PyTorch

Introduction To PyTorch Part 2

> Hamza Fatnaoui

> Hicham Ibn Issaghyr

> Ikram Arif

A.6 Setting up efficient data loaders

Before we can train this model, we have to briefly talk about creating efficient data loaders in PyTorch, which we will iterate over when training the model

Let's start by creating a simple toy dataset of five training examples with two features each. Accompanying the training examples, we also create a tensor containing the corresponding class labels: three examples below to class 0, and two examples belong to class 1. In addition, we also make a test set consisting of two entries. The code to create this dataset is shown below.

```
X_train = torch.tensor([
  [-1.2, 3.1],
  [-0.9, 2.9],
  [-0.5, 2.6],
  [2.3, -1.1],
  [2.7, -1.5]
  ])
y_train = torch.tensor([0, 0, 0, 1, 1])
X_test = torch.tensor([
  [-0.8, 2.8],
  [2.6, -1.6],
  ])
y_test = torch.tensor([0, 1])
```

PyTorch requires that class labels start with label 0, and the largest class label value should not exceed the number of output nodes minus 1 (since Python index counting starts at 0. So, if we have class labels 0, 1, 2, 3, and 4, the neural network output layer should consist of 5 nodes.

Next, we create a custom dataset class, ToyDataset, by subclassing from PyTorch's Dataset parent class, as shown below.

```
# Defining a custom Dataset class
from torch.utils.data import Dataset

class ToyDataset(Dataset):
  def __init__(self, X, y):
    self.features = X
    self.labels = y
  def __getitem__(self, index):
    one_x = self.features[index]
    one_y = self.labels[index]
    return one_x, one_y
  def __len__(self):
    return self.labels.shape[0]

train_ds = ToyDataset(X_train,y_train)
test_ds = ToyDataset(X_test,y_test)
```

we can use PyTorch's DataLoader class to sample from it, as shown in the code listing below:

```
from torch.utils.data import DataLoader
torch.manual_seed(123)   # for the exact same shuffling
train_loader = DataLoader(
  dataset=train_ds,
  batch_size=2,
  shuffle=True,
  num_workers=0
)
test_loader = DataLoader(
  dataset=test_ds,
  batch_size=2,
  shuffle=False,
  num_workers=0
)
```

After instantiating the training data loader, we can iterate over it as shown below

```
for idx,batch in enumerate(train_loader):
  print(f"batch {idx+1} : {batch}")
```

```
batch 1 : [tensor([[ 2.3000, -1.1000],
        [-0.9000,  2.9000]]), tensor([1, 0])]
batch 2 : [tensor([[-1.2000,  3.1000],
        [-0.5000,  2.6000]]), tensor([0, 0])]
batch 3 : [tensor([[ 2.7000, -1.5000]]), tensor([1])]
```

As we can see based on the output above, the train_loader iterates over the training dataset visiting each training example exactly once. This is known as a training epoch. Since we seeded the random number generator using torch.manual_seed(123) above, you should get the exact same shuffling order of training examples as shown above. However if you iterate over the dataset a second time, you will see that the shuffling order will change. This is desired to prevent deep neural networks getting caught in repetitive update cycles during training.

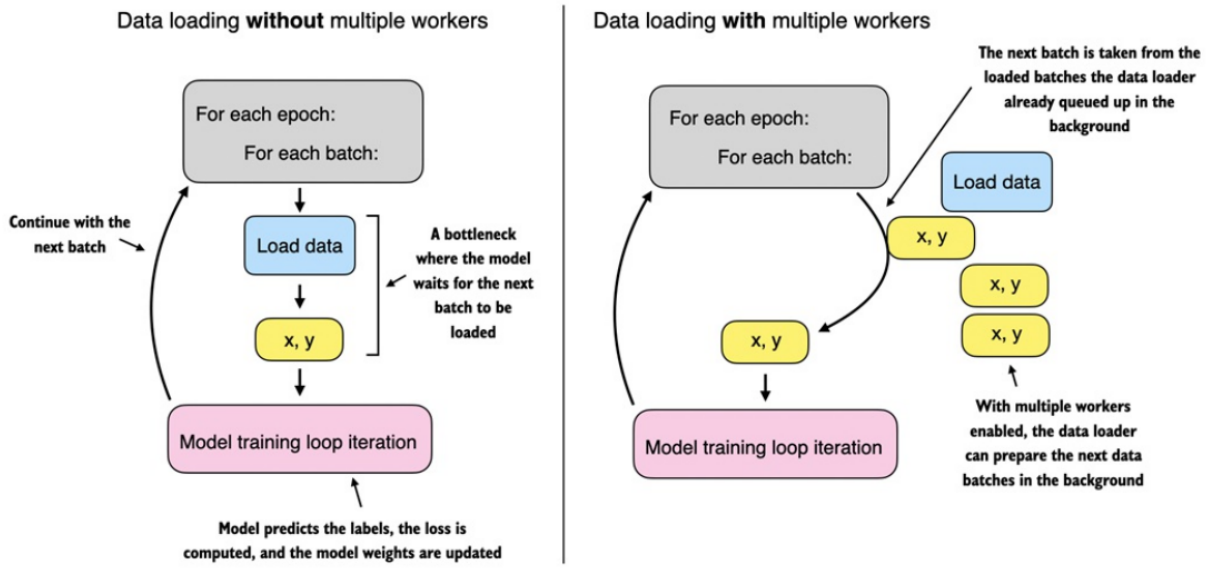
Note that we specified a batch size of 2 above, but the 3rd batch only contains a single example. That's because we have five training examples, which is not evenly divisible by 2. In practice, having a substantially smaller batch as the last batch in a training epoch can disturb the convergence during training. To prevent this, it's recommended to set drop_last=True, which will drop the last batch in each epoch, as shown below:

```
train_loader = DataLoader(
dataset=train_ds,
batch_size=2,
shuffle=True,
num_workers=0,
drop_last=True
)
```

```
for idx, (x, y) in enumerate(train_loader):
  print(f"Batch {idx+1}:", x, y)
```

```
Batch 1: tensor([[-1.2000,  3.1000],
        [-0.5000,  2.6000]]) tensor([0, 0])
Batch 2: tensor([[ 2.3000, -1.1000],
        [-0.9000,  2.9000]]) tensor([1, 0])
```

Lastly, let's discuss the setting num_workers=0 in the DataLoader. This parameter in PyTorch's DataLoader function is crucial for parallelizing data loading and preprocessing. When num_workers is set to 0, the data loading will be done in the main process and not in separate worker processes. This might seem unproblematic, but it can lead to significant slowdowns during model training when we train larger networks on a GPU. This is because instead of focusing solely on the processing of the deep learning model, the CPU must also take time to load and preprocess the data. As a result, the GPU can sit idle while waiting for the CPU to finish these tasks. In contrast, when num_workers is set to a number greater than zero, multiple worker processes are launched to load data in parallel, freeing the main process to focus on training your model and better utilizing your system's resources



However, if we are working with very small datasets, setting num_workers to 1 or larger may not be necessary since the total training time takes only fractions of a second anyway. On the contrary, if you are working with tiny datasets or interactive environments such as Jupyter notebooks, increasing num_workers may not provide any noticeable speedup. They might, in fact, lead to some issues. One potential issue is the overhead of spinning up multiple worker processes, which could take longer than the actual data loading when your dataset is small.

Furthermore, for Jupyter notebooks, setting num_workers to greater than 0 can sometimes lead to issues related to the sharing of resources between different processes, resulting in errors or notebook crashes. Therefore, it's essential to understand the trade-off and make a calculated decision on setting the num_workers parameter. When used correctly, it can be a beneficial tool but should be adapted to your specific dataset size and computational environment for optimal results.

Setting num_workers=4 usually leads to optimal performance on many real-world datasets, but optimal settings depend on your hardware and the code used for loading a training example defined in the Dataset class.

A.7 A typical training loop

The training code is shown in code listing below.

```
# Neural Network Training
import torch.nn.functional as F
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)
optimizer = torch.optim.SGD(model.parameters(), lr=0.5)

num_epochs = 3
for epoch in range(num_epochs):
  model.train()
  for batch_idx, (features, labels) in enumerate(train_loader):
    logits = model(features)
    loss = F.cross_entropy(logits, labels)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
    f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
    f" | Train Loss: {loss:.2f}")
    model.eval()
# Optional model evaluation
```

```
Epoch: 001/003 | Batch 000/002 | Train Loss: 0.75
Epoch: 001/003 | Batch 001/002 | Train Loss: 0.65
Epoch: 002/003 | Batch 000/002 | Train Loss: 0.44
Epoch: 002/003 | Batch 001/002 | Train Loss: 0.13
Epoch: 003/003 | Batch 000/002 | Train Loss: 0.03
Epoch: 003/003 | Batch 001/002 | Train Loss: 0.00
```

We introduced new settings called model.train() and model.eval(). As these names imply, these settings are used to put the model into a training and an evaluation mode. This is necessary for components that behave differently during training

and inference, such as dropout or batch normalization layers. Since we don't have dropout or other components in our NeuralNetwork class that are affected by these settings, using model.train() and model.eval() is redundant in our code above. However, it's best practice to include them anyway to avoid unexpected behaviors when we change the model architecture or reuse the code to train a different model.

As discussed earlier, we pass the logits directly into the cross_entropy loss function, which will apply the softmax function internally for efficiency and numerical stability reasons. Then, calling loss.backward() will calculate the gradients in the computation graph that PyTorch constructed in the background. The optimizer.step() method will use the gradients to update the model parameters to minimize the loss. In the case of the SGD optimizer, this means multiplying the gradients with the learning rate and adding the scaled negative gradient to the parameters.

It is important to include an optimizer.zero_grad() call in each update round to reset the gradients to zero. Otherwise, the gradients will accumulate, which may be undesired.

```
model.eval()
with torch.no_grad():
  outputs = model(X_train)
print(outputs)
```

```
tensor([[ 2.8569, -4.1618],
        [ 2.5382, -3.7548],
        [ 2.0944, -3.1820],
        [-1.4814,  1.4816],
        [-1.7176,  1.7342]])
```

We can convert these values into class labels predictions using PyTorch's argmax function, which returns the index position of the highest value in each row if we set dim=1 (setting dim=0 would return the highest value in each column, instead):

```
predictions = torch.argmax(outputs, dim=1)
print(predictions)
```

```
tensor([0, 0, 0, 1, 1])
```

Above, we computed the predicted labels for the training dataset. Since the training dataset is relatively small, we could compare it to the true training labels by eye and see that the model is 100% correct. We can double-check this using the == comparison operator:

```
# to check
predictions == y_train
```

```
tensor([True, True, True, True, True])
```

Using torch.sum, we can count the number of correct prediction as follows:

```
# we can count the number of the correct prediction
torch.sum(predictions == y_train)
```

```
tensor(5)
```

However, to generalize the computation of the prediction accuracy, let's implement a compute_accuracy function as shown in the following code listing.

```
# A function to compute the prediction accuracy
def compute_accuracy(model, dataloader):
  model = model.eval()
  correct = 0.0
  total_examples = 0
  for idx, (features, labels) in enumerate(dataloader):
    with torch.no_grad():
      logits = model(features)
    predictions = torch.argmax(logits, dim=1)
    compare = labels == predictions
    correct += torch.sum(compare)
    total_examples += len(compare)
  return (correct / total_examples).item()
```

We can then apply the function to the training and the testing as follows:

```
print(compute_accuracy(model, train_loader))
```

```
1.0
```

```
print(compute_accuracy(model, test_loader))
```

```
1.0
```

A.8 Saving and loading models

In the previous section, we successfully trained a model. Let's now see how we can save a trained model to reuse it later.

Here's the recommended way how we can save and load models in PyTorch:

```
# save the model
torch.save(model.state_dict(), "model.pth")
```

The model's state_dict is a Python dictionary object that maps each layer in the model to its trainable parameters (weights and biases). Note that "model.pth" is an arbitrary filename for the model file saved to disk. We can give it any name and file ending we like; however, .pth and .pt are the most common conventions.

Once we saved the model, we can restore it from disk as follows:

```
model = NeuralNetwork(2, 2)    # the input and the output size needs to match the ones in the saved model
```

```
model.load_state_dict(torch.load("model.pth"))
```

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

The torch.load("model.pth") function reads the file "model.pth" and reconstructs the Python dictionary object containing the model's parameters while model.load_state_dict() applies these parameters to the model, effectively restoring its learned state from when we saved it.

A.9 Optimizing training performance with GPUs

In this last section of this chapter, we will see how we can utilize GPUs, which will accelerate deep neural network training compared to regular CPUs

A.9.1 PyTorch computations on GPU devices

First, we need to introduce the notion of devices. In PyTorch, a device is where computations occur, and data resides. The CPU and the GPU are examples of

devices. A PyTorchtensor resides in a device, and its operations are executed on the same device.

Let's see how this works in action. Assuming that you installed a GPU- compatible version of PyTorch, Installing PyTorch, we can double-check that our runtime indeed supports GPU computing via the following code:

```
torch.cuda.is_available()
```

```
True
```

We can now use the .to() method[1] to transfer these tensors onto a GPU and perform the addition there:

```
tensor_1 = tensor_1.to("cuda")
tensor_2 = tensor_2.to("cuda")
print(tensor_1 + tensor_2)
```

Notice that the resulting tensor now includes the device information, device='cuda:0', which means that the tensors reside on the first GPU. If your machine hosts multiple GPUs, you have the option to specify which GPU you'd like to transfer the tensors to. You can do this by indicating the device ID in the transfer command. For instance, you can use .to("cuda:0"), .to("cuda:1"), and so on.

However, it is important to note that all tensors must be on the same device. Otherwise, the computation will fail

A.9.2 Single-GPU training

Now that we are familiar with transferring tensors to the GPU, we can modify the training loop to run on a GPU. This requires only changing three lines of code, as shown in code listing below.

```
torch.manual_seed(123)
model = NeuralNetwork(num_inputs=2, num_outputs=2)

device = torch.device("cuda")
model = model.to(device)

optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
num_epochs = 3
for epoch in range(num_epochs):
  model.train()
  for batch_idx, (features, labels) in enumerate(train_loader):

    features, labels = features.to(device), labels.to(device)

    logits = model(features)
    loss = F.cross_entropy(logits, labels) # Loss function
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    ### LOGGING
    print(f"Epoch: {epoch+1:03d}/{num_epochs:03d}"
    f" | Batch {batch_idx:03d}/{len(train_loader):03d}"
    f" | Train/Val Loss: {loss:.2f}")
  model.eval()
  # Optional model evaluation
```

We can also use .to("cuda") instead of device = torch.device("cuda"). We can also modify the statement to the following, which will make the same code executable on a CPU if a GPU is not available, which is usually considered best practice when sharing PyTorch code:

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

In the case of the modified training loop above, we probably won't see a speed-up because of the memory transfer cost from CPU to GPU. However, we can expect a significant speed-up when training deep neural networks, especially large language models.

As we saw in this section, training a model on a single GPU in PyTorch is relatively easy. Next, let's introduce another concept: training models on multiple GPUs.

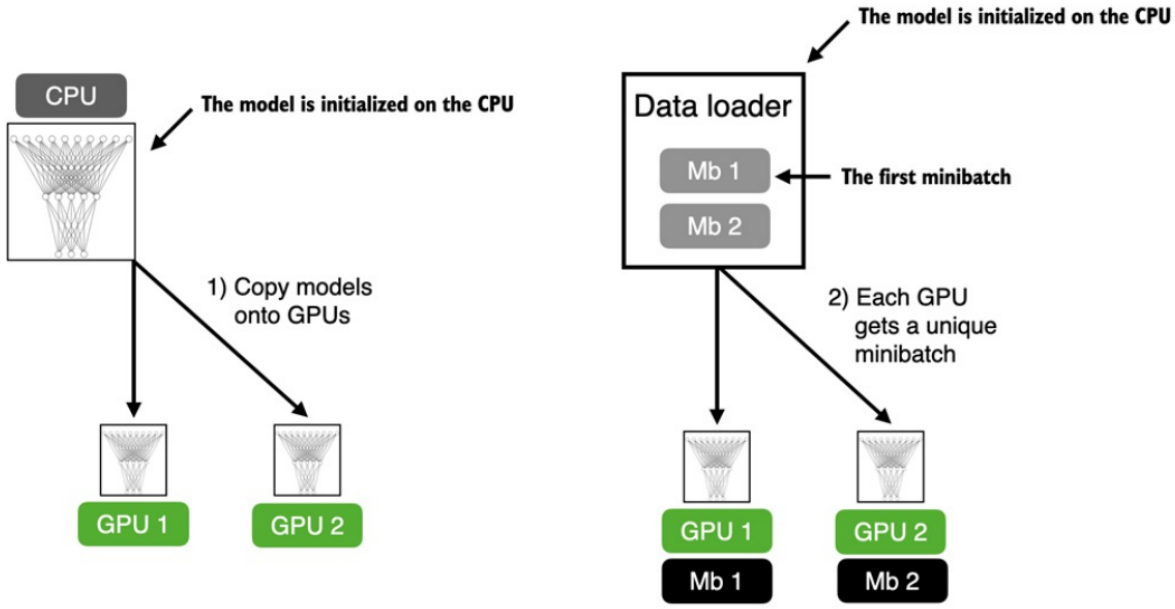
A.9.3 Training with multiple GPUs

In this section, we will briefly go over the concept of distributed training. Distributed training is the concept of dividing the model training across multiple GPUs and machines.

Why do we need this? Even when it is possible to train a model on a single GPU or machine, the process could be exceedingly time-consuming. The training time can be significantly reduced by distributing the training process across multiple machines, each with potentially multiple GPUs. This is particularly crucial in the experimental stages of model development, where numerous training iterations might be necessary to finetune the model parameters and architecture.

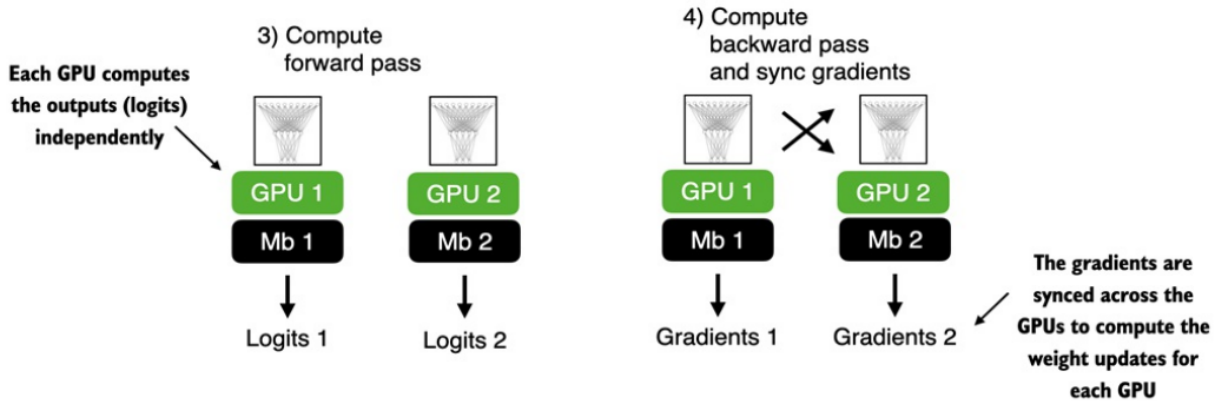
In this section, we will look at the most basic case of distributed training: PyTorch's DistributedDataParallel (DDP) strategy. DDP enables parallelism by splitting the input data across the available devices and processing these data subsets simultaneously.

How does this work? PyTorch launches a separate process on each GPU, and each process receives and keeps a copy of the model -- these copies will be synchronized during training. To illustrate this, suppose we have two GPUs that we want to use to train a neural network, as shown



Each of the two GPUs will receive a copy of the model. Then, in every training iteration, each model will receive a minibatch (or just batch) from the data loader. We can use a DistributedSampler to ensure that each GPU will receive a different, non-overlapping batch when using DDP.

Since each model copy will see a different sample of the training data, themodel copies will return different logits as outputs and compute different gradients during the backward pass. These gradients are then averaged and synchronized during training to update the models. This way, we ensure that the models don't diverge



The benefit of using DDP is the enhanced speed it offers for processing the dataset compared to a single GPU. Barring a minor communication overhead between devices that comes with DDP use, it can theoretically process a training epoch in half the time with two GPUs compared to just one. The time efficiency scales up with the number of GPUs, allowing us to process an epoch eight times faster if we have eight GPUs, and so on.

DDP does not function properly within interactive Python environments like Jupyter notebooks, which don't handle multiprocessing in the same way a standalone Python script does. Therefore, the following code should be executed as a script, not within a notebook interface like Jupyter. This is because DDP needs to spawn multiple processes, and each process should have its own Python interpreter instance.

First, we will import a few additional submodules, classes, and functions for distributed training PyTorch as shown in code listing below

```
import torch.multiprocessing as mp
from torch.utils.data.distributed import DistributedSampler
from torch.nn.parallel import DistributedDataParallel as DDP
from torch.distributed import init_process_group, destroy_process_group
```

PyTorch's multiprocessing submodule contains functions such as multiprocessing.spawn, which we will use to spawn multiple processes and apply a function to multiple inputs in parallel. We will use it to spawn one training process per GPU.

If we spawn multiple processes for training, we will need a way to divide the dataset among these different processes. For this, we will use the DistributedSampler.

The init_process_group and destroy_process_group are used to initialize and quit the distributed training mods. The init_process_group function should be called at the beginning of the training script to initialize a process group for each process in the distributed setup, and destroy_process_group should be called at the end of the training script to destroy a given process group and release its resources.

The following code in listing below illustrates how these new components are used to implement DDP training for the NeuralNetwork model we implemented earlier.

```
def ddp_setup(rank, world_size):
  os.environ["MASTER_ADDR"] = "localhost"
  os.environ["MASTER_PORT"] = "12345"
  init_process_group(backend="nccl",rank=rank,world_size=world_size)
  torch.cuda.set_device(rank)

def prepare_dataset():
  ...
  train_loader = DataLoader(
    dataset=train_ds,
    batch_size=2,
    shuffle=False,
    pin_memory=True,
    drop_last=True,
    sampler=DistributedSampler(train_ds)
  )
  return train_loader, test_loader
```

```
def main(rank, world_size, num_epochs):
  ddp_setup(rank, world_size)
  train_loader, test_loader = prepare_dataset()
  model = NeuralNetwork(num_inputs=2, num_outputs=2)
  model.to(rank)
  optimizer = torch.optim.SGD(model.parameters(), lr=0.5)
  model = DDP(model, device_ids=[rank])
  for epoch in range(num_epochs):
    for features, labels in train_loader:
      features, labels = features.to(rank), labels.to(rank)
      ...
      print(f"[GPU{rank}] Epoch: {epoch+1:03d}/{num_epochs:03d}"
      f" | Batchsize {labels.shape[0]:03d}"
      f" | Train/Val Loss: {loss:.2f}")
  model.eval()
  train_acc = compute_accuracy(model, train_loader, device=rank)
  print(f"[GPU{rank}] Training accuracy", train_acc)
  test_acc = compute_accuracy(model, test_loader, device=rank)
  print(f"[GPU{rank}] Test accuracy", test_acc)
  destroy_process_group()

if __name__ == "__main__":
  print("Number of GPUs available:", torch.cuda.device_count())
  torch.manual_seed(123)
  num_epochs = 3
  world_size = torch.cuda.device_count()
  mp.spawn(main, args=(world_size, num_epochs), nprocs=world_size)
```

Before we run the code from listing A.13, here is a summary of how it works, in addition to the annotations above. We have a **name** == "**main**" clause at the bottom containing code that is executed when we run the code as a Python script instead of importing it as a module. This code first prints the number of available GPUs using torch.cuda.device_count(), sets a random seed for reproducibility and then spawns new processes using PyTorch's multiprocesses.spawn function. Here, the spawn function launches one process per GPU setting nproces=world_size, where the world size is the number of available GPUs. This spawn function launches the code in the main function we define in the same script with some additional arguments provided via args. Note that the main function has a rank argument that we don't include in the mp.spawn() call. That's because the rank, which refers to the process ID we use as the GPU ID, is already passed automatically.

The main function sets up the distributed environment via ddp_setup -- another function we defined, loads the training and test sets, sets up the model, and carries out the training. Compared to the single-GPU training, we now transfer the model and data to the target device via .to(rank), which we use to refer to the GPU

device ID. Also, we wrap the model via DDP, which enables the synchronization of the gradients between the different GPUs during training. After the training finishes and we evaluate the models, we use destroy_process_group() to cleanly exit the distributed training and free up the allocated resources.

Earlier, we mentioned that each GPU will receive a different subsample of the training data. To ensure this, we set sampler=DistributedSampler(train_ds) in the training loader.

The last function to discuss is ddp_setup. It sets the main node's address and port to allow for communication between the different processes, initializes the process group with the NCCL backend (designed for GPU-to-GPU communication), and sets the rank (process identifier) and world size (total number of processes). Finally, it specifies the GPU device corresponding to the current model training process rank.

If you wish to restrict the number of GPUs used for training on a multi-GPU machine, the simplest way is to use the CUDA_VISIBLE_DEVICES environment variable. To illustrate this, suppose your machine has multiple GPUs, and you only want to use one GPU, for example, the GPU with index 0. Instead of python some_script.py, you can run the code from the terminal as follows:

```
(base) fatnaoui@hp-probook-640-g2:~$ CUDA_VISIBLE_DEVICES=0 python some_script.p
y
```

Or, if your machine has four GPUs and you only want to use the first and third GPU, you can use

```
(base) fatnaoui@hp-probook-640-g2:~$ CUDA_VISIBLE_DEVICES=0,2 python some_script
.py
```

Setting CUDA_VISIBLE_DEVICES in this way is a simple and effective way to manage GPU allocation without modifying your PyTorch scripts.

A.10 Summary

- PyTorch is an open-source library that consists of three core components: a tensor library, automatic differentiation functions, and deep learning utilities.
- PyTorch's tensor library is similar to array libraries like NumPy
- In the context of PyTorch, tensors are array-like data structures to represent scalars, vectors, matrices, and higher-dimensional arrays.
- PyTorch tensors can be executed on the CPU, but one major advantage of PyTorch's tensor format is its GPU support to accelerate computations.
- The automatic differentiation (autograd) capabilities in PyTorch allow us to conveniently train neural networks using backpropagation without manually deriving gradients.
- The deep learning utilities in PyTorch provide building blocks for creating custom deep neural networks.
- PyTorch includes Dataset and DataLoader classes to set up efficient data loading pipelines.
- It's easiest to train models on a CPU or single GPU.
- Using DistributedDataParallel is the simplest way in PyTorch to accelerate the training if multiple GPUs are available.