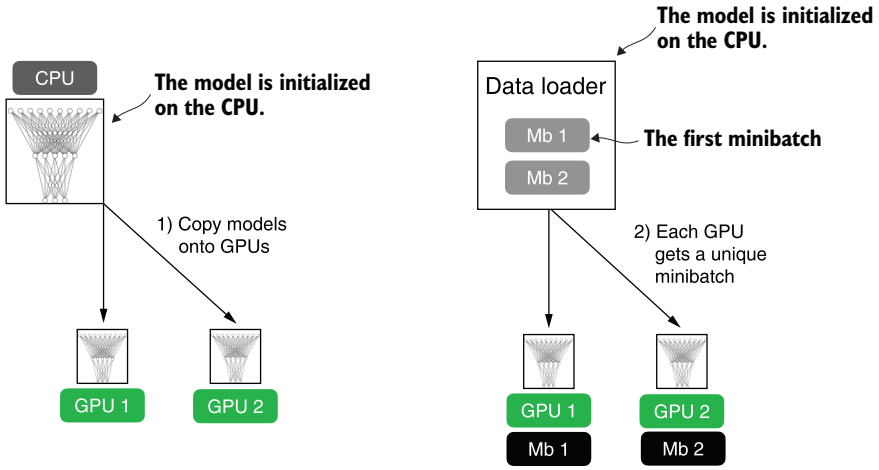

Figure A.12 The model and data transfer in DDP involves two key steps. First, we create a copy of the model on each of the GPUs. Then we divide the input data into unique minibatches that we pass on to each model copy.

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, the model 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, as illustrated in figure A.13.

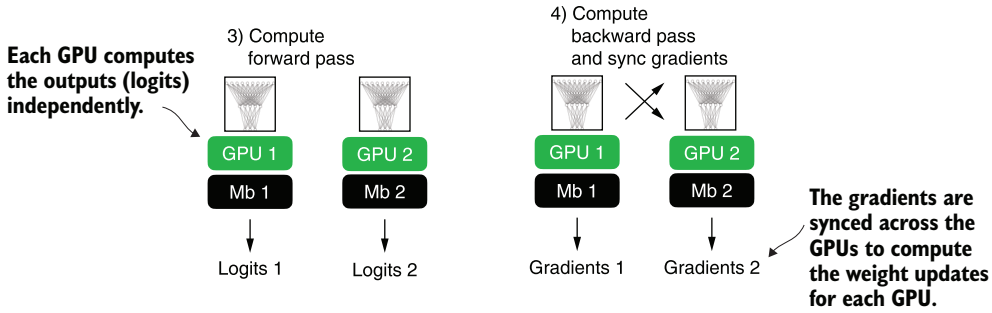

Figure A.13 The forward and backward passes in DDP are executed independently on each GPU with its corresponding data subset. Once the forward and backward passes are completed, gradients from each model replica (on each GPU) are synchronized across all GPUs. This ensures that every model replica has the same updated weights.

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.

> **NOTE** 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. DDP needs to spawn multiple processes, and each process should have its own Python interpreter instance.

Let's now see how this works in practice. For brevity, I focus on the core parts of the code that need to be adjusted for DDP training. However, readers who want to run the code on their own multi-GPU machine or a cloud instance of their choice should use the standalone script provided in this book's GitHub repository at https://github.com/rasbt/LLMs-from-scratch.

First, we import a few additional submodules, classes, and functions for distributed training PyTorch, as shown in the following listing.

Listing A.12 PyTorch utilities for distributed training

```
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
```

Before we dive deeper into the changes to make the training compatible with DDP, let's briefly go over the rationale and usage for these newly imported utilities that we need alongside the `DistributedDataParallel` class.

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`.

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

Listing A.13 Model training with the `DistributedDataParallel` strategy

```
def ddp_setup(rank, world_size):
    os.environ["MASTER_ADDR"] = "localhost"
```

Address of the main node

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

def prepare_dataset():
    # insert dataset preparation code
    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):
        train_loader.sampler.set_epoch(epoch)
        model.train()
        for features, labels in train_loader:
            features, labels = features.to(rank), labels.to(rank)
            # insert model prediction and backpropagation code
            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)
```

world_size is the number of GPUs to use.

Any free port on the machine

nccl stands for NVIDIA Collective Communication Library.

rank refers to the index of the GPU we want to use.

Sets the current GPU device on which tensors will be allocated and operations will be performed

DistributedSampler takes care of the shuffling now.

Enables faster memory transfer when training on GPU

Splits the dataset into distinct, non-overlapping subsets for each process (GPU)

The main function running the model training

rank is the GPU ID

Cleans up resource allocation

Launches the main function using multiple processes, where nprocs=world_size means one process per GPU.