, we basically need to instantiate a VM (AWS instance) where we can start using PyTorch for training and serving models. Creating a VM requires two decisions [13.33]:

Choosing the hardware configuration of the VM, also known as the **AWS instance type**

Choosing the **Amazon Machine Image (AMI)**, which entails all the required software, such as the OS (Ubuntu or Windows), Python, PyTorch, and so on

. Typically, when we refer to an AWS instance, we are referring to an **Elastic Cloud Compute** instance, also known as an **EC2** instance.

Based on the computational requirements of the VM (RAM, CPUs, and GPUs), you can choose from a long list of EC2 instances provided by AWS[13.34] . Because PyTorch heavily leverages GPU compute power, it is recommended to use EC2 instances that include GPUs, though they are generally costlier than CPU-only instances.

Regarding AMIs, there are two possible approaches to choosing an AMI. You may go for a barebones AMI that only has an OS installed, such as Ubuntu (Linux). In this case, you can then manually install Python [13.35] and subsequently install PyTorch [13.36] .

An alternative and more recommended way is to start with a pre-built AMI that has PyTorch installed already. AWS offers Deep Learning AMIs, which make the process of getting started with PyTorch on AWS much faster and easier [13.37] .

Once you have launched an instance successfully using either of the suggested approaches, you may simply connect to the instance using one of the various available methods [13.38] .

SSH is one of the most common ways of connecting to an instance. Once you are inside the instance, it will have the same layout as working on a local machine. One of the first logical steps would then be to test whether PyTorch is working inside the machine.

To test, first open a Python interactive session by simply typing `python` on the command line. Then, execute the following line of code:

Copy Explain

```
import torch
```

If it executes without error, it means that you have PyTorch installed on the system.

At this point, you can simply fetch all the code that we wrote in the preceding sections of this chapter on model serving. On the command line inside your home directory, simply clone this book's GitHub repository by running this:

```
git clone https://github.com/arj7192/MasteringPyTorchV2.git
```

Then, within the `Chapter13` subfolder, you will have all the code to serve the MNIST model that we worked on in the previous sections. You can basically re-run the exercises, this time on the AWS instance instead of your local computer.

Let's review the steps we need to take for working with PyTorch on AWS:

1. Create an AWS account.
2. Log in to the AWS console.
3. Click on the **Launch a virtual machine** button in the console.
4. Select an AMI. For example, select the Deep Learning AMI (Ubuntu).
5. Select an AWS instance type. For example, select **p.2x large**, as it contains a GPU.
6. Click **Launch**.
7. Click **Create a new key pair**. Give the key pair a name and download it locally.
8. Modify permissions of this key-pair file by running this on the command line:

```
chmod 400 downloaded-key-pair-file.pem
```

1. On the console, click on **View Instances** to see the details of the launched instance and specifically note the public IP address of the instance.
2. Using SSH, connect to the instance by running this on the command line:

```
ssh -i downloaded-key-pair-file.pem ubuntu@<Public IP address>
```

The public IP address is the same as obtained in the previous step.

1. Once connected, start a `python` shell and run `import torch` in the shell to ensure that PyTorch is correctly installed on the instance.
2. Clone this book's GitHub repository by running the following on the instance's command line:

```
git clone https://github.com/arj7192/MasteringPyTorchV2.git
```

1. Go to the `chapter13` folder within the repository and start working on the various model-serving exercises that are covered in the preceding sections of this chapter.

This brings us to the end of this section, where we have essentially learned how to start working with PyTorch on a remote AWS instance [13.39] . Next, we will look at AWS's fully dedicated cloud machine learning platform –Amazon SageMaker.

Using TorchServe with Amazon SageMaker

We have already discussed TorchServe in detail in the preceding section. As we know, TorchServe is a PyTorch model-serving library developed by AWS and Facebook. Instead of manually defining a model inference pipeline, model-serving APIs, and microservices, you can use TorchServe, which provides all of these functionalities.

Amazon SageMaker, on the other hand, is a cloud machine learning platform that offers functionalities such as the training of massive deep learning models as well as deploying and hosting trained models on custom instances. When working with SageMaker, all we need to do is this:

Specify the type and number of AWS instances we would like to spin up to serve the model.

Provide the location of the stored pre-trained model object.

We do not need to manually connect to the instance and serve the model using TorchServe. SageMaker takes care of all that. AWS website has some useful blogs t o get started with using SageMaker and TorchServe to serve PyTorch models on an industrial scale and within a few clicks [13.40] .AWS blogs also provides the use cases of Amazon SageMaker when working with PyTorch [13.41] .

Tools such as SageMaker are incredibly useful for scalability during both model training and serving. However, while using such one-click tools, we often tend to lose some flexibility and debuggability. Therefore, it is for you to decide what set of tools works best for your use case. This concludes our discussion on using AWS as a cloud platform for working with PyTorch. Next, we will look at another cloud platform – Google Cloud.

Serving PyTorch model on Google Cloud

Similar to AWS, you first need to create a Google account (*@gmail.com) if you do not have one already. Furthermore, to be able to log in to the Google Cloud console [13.42] , you will need to add a payment method (credit card details).

> **Note**
>
> We will not be covering the basics of Google Cloud here [13.43]. We will instead focus on using Google Cloud for serving PyTorch models within a VM.

Once inside the console, we need to follow the steps similar to AWS to launch a VM where we can serve our PyTorch model. You can always start with a barebones VM and manually install PyTorch. But we will be using Google's Deep Learning VM Image [13.44] , which has PyTorch pre-installed. Here are the steps for launching a Google Cloud VM and using it to serve PyTorch models:

1. Launch Deep Learning VM Image on Google Cloud using the Google Cloud marketplace [13.45] .
2. Input the deployment name in the command window. This name suffixed with `-vm` acts as the name of the launched VM. The command prompt inside this VM will look like this:

```
<user>@<deployment-name>-vm:~/
```

Here, `user` is the client connecting to the VM and `deployment-name` is the name of the VM chosen in this step.

1. Select `PyTorch` as the `Framework` in the next command window. This tells the platform to pre-install PyTorch in the VM.
2. Select the zone for this machine. Preferably, choose the zone geographically closest to you. Also, different zones have slightly different hardware offerings (VM configurations) and hence you might want to choose a specific zone for a specific machine configuration.

3. Having specified the software requirement in *step 3*, we shall now specify the hardware requirements. In the GPU section of the command window, we need to specify the GPU type and subsequently the number of GPUs to be included in the VM.

Google Cloud provides various GPU devices/configuratins [13.46] . In the GPU section, also tick the checkbox that will automatically install the NVIDIA drivers that are necessary to utilize the GPUs for deep learning.

1. Similarly, under the CPU section, we need to provide the machine type [13.47] . Regarding *step 5* and *step 6*, please be aware that different zones provide different machine and GPU types as well as different combinations of GPU types and GPU numbers.
2. Finally, click on the **Deploy** button. This will launch the VM and lead you to a page that will have all the instructions needed to connect to the VM from your local computer.
3. At this point, you may connect to the VM and ensure that PyTorch is correctly installed by trying to import PyTorch from within a Python shell. Once verified, clone this book's GitHub repository. Go to the `Chapter13` folder and start working on the model-serving exercises within this VM.

You can read more about creating the PyTorch deep learning VM on Google Cloud blogs [13.48]. This concludes our discussion of using Google Cloud as a cloud platform to work with PyTorch model serving. As you may have noticed, the process is very similar to that of AWS. In the next and final section, we will briefly look at using Microsoft's cloud platform, Azure, to work with PyTorch.

Serving PyTorch models with Azure

Once again, similar to AWS and Google Cloud, Azure requires a Microsoft-recognized email ID for signing up, along with a valid payment method.

Note
We assume a basic understanding of the Microsoft Azure cloud platform for this section [13.49] .

Once you have access to the Azure portal [13.50] , there are broadly two recommended ways of getting started with using PyTorch on Azure:

Data Science Virtual Machine (DSVM)

Azure Machine Learning

We will now discuss these approaches briefly.

Working on Azure's Data Science Virtual Machine

Similar to Google Cloud's Deep Learning VM Image, Azure offers its own DSVM image [13.51] , which is a fully dedicated VM image for data science and machine learning, including deep learning.

These images are available for Windows [13.52] as well as Linux/Ubuntu [13.53].

The steps to create a DSVM instance using this image are quite similar to the steps discussed for Google Cloud for both Windows [13.54] as well as Linux/Ubuntu [13.55].

Once you have created the DSVM, you can launch a Python shell and try to import the PyTorch library to ensure that it is correctly installed. You may further test the functionalities available in this DSVM for Linux [13.56] as well as Windows [13.57] .

Finally, you may clone this book's GitHub repository within the DSVM instance and use the code within the `Chapter13` folder to work on the PyTorch model-serving exercises discussed in this chapter.

Discussing Azure Machine Learning Service

Similar to and predating Amazon's SageMaker, Azure provides an end-to-end cloud machine learning platform. The Azure Machine Learning Service (AMLS) comprises the following (to name just a few):

Azure Machine Learning VMs

Notebooks

Virtual environments

Datastores

Tracking machine learning experiments

Data labeling

A key difference between AMLS VMs and DSVMs is that the former are fully managed. For instance, they can be scaled up or down based on the model training or serving requirements [13.58] .

Just like SageMaker, Azure Machine Learning is useful both for training large-scale models as well as deploying and serving those models. Azure website has a great tutorial for training PyTorch models on AMLS as well as for deploying PyTorch models on AMLS for Windows [13.59] as well as Linux [13.60] .

Azure Machine Learning aims at providing a one-click interface to the user for all machine learning tasks. Hence, it is important to keep in mind the flexibility trade-off. Although we have not covered all the details about Azure Machine Learning here, Azure's website is a good resource for further reading [13.61] .

This brings us to the end of discussing what Azure has to offer as a cloud platform for working with PyTorch [13.62] .

And that also concludes our discussion of using PyTorch to serve models on the cloud. We have discussed AWS, Google Cloud, and Microsoft Azure in this section. Although there are more cloud platforms available out there, the nature of their offerings and the ways of using PyTorch within those platforms will be similar to what we have discussed. This section will help you in getting started with working on your PyTorch projects on a VM in the cloud.

Summary

In this chapter, we have explored the world of deploying trained PyTorch deep learning models in production systems.

In the next chapter, we will look at another practical aspect of working with models in PyTorch that helps immensely in saving time and resources while training and validating deep learning models .

Previous Chapter

Next Chapter