```
tensor_1 = torch.tensor([1., 2., 3.])
tensor_2 = torch.tensor([4., 5., 6.])
print(tensor_1 + tensor_2)
```

This outputs

```
tensor([5., 7., 9.])
```

We can now use the `.to()` method. This method is the same as the one we use to change a tensor's datatype (see 2.2.2) 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)
```

The output is

```
tensor([5., 7., 9.], device='cuda:0')
```

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 can specify which GPU you'd like to transfer the tensors to. You do so by indicating the device ID in the transfer command. For instance, you can use `.to("cuda:0")`, `.to("cuda:1")`, and so on.

However, all tensors must be on the same device. Otherwise, the computation will fail, where one tensor resides on the CPU and the other on the GPU:

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

The results are

```
RuntimeError      Traceback (most recent call last)
<ipython-input-7-4ff3c4d20fc3> in <cell line: 2>()
      1 tensor_1 = tensor_1.to("cpu")
----> 2 print(tensor_1 + tensor_2)
RuntimeError: Expected all tensors to be on the same device, but found at
least two devices, cuda:0 and cpu!
```

In sum, we only need to transfer the tensors onto the same GPU device, and PyTorch will handle the rest.

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 step requires only changing three lines of code, as shown in the following listing.

Listing A.11 A training loop on a GPU

```
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()
    # Insert optional model evaluation code
```

Defines a device variable that defaults to a GPU

Transfers the model onto the GPU

Transfers the data onto the GPU

Running the preceding code will output the following, similar to the results obtained on the CPU (section A.7):

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

We can use `.to("cuda")` instead of `device = torch.device("cuda")`. Transferring a tensor to `"cuda"` instead of `torch.device("cuda")` works as well and is shorter (see section A.9.1). We can also modify the statement, which will make the same code executable on a CPU if a GPU is not available. This is 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 here, we probably won't see a speedup due to the memory transfer cost from CPU to GPU. However, we can expect a significant speedup when training deep neural networks, especially LLMs.

PyTorch on macOS

On an Apple Mac with an Apple Silicon chip (like the M1, M2, M3, or newer models) instead of a computer with an Nvidia GPU, you can change

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

to

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

to take advantage of this chip.

Exercise A.4

Compare the run time of matrix multiplication on a CPU to a GPU. At what matrix size do you begin to see the matrix multiplication on the GPU being faster than on the CPU? Hint: use the `%timeit` command in Jupyter to compare the run time. For example, given matrices `a` and `b`, run the command `%timeit a @ b` in a new notebook cell.

A.9.3 Training with multiple GPUs

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 fine-tune the model parameters and architecture.

> **NOTE** For this book, access to or use of multiple GPUs is not required. This section is included for those interested in how multi-GPU computing works in PyTorch.

Let's begin with 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 in figure A.12.

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