Chapter 10. Building Neural Networks with PyTorch

PyTorch is a powerful open source deep learning library developed by Facebook's AI Research lab (FAIR, now called Meta AI). It is the Python successor of the Torch library, originally written in the Lua programming language. With PyTorch, you can build all sorts of neural network models and train them at scale using GPUs (or other hardware accelerators, as we will see). In many ways it is similar to NumPy, except it also supports hardware acceleration and autodiff (see Chapter 9), and includes optimizers and ready-to-use neural net components.

When PyTorch was released in 2016, Google's TensorFlow library was by far the most popular: it was fast, it scaled well, and it could be deployed across many platforms. But its programming model was complex and static, making it difficult to use and debug. In contrast, PyTorch was designed from the ground up to provide a more flexible, Pythonic approach to building neural networks. In particular, as you will see, it uses dynamic computation graphs (also known as define-by-run), making it intuitive and easy to debug. PyTorch is also beautifully coded and documented, and focuses on its core task: making it easy to build and train high performance neural networks. Last but not least, it leans strongly into the open source culture and benefits from an enthusiastic and dedicated community, and a rich ecosystem. In September 2022, PyTorch's governance was even transferred to the PyTorch Foundation, a subsidiary of the Linux Foundation. All these qualities resonated well with researchers: PyTorch quickly became the most used framework in academia, and once a majority of deep learning papers were based on PyTorch, a large part of the industry was gradually converted as well.[1]

In this chapter, you will learn how to train, evaluate, fine-tune, optimize, and save neural nets with PyTorch. We will start by getting familiar with the core building blocks of PyTorch, namely tensors and autograd, next we will test the waters by building and training a simple linear regression model, and then we will upgrade this model to a multilayer neural network, first for regression, then for classification. Along the way, we will see how to build custom neural networks with multiple inputs or outputs. Finally, we will discuss how to automatically fine-tune hyperparameters using the Optuna library, and how to optimize and export your models. Hop on board, we're diving into deep learning!

NOTE

Colab runtimes come with a recent version of PyTorch preinstalled. However, if you prefer to install it on your own machine, please see the installation instructions at *https://homl.info/install-p*: this involves installing Python, many libraries, and a GPU driver (if you have one).

PyTorch Fundamentals

The core data structure of PyTorch is the *tensor*.[2] It's a multidimensional array with a shape and a data type, used for numerical computations. Isn't that exactly like a NumPy array? Well, yes, it is! But a tensor also has two extra features: it can live on a GPU (or other hardware accelerators, as we will see), and it supports auto-differentiation. Every neural network we will build from now on will input and output tensors (much like Scikit-Learn models input and output NumPy arrays). So let's start by looking at how to create and manipulate tensors.

PyTorch Tensors

First, let's import the PyTorch library:

```
>>> import torch
```

Next you can create a PyTorch tensor much like you would create a NumPy array. For example, let's create a 2 × 3 array:

```
>>> X = torch.tensor([[1.0, 4.0, 7.0], [2.0, 3.0, 6.0]])
>>> X
tensor([[1., 4., 7.],
        [2., 3., 6.]])
```

Just like a NumPy array, a tensor can contain floats, integers, booleans, or complex numbers—just one data type per tensor. If you initialize a tensor with values of different types, then the most general one will be selected (i.e., complex > float > integer > bool). You can also select the data type explicitly when creating the tensor, for example `dtype=torch.float16` for 16-bit floats. Note that tensors of strings or objects are not supported.

You can get a tensor's shape and data type like this:

```
>>> X.shape
torch.Size([2, 3])
>>> X.dtype
torch.float32
```

Indexing works just like for NumPy arrays:

```
>>> X[0, 1]
tensor(4.)
>>> X[:, 1]
tensor([4., 3.])
```

You can also run all sorts of computations on tensors, and the API is conveniently similar to NumPy's: for example, there's `torch.abs()`, `torch.cos()`, `torch.exp()`, `torch.max()`,

`torch.mean()`, `torch.sqrt()`, and so on. PyTorch tensors also have methods for most of these operations, so you can write `X.exp()` instead of `torch.exp(X)`. Let's try a few operations:

```
>>> 10 * (X + 1.0)  # itemwise addition and multiplication
tensor([[20., 50., 80.],
        [30., 40., 70.]])
>>> X.exp()  # itemwise exponential
tensor([[   2.7183,   54.5982, 1096.6332],
        [   7.3891,   20.0855,  403.4288]])
>>> X.mean()
tensor(3.8333)
>>> X.max(dim=0)  # max values along dimension 0 (i.e., max value per column)
torch.return_types.max(values=tensor([2., 4., 7.]), indices=tensor([1, 0, 0]))
>>> X @ X.T  # matrix transpose and matrix multiplication
tensor([[66., 56.],
        [56., 49.]])
```

NOTE

PyTorch prefers the argument name `dim` in operations such as `max()`, but it also supports `axis` (as in NumPy or Pandas).

You can also convert a tensor to a NumPy array using the `numpy()` method, and create a tensor from a NumPy array:

```
>>> import numpy as np
>>> X.numpy()
array([[1., 4., 7.],
       [2., 3., 6.]], dtype=float32)
>>> torch.tensor(np.array([[1., 4., 7.], [2., 3., 6.]]))
tensor([[1., 4., 7.],
        [2., 3., 6.]], dtype=torch.float64)
```

Notice that the default precision for floats is 32 bits in PyTorch, whereas it's 64 bits in NumPy. It's generally better to use 32 bits in deep learning because this takes half the RAM and speeds up computations, and neural nets do not actually need the extra precision offered by 64-bit floats. So when calling the `torch.tensor()` function to convert a NumPy array to a tensor, it's best to specify `dtype=torch.float32`. Alternatively, you can use `torch.FloatTensor()` which automatically converts the array to 32 bits:

```
>>> torch.FloatTensor(np.array([[1., 4., 7.], [2., 3., 6.]]))
tensor([[1., 4., 7.],
        [2., 3., 6.]])
```

TIP

Both `torch.tensor()` and `torch.FloatTensor()` make a copy of the given NumPy array. If you prefer, you can use `torch.from_numpy()` which creates a tensor on the CPU that just uses the NumPy array's data directly, without copying it. But beware: modifying the NumPy array will also modify the tensor, and vice versa.

You can also modify a tensor in place using indexing and slicing, as with a NumPy array:

```
>>> X[:, 1] = -99
>>> X
tensor([[  1., -99.,   7.],
        [  2., -99.,   6.]])
```

PyTorch's API provides many in place operations, such as `abs_()`, `sqrt_()`, and `zero_()`, which modify the input tensor directly: they can sometimes save some memory and speed up your models. For example, the `relu_()` method applies the ReLU activation function in place by replacing all negative values with 0s:

```
>>> X.relu_()
>>> X
tensor([[1., 0., 7.],
        [2., 0., 6.]])
```

TIP

PyTorch's in place operations are easy to spot at a glance because their name always ends with an underscore. With very few exceptions (e.g., `zero_()`), removing the underscore gives you the regular operation (e.g., `abs_()` is in place, `abs()` is not).

We will cover many more operations as we go, but now let's look at how to use hardware acceleration to make computations much faster.

Hardware Acceleration

PyTorch tensors can be copied easily to the GPU, assuming your machine has a compatible GPU, and you have the required libraries installed. On Colab, all you need to do is ensure that you are using a GPU runtime: for this, go to the Runtime menu and select "Change runtime type", then make sure a GPU is selected (e.g., an Nvidia T4 GPU). The GPU runtime will automatically have the appropriate PyTorch library installed—compiled with GPU support—as well as the appropriate GPU drivers and any other required library (e.g., Nvidia's CUDA and cuDNN libraries).[3] If you prefer to run the code on your own machine, you will need to ensure that you have all the drivers and libraries required. Please follow the instructions at *https://homl.info/install-p*.

PyTorch has excellent support for Nvidia GPUs, as well as several other hardware accelerators:

- Apple's *Metal Performance Shaders* (MPS) to accelerate computations on Apple silicon such as the M1, M2, and later chips, as well as some Intel Macs with a compatible GPU.
- AMD Instinct accelerators and AMD Radeon GPUs, through the ROCm software stack, or via DirectML on Windows.
- Intel GPUs and CPUs on Linux and Windows via Intel's oneAPI.
- Google TPUs via the `torch_xla` library.

WARNING

Deep learning generally requires a *lot* of compute power, especially once we start diving into computer vision and natural language processing, in the following chapters. You will need a reasonably powerful machine, but most importantly you will need a hardware accelerator (or several). If you don't have one, you can try using Colab or Kaggle, they offer runtimes with free GPUs. Or consider using other cloud services. Otherwise, prepare to be very, very patient.

Let's check whether PyTorch can access an Nvidia GPU or Apple's MPS, otherwise let's fall back to the CPU:

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

On a Colab GPU Runtime, `device` will be equal to `"cuda"`. Now let's create a tensor on that GPU. To do that, one option is to create the tensor on the CPU, then copy it to the GPU using the `to()` method:

```
>>> M = torch.tensor([[1., 2., 3.], [4., 5., 6.]])
>>> M = M.to(device)
```

TIP

The `cpu()` and `cuda()` methods are short for `to("cpu")` and `to("cuda")`, respectively.

You can always tell which device a tensor lives on by looking at its `device` attribute:

```
>>> M.device
device(type='cuda', index=0)
```

Alternatively, we can create the tensor directly on the GPU using the `device` argument:

```
>>> M = torch.tensor([[1., 2., 3.], [4., 5., 6.]], device=device)
```

TIP

If you have multiple Nvidia GPUs, you can refer to the desired GPU by appending the GPU index: `"cuda:0"` (or just `"cuda"`) for GPU #0, `"cuda:1"` for GPU #1, and so on.

Once the tensor is on the GPU, we can run operations on it normally, and they will all take place on the GPU:

```
>>> R = M @ M.T  # run some operations on the GPU
>>> R
tensor([[14., 32.],
        [32., 77.]], device='cuda:0')
```

Note that the result `R` also lives on the GPU. This means we can perform multiple operations on the GPU without having to transfer data back and forth between the CPU and the GPU. This is crucial in deep learning because data transfer between devices can often become a performance bottleneck.

How much does a GPU accelerate the computations? Well it depends on the GPU, of course: the more expensive ones are dozens of times faster than the cheap ones. But speed alone is not the only important factor: the data throughput is also crucial, as we just saw. If your model is compute heavy (e.g., a very deep neural net), the GPU's speed and amount of RAM will typically matter most, but if it is a shallower model, then pumping the training data into the GPU might become the bottleneck. Let's run a little test to compare the speed of a matrix multiplication running on the CPU versus the GPU:[4]

```
>>> M = torch.rand((1000, 1000))  # on the CPU
>>> %timeit M @ M.T
16.2 ms ± 3.24 ms per loop (mean ± std. dev. of 7 runs, 100 loops each)
>>> M = torch.rand((1000, 1000), device="cuda")  # on the GPU
>>> %timeit M @ M.T
605 µs ± 13.2 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Wow! The GPU gave us a 26× speed boost! And that's just using the free Nvidia T4 GPU on Colab; imagine the speedup we could get using a more powerful GPU. Now try playing around with the matrix size: you will notice that the speedup is much less impressive on smaller matrices (e.g., it's just 2× for 100 × 100 matrices). That's because GPUs work by breaking large operations into smaller operations and running them in parallel across thousands of cores. If the

task is small, it cannot be broken up into that many pieces, and the performance gain is therefore smaller. In fact, when running many tiny tasks, it can sometimes be faster to just run the operations on the CPU.

All right, now that we've seen what tensors are and how to use them on the CPU or the GPU, let's look at PyTorch's auto-differentiation feature.

Autograd

PyTorch comes with an efficient implementation of reverse-mode auto-differentiation (introduced in Chapter 9 and detailed in Appendix A), called *autograd*, which stands for automatic gradients. It is quite easy to use. For example, consider a simple function, f(x) = x^2. Differential calculus tells us that the derivative of this function is $f'(x) = 2x$. If we evaluate f(5) and f'(5), we get 25 and 10, respectively. Let's see if PyTorch agrees:

```
>>> x = torch.tensor(5.0, requires_grad=True)
>>> f = x ** 2
>>> f
tensor(25., grad_fn=<PowBackward0>)
>>> f.backward()
>>> x.grad
tensor(10.)
```

Great, we got the correct results: `f` is 25, and `x.grad` is 10! Note that the `backward()` function automatically computed the gradient f'(x) at the same point x = 5.0. Let's go through this code line by line:

- First, we created a tensor `x`, equal to 5.0, and we told PyTorch that it's a variable (not a constant) by specifying `requires_grad=True`. Knowing this, PyTorch will automatically keep track of all operations involving `x`: this is needed because PyTorch must capture the computation graph in order to run backprop on it and obtain the derivative of `f` with regard to `x`. In this computation graph, the tensor `x` is a *leaf node*.
- Then we compute `f = x ** 2`. The result is a tensor equal to 25.0, the square of 5.0. But wait, there's more to it: `f` also carries a `grad_fn` attribute which represents the operation that created this tensor (`**`, power, hence the name `PowBackward0`), and which tells PyTorch how to backpropagate the gradients through this particular operation. This `grad_fn` attribute is how PyTorch keeps track of the computation graph.
- Next, we call `f.backward()`: this backpropagates the gradients through the computation graph, starting with `f`, and all the way back to the leaf nodes (just `x` in this case).
- Lastly, we can just read the `x` tensor's `grad` attribute, which was computed during backprop: this gives us the derivative of `f` with regard to `x`. Ta-da!

PyTorch creates a new computation graph on the fly during each forward pass, as the operations are executed. This allows PyTorch to support very dynamic models containing loops and conditionals.

WARNING

The way PyTorch accumulates gradients in each variable's `grad` attribute can be surprising at first, especially coming from TensorFlow or JAX. In these frameworks, computing the gradients of `f` with regard to `x` just returns the gradients, without affecting `x`. In PyTorch, if you call `backward()` on a tensor, it will accumulate the gradients in every variable that was used to compute it. So if you call `backward()` on two tensors `t1` and `t2` that both used the same variable `v`, then `v.grad` will be the sum of their gradients.

After computing the gradients, you generally want to perform a gradient descent step by subtracting a fraction of the gradients from the model variables (at least when training a neural network). In our simple example, running gradient descent will gradually push `x` toward 0, since that's the value that minimizes $f(x) = x^2$. To do a gradient descent step, you must temporarily disable gradient tracking since you don't want to track the gradient descent step itself in the computation graph (in fact, PyTorch would raise an exception if you tried to run an in-place operation on a tracked variable). This can be done by placing the gradient descent step inside a `torch.no_grad()` context, like this:

```
learning_rate = 0.1
with torch.no_grad():
    x -= learning_rate * x.grad  # gradient descent step
```

The variable `x` gets decremented by 0.1 * 10.0 = 1.0, down from 5.0 to 4.0.

Another way to avoid gradient computation is to use the variable's `detach()` method: this creates a new tensor detached from the computation graph, with `requires_grad=False`, but still pointing to the same data in memory. You can then update this detached tensor:

```
x_detached = x.detach()
x_detached -= learning_rate * x.grad
```

Since `x_detached` and `x` share the same memory, modifying `x_detached` also modifies `x`.

The `detach()` method can be handy when you need to run some computation on a tensor without affecting the gradients (e.g., for evaluation or logging), or when you need fine-grained control over which operations should contribute to gradient computation. Using `no_grad()` is generally preferred when performing inference or doing a gradient descent step, as it provides a convenient context-wide method to disable gradient tracking.

Lastly, before you repeat the whole process (forward pass + backward pass + gradient descent step), it's essential to zero out the gradients of every model parameter (you don't need a `no_grad()` context for this since the gradient tensor has `requires_grad=False`):

```
    x.grad.zero_()
```

WARNING

If you forget to zero out the gradients at each training iteration, the `backward()` method will just accumulate them, causing incorrect gradient descent updates. Since there won't be any explicit error, just low performance (and perhaps infinite or NaN values), this issue may be hard to debug.

Putting everything together, the whole training loop looks like this:

```
learning_rate = 0.1
x = torch.tensor(5.0, requires_grad=True)
for iteration in range(100):
    f = x ** 2  # forward pass
    f.backward()  # backward pass
    with torch.no_grad():
        x -= learning_rate * x.grad  # gradient descent step

    x.grad.zero_()  # reset the gradients
```

If you want to use in-place operations to save memory and speed up your models a bit by avoiding unnecessary copy operations, you have to be careful: in-place operations don't always play nicely with autograd. Firstly, as we saw earlier, you cannot apply an in-place operation to a leaf node (i.e., a tensor with `requires_grad=True`), as PyTorch wouldn't know where to store the computation graph. For example `x.cos_()` or `x += 1` would cause a `RuntimeError`. Secondly, consider the following code, which computes z(t) = exp(t) + 1 at t = 2 and then tries to compute the gradients:

```
t = torch.tensor(2.0, requires_grad=True)
z = t.exp()  # this is an intermediate result
z += 1  # this is an in-place operation
z.backward()  # ⚠️ RuntimeError!
```

Oh no! Although `z` is computed correctly, the last line causes a `RuntimeError`, complaining that "one of the variables needed for gradient computation has been modified by an in-place operation". Indeed, the intermediate result `z = t.exp()` was lost when we ran the in-place operation `z += 1`, so when the backward pass reached the exponential operation, the gradients could not be computed. A simple fix is to replace `z += 1` with `z = z + 1`. It looks similar, but it's no longer an in-place operation: a new tensor is created and assigned to the same variable, but the original tensor is unchanged and recorded in the computation graph of the final tensor.

Surprisingly, if you replace `exp()` with `cos()` in the previous code example, the gradients will be computed correctly: no error! Why is that? Well, the outcome depends on the way each operation is implemented:

- Some operations—such as `exp()`, `relu()`, `rsqrt()`, `sigmoid()`, `sqrt()`, `tan()`, and `tanh()`—save their outputs in the computation graph during the forward pass, then use these outputs to compute the gradients during the backward pass.[5] This means that you must not modify such an operation's output in place, or you will get an error during the backward pass (as we just saw).
- Other operations—such as `abs()`, `cos()`, `log()`, `sin()`, `square()`, and `var()`—save their inputs instead of their output.[6] Such an operation doesn't care if you modify its output in place, but you must not modify its inputs in place before the backward pass (e.g., to compute something else based on the same inputs).
- Some operations—such as `max()`, `min()`, `norm()`, `prod()`, `sgn()`, and `std()`—save both the inputs and the outputs, so you must not modify either of them in place before the backward pass.
- Lastly, a few operations—such as `ceil()`, `floor()`, `mean()`, `round()`, and `sum()`—save neither their inputs nor their outputs.[7] You can safely modify them in place.

TIP

Implement your models first without any in place operations, then if you need to save some memory or speed up your model a bit, you can try converting some of the most costly operations to their in place counterparts. Just make sure that your model still outputs the same result for a given input, and also make sure you don't modify in place a tensor needed for backprop (you will get a `RuntimeError` in this case).

OK, let's step back a bit. We've discussed all the fundamentals of PyTorch: how to create tensors and use them to perform all sorts of computations, how to accelerate the computations with a GPU, and how to use autograd to compute gradients for gradient descent. Great! Now let's apply what we've learned so far by building and training a simple linear regression model with PyTorch.

Implementing Linear Regression

We will start by implementing linear regression using tensors and autograd directly, then we will simplify the code using PyTorch's high-level API, and also add GPU support.

Linear Regression Using Tensors and Autograd

Let's tackle the same California housing dataset as in Chapter 9. I will assume you have already downloaded it using `sklearn.datasets.fetch_california_housing()`, and you have split it into a training set (`X_train` and `y_train`), a validation set (`X_valid` and `y_valid`), and a test set (`X_test` and `y_test`), using `sklearn.model_selection.train_test_split()`. Next, let's convert it to tensors and normalize it. We could use a `StandardScaler` for this, like we did in Chapter 9, but let's just use tensor operations instead, to get a bit of practice:

```
X_train = torch.FloatTensor(X_train)
X_valid = torch.FloatTensor(X_valid)
X_test = torch.FloatTensor(X_test)
means = X_train.mean(dim=0, keepdims=True)
stds = X_train.std(dim=0, keepdims=True)
X_train = (X_train - means) / stds
X_valid = (X_valid - means) / stds
X_test = (X_test - means) / stds
```

Let's also convert the targets to tensors. Since our predictions will be column vectors (i.e., matrices with a single column), we need to ensure that our targets are also column vectors.[8] Unfortunately, the NumPy arrays representing the targets are one-dimensional, so we need to reshape the tensors to column vectors by adding a second dimension of size 1:[9]

```
y_train = torch.FloatTensor(y_train).reshape(-1, 1)
y_valid = torch.FloatTensor(y_valid).reshape(-1, 1)
y_test = torch.FloatTensor(y_test).reshape(-1, 1)
```

Now that the data is ready, let's create the parameters of our linear regression model:

```
torch.manual_seed(42)
n_features = X_train.shape[1]  # there are 8 input features
w = torch.randn((n_features, 1), requires_grad=True)
b = torch.tensor(0., requires_grad=True)
```

We now have a weights parameter `w` (a column vector with one weight per input dimension, in this case 8), and a bias parameter `b` (a single scalar). The weights are initialized randomly, while the bias is initialized to zero. We could have initialized the weights to zero as well in this case, but when we get to neural networks it will be important to initialize the weights randomly to break the symmetry between neurons (as explained in Chapter 9), so we might as well get into the habit now.

WARNING

We called `torch.manual_seed()` to ensure that the results are reproducible. However, PyTorch does not guarantee perfectly reproducible results across different releases, platforms, or devices, so if you do not run the code in this chapter with PyTorch 2.5 on a Colab runtime with an Nvidia T4 GPU, you may get different results. Moreover, since a GPU splits each operation into multiple chunks and runs them in parallel, the order in which these chunks finish may vary across runs, and this may slightly affect the result due to floating point precision errors. These minor differences may compound during training, and lead to very different models. To reduce this risk, you can tell PyTorch to use only deterministic algorithms by calling `torch.use_deterministic_algorithms(True)`. However, deterministic algorithms are often slower than stochastic ones, and some operations don't have a deterministic version at all, so you will get an error if your code tries to use one.

Next, let's train our model, very much like we did in Chapter 4, except we will use autodiff to compute the gradients rather than using a closed-form equation. For now we will use batch gradient descent (BGD), using the full training set at each training step:

```
learning_rate = 0.4
n_epochs = 20
for epoch in range(n_epochs):
    y_pred = X_train @ w + b
    loss = ((y_pred - y_train) ** 2).mean()
    loss.backward()
    with torch.no_grad():
        b -= learning_rate * b.grad
        w -= learning_rate * w.grad
        b.grad.zero_()
        w.grad.zero_()
    print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {loss.item()}")
```

Let's walk through this code:

- First we define the `learning_rate` hyperparameter. You can experiment with different values to find a value that converges fast and gives a precise result.
- Next, we run 20 epochs. We could implement early stopping to find the right moment to stop and avoid overfitting, like we did in Chapter 4, but we will keep things simple for now.
- Next, we run the forward pass: we compute the predictions `y_pred`, and the mean squared error `loss`.
- Then we run `loss.backward()` to compute the gradients of the loss with regard to every model parameter. This is autograd in action.
- Next, we use the gradients `b.grad` and `w.grad` to perform a gradient descent step. Notice that we're running this code inside a `with torch.no_grad()` context, as discussed earlier.
- Once we've done the gradient descent step, we reset the gradients to zero (very important!).
- Lastly, we print the epoch number and the current loss at each epoch. The `item()` method extracts the value of a scalar tensor.

And that's it, if you run this code, you should see the training loss going down like this:

```
Epoch 1/20, Loss: 16.158458709716797
Epoch 2/20, Loss: 4.879374027252197
Epoch 3/20, Loss: 2.255225896835327
[...]
Epoch 20/20, Loss: 0.5684100389480591
```

Congratulations, you just trained your first model using PyTorch! You can now use the model to make predictions for some new data `X_new` (which must be represented as a PyTorch tensor). For example, let's make predictions for the first three instances in the test set:

```
>>> X_new = X_test[:3]  # pretend these are new instances
>>> with torch.no_grad():
...     y_pred = X_new @ w + b  # use the trained parameters to make predictions
...
>>> y_pred
tensor([[0.8916],
        [1.6480],
        [2.6577]])
```

TIP

It's best to use a `with torch.no_grad()` context during inference: PyTorch will consume less RAM and run faster since it won't have to keep track of the computation graph.

Implementing linear regression using PyTorch's low-level API wasn't too hard, but using this approach for more complex models would get really messy and difficult. So PyTorch offers a higher-level API to simplify all this. Let's rewrite our model using this higher-level API.

Linear Regression Using PyTorch's High-Level API

PyTorch provides an implementation of linear regression in the `torch.nn.Linear` class, so let's use it:

```
import torch.nn as nn  # by convention, this module is usually imported this way

torch.manual_seed(42)  # to get reproducible results
model = nn.Linear(in_features=n_features, out_features=1)
```

The `nn.Linear` class (short for `torch.nn.Linear`) is one of many *modules* provided by PyTorch. Each module is a subclass of the `nn.Module` class. To build a simple linear regression model, a single `nn.Linear` module is all you need. However, for most neural networks you will need to assemble many modules, as we will see later in this chapter, so you can think of modules as math LEGO® bricks. Many modules contain model parameters. For example, the `nn.Linear` module contains a `bias` vector (with one bias term per neuron), and a `weight` matrix (with one row per neuron and one column per input dimension, which is the transpose of the weight matrix we used earlier and in Equation 9-2). Since our model has a single neuron (because `out_features=1`), the `bias` vector contains a single bias term, and the `weight` matrix contains a single row. These parameters are accessible directly as attributes of the `nn.Linear` module:

```
>>> model.bias
Parameter containing:
tensor([0.3117], requires_grad=True)
```

```
>>> model.weight
Parameter containing:
tensor([[ 0.2703,  0.2935, -0.0828,  0.3248, -0.0775,  0.0713, -0.1721,  0.2076]],
       requires_grad=True)
```

Notice that both parameters were automatically initialized randomly (which is why we used `manual_seed()` to get reproducible results). These parameters are instances of the `torch.nn.Parameter` class, which is a subclass of the `tensor.Tensor` class: this means that you can use them exactly like normal tensors. A module's `parameters()` method returns an iterator over all of the module's attributes of type `Parameter`, as well as all the parameters of all its submodules, recursively (if it has any). It does *not* return regular tensors, even those with `requires_grad=True`: that's the main difference between a regular tensor and a `Parameter`:

```
>>> for param in model.parameters():
...     [...]  # do something with each parameter
```

There's also a `named_parameters()` method that returns an iterator over pairs of parameter names and values.

A module can be called just like a regular function. For example, let's make some predictions for the first two instances in the training set (since the model is not trained yet, its parameters are random and the predictions are terrible):

```
>>> model(X_train[:2])
tensor([[-0.4718],
        [ 0.1131]], grad_fn=<AddmmBackward0>)
```

When we use a module as a function, PyTorch internally calls the module's `forward()` method. In the case of the `nn.Linear` module, the `forward()` method computes `X @ self.weight.T + self.bias` (where `X` is the input). That's just what we need for linear regression!

Notice that the result contains the `grad_fn` attribute, showing that autograd did its job and tracked the computation graph while the model was making its predictions.

TIP

If you pass a custom function to a module's `register_forward_hook()` method, it will be called automatically every time the module itself is called. This is particularly handy for logging or debugging. To remove a hook, just call the `remove()` method on the object returned by `register_forward_hook()`. Note that hooks only work if you call the model like a function, not if you call its `forward()` method directly (which is why you should never do that). You can also register functions to run during the backward pass using `register_backward_hook()`.

Now that we have our model, we need to create an optimizer to update the model parameters, and we must also choose a loss function:

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
mse = nn.MSELoss()
```

PyTorch provides a few different optimizers (we will discuss them in the next chapter). Here we're using the simple stochastic gradient descent (SGD) optimizer, which can be used for SGD, mini-batch GD, or batch gradient descent. To initialize it, we must give it the model parameters and the learning rate.

For the loss function, we create an instance of the `nn.MSELoss` class: this is also a module, so we can use it like a function, giving it the predictions and the targets, and it will compute the MSE. The `nn` module contains many other loss functions and other neural net tools, as we will see. Next, let's write a small function to train our model:

```
def train_bgd(model, optimizer, criterion, X_train, y_train, n_epochs):
    for epoch in range(n_epochs):
        y_pred = model(X_train)
        loss = criterion(y_pred, y_train)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
        print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {loss.item()}")
```

Compare this training loop with our earlier training loop: it's very similar, but we're now using higher-level constructs rather than working directly with tensors and autograd. Here are a few things to note:

- In PyTorch, the loss function object is commonly referred to as the *criterion*, to distinguish it from the loss value itself (which is computed at each training iteration using the criterion). In this example, it's the `MSELoss` instance.
- The `optimizer.step()` line corresponds to the two lines that updated `b` and `w` in our earlier code.
- And of course the `optimizer.zero_grad()` line corresponds to the two lines that zeroed out `b.grad` and `w.grad`. Notice that we don't need to use `with torch.no_grad()` here since this is done automatically by the optimizer, inside the `step()` and `zero_grad()` functions.

NOTE

Most people prefer to call `zero_grad()` *before* calling `loss.backward()`, rather than after: this might be a bit safer in case the gradients are non-zero when calling the function, but in general it makes no difference since gradients are automatically initialized to `None`.

Now let's call this function to train our model!

```
>>> train_bgd(model, optimizer, mse, X_train, y_train, n_epochs)
Epoch 1/20, Loss: 4.3378496170043945
Epoch 2/20, Loss: 0.780293345451355
[...]
Epoch 20/20, Loss: 0.5374288558959961
```

All good, the model is trained, you can now use it to make predictions by simply calling it like a function (preferably inside a `no_grad()` context, as we saw earlier):

```
>>> X_new = X_test[:3]  # pretend these are new instances
>>> with torch.no_grad():
...     y_pred = model(X_new)  # use the trained model to make predictions
...
>>> y_pred
tensor([[0.8061],
        [1.7116],
        [2.6973]])
```

These predictions are similar to the ones our previous model made, but not exactly the same: that's because the `nn.Linear` module initializes the parameters slightly differently: it uses a uniform random distribution from to for both the weights and the bias term (we will discuss initialization methods in Chapter 11).

Now that you are familiar with PyTorch's high-level API, you are ready to go beyond linear regression and build a multilayer perceptron (introduced in Chapter 9).

Implementing a Regression MLP

PyTorch provides a helpful `nn.Sequential` module that chains multiple modules: when you call this module with some inputs, it feeds these inputs to the first module, then feeds the output of the first module to the second module, and so on. Most neural networks contain stacks of modules, and in fact many neural networks are just one big stack of modules: this makes the `nn.Sequential` module one of the most useful modules in PyTorch. The MLP we want to build is just that: a simple stack of modules—two hidden layers and one output layer. So let's build it using the `nn.Sequential` module:

```
torch.manual_seed(42)
model = nn.Sequential(
    nn.Linear(n_features, 50),
    nn.ReLU(),
    nn.Linear(50, 40),
    nn.ReLU(),
```

```
    nn.Linear(40, 1)
)
```

Let's go through each layer:

- The first layer must have the right number of inputs for our data: `n_features` (equal to 8 in our case). However, it can have any number of outputs: let's pick 50 (that's a hyperparameter we can tune).
- Next we have a `nn.ReLU` module, which implements the ReLU activation function for the first hidden layer. This module does not contain any model parameters, and it acts itemwise so the shape of its output is equal to the shape of its input.
- The second hidden layer must have the same number of inputs as the output of the previous layer: in this case, 50. However, it can have any number of outputs. It's common to use the same number of output dimensions in all hidden layers, but in this example I used 40 to make it clear that the output of one layer must match the input of the next layer.
- Then again, a `nn.ReLU` module to implement the second hidden layer's activation function.
- Finally, the output layer must have 40 inputs, but this time its number of outputs is not free: it must match the targets' dimensionality. Since our targets have a single dimension, we must have just one output dimension in the output layer.

Now let's train the model just like we did before:

```
>>> learning_rate = 0.1
>>> optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
>>> mse = nn.MSELoss()
>>> train_bgd(model, optimizer, mse, X_train, y_train, n_epochs)
Epoch 1/20, Loss: 5.045480251312256
Epoch 2/20, Loss: 2.0523128509521484
[...]
Epoch 20/20, Loss: 0.565444827079773
```

That's it, you can tell your friends you trained your first neural network with PyTorch! However, we are still using batch gradient descent, computing the gradients over the entire training set at each iteration. This works with small datasets, but if we want to be able to scale up to large datasets and large models, we need to switch to mini-batch gradient descent.

Implementing Mini-Batch Gradient Descent Using DataLoaders

To help implement mini-batch GD, PyTorch provides a class named `DataLoader` in the `torch.utils.data` module. It can efficiently load batches of data of the desired size, and shuffle the data at each epoch if we want it to. The `DataLoader` expects the dataset to be represented as an object with at least two methods: `__len__(self)` to get the number of samples in the dataset, and `__getitem__(self, index)` to load the sample at the given index

(including the target).

In our case, the training set is available in the `X_train` and `y_train` tensors, so we first need to wrap these tensors in a dataset object with the required API. To help with this, PyTorch provides a `TensorDataset` class. So let's build a `TensorDataset` to wrap our training set, and a `DataLoader` to pull batches from this dataset. During training, we want the dataset to be shuffled, so we specify `shuffle=True`:

```
from torch.utils.data import TensorDataset, DataLoader

train_dataset = TensorDataset(X_train, y_train)
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

Now that we have a larger model and we have the tools to train it one batch at a time, it's a good time to start using hardware acceleration. It's really quite simple: we just need to move the model to the GPU, which will move all of its parameters to the GPU RAM, and then at the start of each iteration during training we must copy each batch to the GPU. To move the model, we can just use its `to()` method, just like we did with tensors:

```
torch.manual_seed(42)
model = nn.Sequential([...])  # create the model just like earlier
model = model.to(device)
```

We can also create the loss function and optimizer, as earlier (but using a lower learning rate, such as 0.02).

WARNING

Some optimizers have some internal state, as we will see in Chapter 11. The optimizer will usually allocate its state on the same device as the model parameters, so it's important to create the optimizer *after* you have moved the model to the GPU.

Now let's create a `train()` function to implement mini-batch GD:

```
def train(model, optimizer, criterion, train_loader, n_epochs):
    model.train()
    for epoch in range(n_epochs):
        total_loss = 0.
        for X_batch, y_batch in train_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            loss = criterion(y_pred, y_batch)
            total_loss += loss.item()
            loss.backward()
            optimizer.step()
```

```
            optimizer.zero_grad()

        mean_loss = total_loss / len(train_loader)
        print(f"Epoch {epoch + 1}/{n_epochs}, Loss: {mean_loss:.4f}")
```

At every epoch, the function iterates through the whole training set, one batch at a time, and processes each batch just like earlier. But what about the very first line: `model.train()`? Well, this switches the model and all of its submodules to *training mode*. For now, this makes no difference at all, but it will be important in Chapter 11 when we start using layers that behave differently during training and evaluation (e.g., `nn.Dropout` or `nn.BatchNorm1d`). Whenever you want to use the model outside of training (e.g., for evaluation, or to make predictions on new instances), you must first switch the model to *evaluation mode* by running `model.eval()`. Note that `model.training` holds a boolean that indicates the current mode.

TIP

PyTorch itself does not provide a training loop implementation; you have to build it yourself. As we just saw, it's not that long, and many people enjoy the freedom, clarity, and control this provides. However, if you would prefer to use a well-tested, off-the-shelf training loop with all the bells and whistles you need (such as multi-GPU support), then you can use a library such as PyTorch Lightning, FastAI, Catalyst, or Keras. These libraries are built on top of PyTorch and include a training loop and many other features (Keras supports PyTorch since version 3, and also supports TensorFlow and JAX). Check them out!

Now let's call this `train()` function to train our model on the GPU:

```
>>> train(model, optimizer, mse, train_loader, n_epochs)
Epoch 1/20, Loss: 0.6958
Epoch 2/20, Loss: 0.4480
[...]
Epoch 20/20, Loss: 0.3227
```

It worked great: we actually reached a much lower loss in the same number of epochs! However, you probably noticed that each epoch was much slower. There are two easy tweaks you can make to considerably speed up training:

- If you are using a CUDA device, you should generally set `pin_memory=True` when creating the data loader: this will allocate the data in *page-locked memory* which guarantees a fixed physical memory location in the CPU RAM, and therefore allows direct memory access (DMA) transfers to the GPU, eliminating an extra copy operation that would otherwise be needed. While this could use more CPU RAM since the memory cannot be swapped out to disk, it typically results in significantly faster data transfers and thus faster training. When transferring a tensor to the GPU using its `to()` method, you may also set `non_blocking=True` to avoid blocking the CPU during the data transfer (this only works if `pin_memory=True`).
- The current training loop waits until a batch has been fully processed before it loads the

next batch. You can often speed up training by pre-fetching the next batches on the CPU while the GPU is still working on the current batch. For this, set the data loader's `num_workers` argument to the number of processes you want to use for data loading and preprocessing. The optimal number depends on your platform, hardware, and workload, so you should experiment with different values. You can also tweak the number of batches that each worker pre-fetches by setting the data loader's `prefetch_factor` argument. Note that the overhead of spawning and synchronizing workers can often slow down training rather than speed it up (especially on Windows). In this case, you can try setting `persistent_workers=True` to reuse the same workers across epochs.

OK, time to step back a bit: you know the PyTorch fundamentals (tensors and autograd), you can build neural nets using PyTorch's high-level API, and train them using mini-batch gradient descent, with the help of an optimizer, a criterion, and a data loader. The next step is to learn how to evaluate your model.

Model Evaluation

Let's write a function to evaluate the model. It takes the model and a `DataLoader` for the dataset that we want to evaluate the model on, as well as a function to compute the metric for a given batch, and lastly a function to aggregate the batch metrics (by default, it just computes the mean):

```
def evaluate(model, data_loader, metric_fn, aggregate_fn=torch.mean):
    model.eval()
    metrics = []
    with torch.no_grad():
        for X_batch, y_batch in data_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            metric = metric_fn(y_pred, y_batch)
            metrics.append(metric)
    return aggregate_fn(torch.stack(metrics))
```

Now let's build a `TensorDataset` and a `DataLoader` for our validation set, and pass it to our `evaluate()` function to compute the validation MSE:

```
>>> valid_dataset = TensorDataset(X_valid, y_valid)
>>> valid_loader = DataLoader(valid_dataset, batch_size=32)
>>> valid_mse = evaluate(model, valid_loader, mse)
>>> valid_mse
tensor(0.4193, device='cuda:0')
```

It works fine. But now suppose we want to use the RMSE instead of the MSE (as we saw in Chapter 2, it can be easier to interpret). PyTorch does not have a built-in function for that, but it's easy enough to write:

```
>>> def rmse(y_pred, y_true):
...     return ((y_pred - y_true) ** 2).mean().sqrt()
...
>>> evaluate(model, valid_loader, rmse)
tensor(0.5732, device='cuda:0')
```

But wait a second! The RMSE should be equal to the square root of the MSE; however, when we compute the square root of the MSE that we found earlier, we get a different result:

```
>>> valid_mse.sqrt()
tensor(0.6476, device='cuda:0')
```

The reason is that instead of calculating the RMSE over the whole validation set, we computed it over each batch and then computed the mean of all these batch RMSEs. That's not mathematically equivalent to computing the RMSE over the whole validation set. To solve this, we can use the MSE as our `metric_fn`, and use the `aggregate_fn` to compute the square root of the mean MSE:[10]

```
>>> evaluate(model, valid_loader, mse,
...          aggregate_fn=lambda metrics: torch.sqrt(torch.mean(metrics)))
...
tensor(0.6476, device='cuda:0')
```

That's much better!

Rather than implement metrics yourself, you may prefer to use the TorchMetrics library (made by the same team as PyTorch Lightning), which provides many well-tested *streaming metrics*. A streaming metric is an object that keeps track of a given metric, and can be updated one batch at a time. The TorchMetrics library is not preinstalled on Colab, so we have to run `%pip install torchmetrics`, then we can implement the `evaluate_tm()` function, like this:

```
import torchmetrics

def evaluate_tm(model, data_loader, metric):
    model.eval()
    metric.reset()  # reset the metric at the beginning
    with torch.no_grad():
        for X_batch, y_batch in data_loader:
            X_batch, y_batch = X_batch.to(device), y_batch.to(device)
            y_pred = model(X_batch)
            metric.update(y_pred, y_batch)  # update it at each iteration
    return metric.compute()  # compute the final result at the end
```

Then we can create an RMSE streaming metric, move it to the GPU, and use it to evaluate the

validation set:

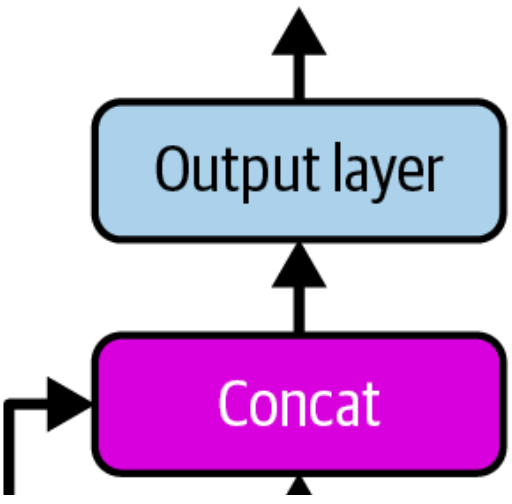
```
>>> rmse = torchmetrics.MeanSquaredError(squared=False).to(device)
>>> evaluate_tm(model, valid_loader, rmse)
tensor(0.6477, device='cuda:0')
```

Sure enough, we get the correct result! Now try updating the `train()` function to evaluate your model's performance during training, both on the training set (during each epoch) and on the validation set (at the end of each epoch). As always, if the performance on the training set is much better than on the validation set, your model is probably overfitting the training set, or there is a bug, such as a data mismatch between the training set and the validation set. This is easier to detect if you plot and analyze the learning curves, much like we did in Chapter 4. For this you can use Matplotlib, or a visualization tool such as TensorBoard (see the notebook for an example).

Now you know how to build, train, and evaluate a regression MLP using PyTorch, and how to use the trained model to make predictions. Great! But so far we have only looked at simple sequential models, composed of a sequence of linear layers and ReLU activation functions. How would you build a more complex, nonsequential model? For this, we will need to build custom modules.

Building Nonsequential Models Using Custom Modules

One example of a nonsequential neural network is a *Wide & Deep* neural network. This neural network architecture was introduced in a 2016 paper by Heng-Tze Cheng et al.[11] It connects all or part of the inputs directly to the output layer, as shown in Figure 10-1. This architecture makes it possible for the neural network to learn both deep patterns (using the deep path) and simple rules (through the short path). The short path can also be used to provide manually engineered features to the neural network. In contrast, a regular MLP forces all the data to flow through the full stack of layers; thus, simple patterns in the data may end up being distorted by this sequence of transformations.



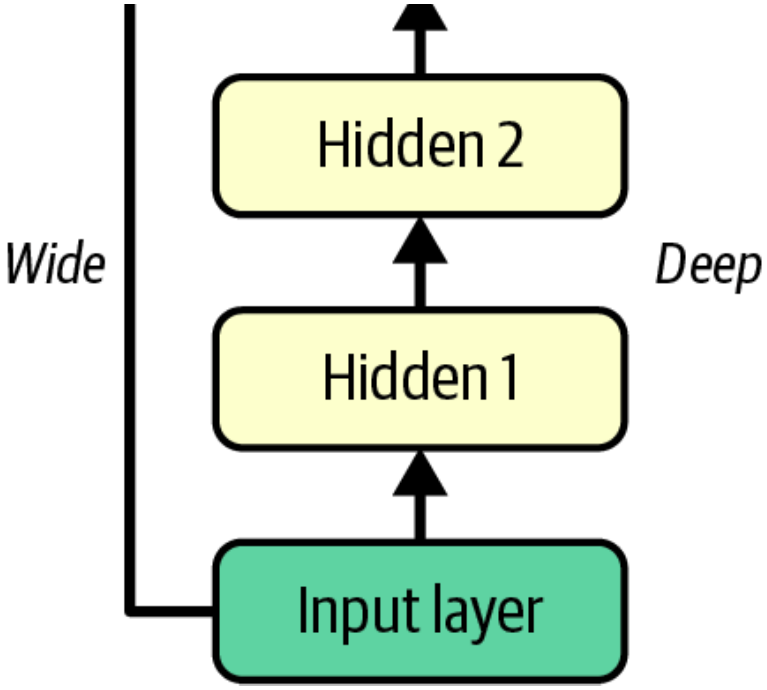


Figure 10-1. Wide & Deep neural network

Let's build such a neural network to tackle the California housing dataset. Because this wide and deep architecture is nonsequential, we have to create a custom module. It's easier than it sounds: just create a class derived from `torch.nn.Module`, then create all the layers you need in the constructor (after calling the base class's `__init__()` method), and define how these layers should be used by the module in the `forward()` method:

```
class WideAndDeep(nn.Module):
    def __init__(self, n_features):
        super().__init__()
        self.deep_stack = nn.Sequential(
            nn.Linear(n_features, 50), nn.ReLU(),
            nn.Linear(50, 40), nn.ReLU(),
        )
        self.output_layer = nn.Linear(40 + n_features, 1)

    def forward(self, X):
        deep_output = self.deep_stack(X)
        wide_and_deep = torch.concat([X, deep_output], dim=1)
        return self.output_layer(wide_and_deep)
```

Notice that we can use any kind of module inside our custom module: in this example, we use a `nn.Sequential` module to build the "deep" part of our model (it's actually not that deep; this is just a toy example). It's the same MLP as earlier, except we separated the output layer because we need to feed it the concatenation of the model's inputs and the deep part's outputs. For this same reason, the output layer now has 40 + `n_features` inputs instead of just 40.

In the `forward()` method, we just feed the input `X` to the deep stack, concatenate the input and the deep stack's output, and feed the result to the output layer.

NOTE

Modules have a `children()` method that returns an iterator over the module's submodules (nonrecursively). There's also a `named_children()` method. If your model has a variable number of submodules, you should store them in a `nn.ModuleList` or a `nn.ModuleDict`, which are returned by the `children()` and `named_children()` methods (as opposed to regular Python lists and dicts). Similarly, if your model has a variable number of parameters, you should store them in a `nn.ParameterList` or a `nn.ParameterDict` to ensure they are returned by the `parameters()` and `named_parameters()` methods.

Now we can create an instance of our custom module, move it to the GPU, train it, evaluate it, and use it exactly like our previous models:

```
torch.manual_seed(42)
model = WideAndDeep(n_features).to(device)
learning_rate = 0.002  # the model changed, so did the optimal learning rate
[...]  # train, evaluate, and use the model, exactly like earlier
```

But what if you want to send a subset of the features through the wide path and a different subset (possibly overlapping) through the deep path, as illustrated in Figure 10-2? In this case, one approach is to split the inputs inside the `forward()` method, for example:

```
class WideAndDeepV2(nn.Module):
    [...]  # same constructor as earlier, except with adjusted input sizes

    def forward(self, X):
        X_wide = X[:, :5]
        X_deep = X[:, 2:]
        deep_output = self.deep_stack(X_deep)
        wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
        return self.output_layer(wide_and_deep)
```

This works fine; however, in many cases it's preferable to just let the model take two separate tensors as input. Let's see why and how.

Building Models with Multiple Inputs

Some models require multiple inputs that cannot easily be combined into a single tensor. For example, the inputs may have a different number of dimensions (e.g., when you want to feed both images and text to the neural network). To make our Wide & Deep model take two separate inputs, as shown in Figure 10-2, we must start by changing the model's `forward()` method:

```
class WideAndDeepV3(nn.Module):
    [...]  # same as WideAndDeepV2

    def forward(self, X_wide, X_deep):
```

```
        deep_output = self.deep_stack(X_deep)
        wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
        return self.output_layer(wide_and_deep)
```

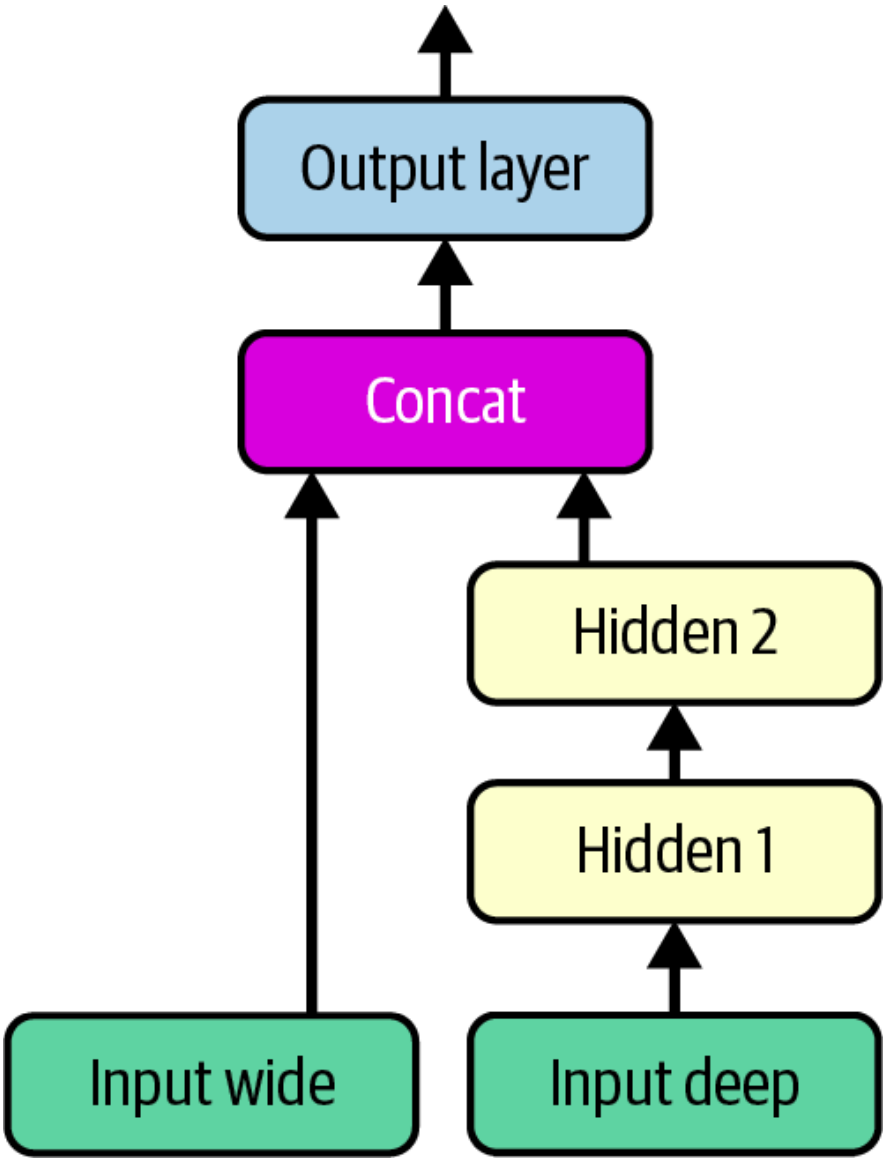


Figure 10-2. Handling multiple inputs

Next, we need to create datasets that return the wide and deep inputs separately:

```
train_data_wd = TensorDataset(X_train[:, :5], X_train[:, 2:], y_train)
train_loader_wd = DataLoader(train_data_wd, batch_size=32, shuffle=True)
[...]  # same for the validation set and test set
```

Since the data loaders now return three tensors instead of two at each iteration, we need to update the main loop in the evaluation and training functions:

```
for X_batch_wide, X_batch_deep, y_batch in data_loader:
    X_batch_wide = X_batch_wide.to(device)
    X_batch_deep = X_batch_deep.to(device)
    y_batch = y_batch.to(device)
    y_pred = model(X_batch_wide, X_batch_deep)
    [...]  # the rest of the function is unchanged
```

Alternatively, since the order of the inputs matches the order of the `forward()` method's arguments, we can use Python's `*` operator to unpack all the inputs returned by the `data_loader` and pass them to the model. The advantage of this implementation is that it will work with models that take any number of inputs, not just two, as long as the order is correct:

```
for *X_batch_inputs, y_batch in data_loader:
    X_batch_inputs = [X.to(device) for X in X_batch_inputs]
    y_batch = y_batch.to(device)
    y_pred = model(*X_batch_inputs)
    [...]
```

When your model has many inputs, it's easy to make a mistake and mix up the order of the inputs, which can lead to hard-to-debug issues. To avoid this, it can be a good idea to name each input. For this, you can define a custom dataset that returns a dictionary from input names to input values, like this:

```
class WideAndDeepDataset(torch.utils.data.Dataset):
    def __init__(self, X_wide, X_deep, y):
        self.X_wide = X_wide
        self.X_deep = X_deep
        self.y = y

    def __len__(self):
        return len(self.y)

    def __getitem__(self, idx):
        input_dict = {"X_wide": self.X_wide[idx], "X_deep": self.X_deep[idx]}
        return input_dict, self.y[idx]
```

Then create the datasets and data loaders:

```
train_data_named = WideAndDeepDataset(
    X_wide=X_train[:, :5], X_deep=X_train[:, 2:], y=y_train)
train_loader_named = DataLoader(train_data_named, batch_size=32, shuffle=True)
[...]  # same for the validation set and test set
```

Once again, we also need to update the main loop in the evaluation and training functions:

```
for inputs, y_batch in data_loader:
    inputs = {name: X.to(device) for name, X in inputs.items()}
    y_batch = y_batch.to(device)
    y_pred = model(X_wide=inputs["X_wide"], X_deep=inputs["X_deep"])
    [...]  # the rest of the function is unchanged
```

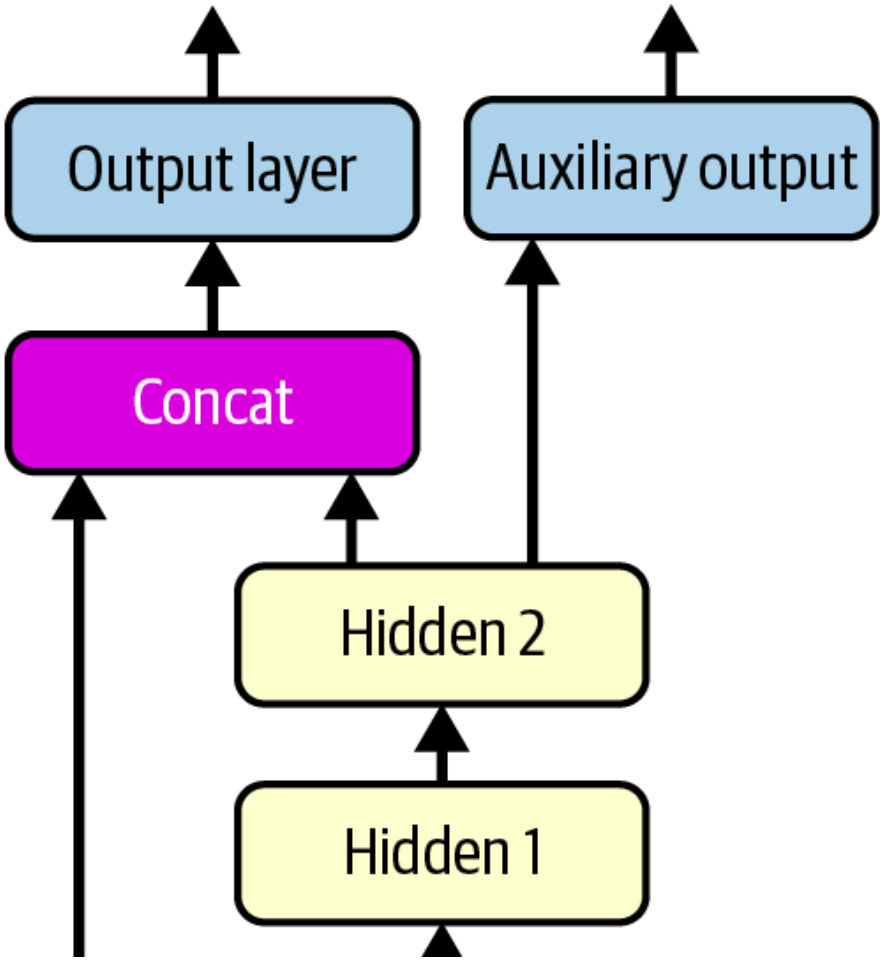
Alternatively, since all the input names match the `forward()` method's argument names, we can use Python's `**` operator to unpack all the tensors in the `inputs` dictionary and pass them as named arguments to the model: `y_pred = model(**inputs)`.

Now that you know how to build sequential and nonsequential models with one or more inputs, let's look at models with multiple outputs.

Building Models with Multiple Outputs

There are many use cases where you may need a neural net with multiple outputs:

- Firstly, the task may demand it. For instance, you may want to locate and classify the main object in a picture. This is both a regression task and a classification task.
- Similarly, you may have multiple independent tasks based on the same data. Sure, you could train one neural network per task, but in many cases you will get better results on all tasks by training a single neural network with one output per task. This is because the neural network can learn features in the data that are useful across tasks. For example, you could perform *multitask classification* on pictures of faces, using one output to classify the person's facial expression (smiling, surprised, etc.) and another output to identify whether they are wearing glasses or not.
- Another use case is regularization (i.e., a training constraint whose objective is to reduce overfitting and thus improve the model's ability to generalize). For example, you may want to add an auxiliary output in a neural network architecture (see Figure 10-3) to ensure that the underlying part of the network learns something useful on its own, without relying on the rest of the network.



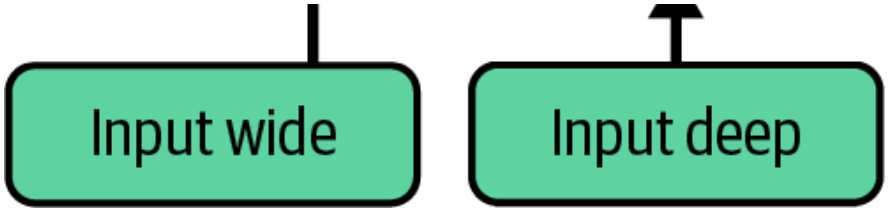


Figure 10-3. Handling multiple outputs, in this example to add an auxiliary output for regularization

Let's add an auxiliary output to our Wide & Deep model to ensure the deep part can make good predictions on its own. Since the deep stack's output dimension is 40, and the targets have a single dimension, we must add a `nn.Linear` layer for the auxiliary output to go from 40 dimensions down to 1. We also need to make the `forward()` method compute the auxiliary output, and return both the main output and the auxiliary output:

```
class WideAndDeepV4(nn.Module):
    def __init__(self, n_features):
        [...]  # same as earlier
        self.aux_output_layer = nn.Linear(40, 1)

    def forward(self, X_wide, X_deep):
        deep_output = self.deep_stack(X_deep)
        wide_and_deep = torch.concat([X_wide, deep_output], dim=1)
        main_output = self.output_layer(wide_and_deep)
        aux_output = self.aux_output_layer(deep_output)
        return main_output, aux_output
```

Next, we need to update the main loop in the training function:

```
for inputs, y_batch in train_loader:
    y_pred, y_pred_aux = model(**inputs)
    main_loss = criterion(y_pred, y_batch)
    aux_loss = criterion(y_pred_aux, y_batch)
    loss = 0.8 * main_loss + 0.2 * aux_loss
    [...]  # the rest is unchanged
```

Notice that the model now returns both the main predictions `y_pred` and the auxiliary predictions `y_pred_aux`. In this example, we can use the same targets and the same loss function to compute the main output's loss and the auxiliary output's loss. In other cases, you may have different targets and loss functions for each output, in which case you would need to create a custom dataset to return all the necessary targets. Once we have a loss for each output, we must combine them into a single loss that will be minimized by gradient descent. In general, this final loss is just a weighted sum of all the output losses. In this example, we use a higher weight for the main loss (0.8), because that's what we care about the most, and a lower weight for the auxiliary loss (0.2). This ratio is a regularization hyperparameter that you can tune.

We also need to update the main loop in the evaluation function. However, in this case we can just ignore the auxiliary output, since we only really care about the main output—the auxiliary output is just there for regularization during training:

```
for inputs, y_batch in data_loader:
    y_pred, _ = model(**inputs)
    metric.update(y_pred, y_batch)
    [...]  # the rest is unchanged
```

Voilà! You can now build and train all sorts of neural net architectures, combining predefined modules and custom modules in any way you please, and with any number of inputs and outputs. The flexibility of neural networks is one of their main qualities. But so far we have only tackled a regression task, so let's now turn to classification.

Building an Image Classifier with PyTorch

As in Chapter 9, we will tackle the Fashion MNIST dataset, so the first thing we need to do is to download the dataset. We could use the `fetch_openml()` function like we did in Chapter 9, but we will show another method instead, using the TorchVision library.

Using TorchVision to Load the Dataset

The TorchVision library is an important part of the PyTorch ecosystem: it provides many tools for computer vision, including utility functions to download common datasets, such as MNIST or Fashion MNIST, as well as pretrained models for various computer vision tasks (see Chapter 12), functions to transform images (e.g., crop, rotate, resize, etc.), and more. It is preinstalled on Colab, so let's go ahead and use it to load Fashion MNIST. It is already split into a training set (60,000 images) and a test set (10,000 images), but we'll hold out the last 5,000 images from the training set for validation, using PyTorch's `random_split()` function:

```
import torchvision
import torchvision.transforms.v2 as T

toTensor = T.Compose([T.ToImage(), T.ToDtype(torch.float32, scale=True)])

train_and_valid_data = torchvision.datasets.FashionMNIST(
    root="datasets", train=True, download=True, transform=toTensor)
test_data = torchvision.datasets.FashionMNIST(
    root="datasets", train=False, download=True, transform=toTensor)

torch.manual_seed(42)
train_data, valid_data = torch.utils.data.random_split(
    train_and_valid_data, [55_000, 5_000])
```

After the imports and before loading the datasets, we create a `toTensor` object. What's that about? Well, by default, the `FashionMNIST` class loads images as PIL (Python Image Library) images, with integer pixel values ranging from 0 to 255. But we need PyTorch float tensors instead, with scaled pixel values. Luckily, TorchVision datasets accept a `transform` argument

which lets you pass a preprocessing function that will get executed on the fly whenever the data is accessed (there's also a `target_transform` argument if you need to preprocess the targets). TorchVision provides many transform objects that you can use for this (most of these transforms are PyTorch modules).

In this code, we create a `Compose` transform to chain two transforms: a `ToImage` transform followed by a `ToDtype` transform. `ToImage` converts various formats—including PIL images, NumPy arrays, and tensors—to TorchVision's `Image` class, which is a subclass of `Tensor`. The `ToDtype` transform converts the data type, in this case to 32-bit floats. We also set its `scale` argument to `True` to ensure the values get scaled between 0.0 and 1.0.[12]

NOTE

Version 1 of TorchVision's transforms API is still available for backward compatibility and can be imported using `import torchvision.transforms`, but you should use version 2 (`torchvision.transformers.v2`) instead, since it's faster and has more features.

Next, we load the dataset: first the training and validation data, then the test data. The `root` argument is the path to the directory where TorchVision will create a subdirectory for the Fashion MNIST dataset. The `train` argument indicates whether you want to load the training set (`True` by default) or the test set. The `download` argument indicates whether to download the dataset if it cannot be found locally (`False` by default). And we also set `transform=toTensor` to use our custom preprocessing pipeline.

As usual, we must create data loaders:

```
train_loader = DataLoader(train_data, batch_size=32, shuffle=True)
valid_loader = DataLoader(valid_data, batch_size=32)
test_loader = DataLoader(test_data, batch_size=32)
```

Now let's look at the first image in the training set:

```
>>> X_sample, y_sample = train_data[0]
>>> X_sample.shape
torch.Size([1, 28, 28])
>>> X_sample.dtype
torch.float32
```

In Chapter 9, each image was represented by a 1D array containing 784 pixel intensities, but now each image tensor has 3 dimensions, and its shape is: `[1, 28, 28]`. The first dimension is the *channel* dimension. For grayscale images, there is a single channel (color images usually have three channels, as we will see in Chapter 12). The other two dimensions are the height and width dimensions. For example, `X_sample[0, 2, 4]` represents the pixel located in channel 0, row 2, column 4. In Fashion MNIST, a larger value means a darker pixel.

WARNING

PyTorch expects the channel dimension to be first, while many other libraries, such as Matplotlib, PIL, TensorFlow, OpenCV, or Scikit-Image, expect it to be last. Always make sure to move the channels dimension to the right place, depending on the library you are using. `ToImage` already took care of moving the channel dimension to the first position, otherwise we could have used the `torch.permute()` function.

As for the targets, they are integers from 0 to 9, and we can interpret them using the same `class_names` array as in Chapter 9. In fact, many datasets—including `FashionMNIST`—have a `classes` attribute containing the list of class names. For example, here's how we can tell that the sample image represents an ankle boot:

```
>>> train_and_valid_data.classes[y_sample]
'Ankle boot'
```

Building the Classifier

Let's build a custom module for a classification MLP with two hidden layers:

```
class ImageClassifier(nn.Module):
    def __init__(self, n_inputs, n_hidden1, n_hidden2, n_classes):
        super().__init__()
        self.mlp = nn.Sequential(
            nn.Flatten(),
            nn.Linear(n_inputs, n_hidden1),
            nn.ReLU(),
            nn.Linear(n_hidden1, n_hidden2),
            nn.ReLU(),
            nn.Linear(n_hidden2, n_classes)
        )

    def forward(self, X):
        return self.mlp(X)

torch.manual_seed(42)
model = ImageClassifier(n_inputs=28 * 28, n_hidden1=300, n_hidden2=100,
                        n_classes=10)
xentropy = nn.CrossEntropyLoss()
```

There are a few things to note in this code:

- First, the model is composed of a single sequence of layers, which is why we used the `nn.Sequential` module. We did not have to create a custom module; we could have written `model = nn.Sequential(...)` instead, but it's generally preferable to wrap your models in custom modules, as it makes your code easier to deploy and reuse, and it's also

easier to tune the hyperparameters.

- The model starts with a `nn.Flatten` layer: this layer does not have any parameters, it just reshapes each input sample to a single dimension, which is needed for the `nn.Linear` layers. For example, a batch of 32 Fashion MNIST images has a shape of `[32, 1, 28, 28]`, but after going through the `nn.Flatten` layer, it ends up with a shape of `[32, 784]` (since 28 × 28 = 784).
- The first hidden layer must have the correct number of inputs (28 × 28 = 784), and the output layer must have the correct number of outputs (10, one per class).
- We use a ReLU activation function after each hidden layer, and no activation function at all after the output layer.
- Since this is a multiclass classification task, we use `nn.CrossEntropyLoss`. It accepts either class indices as targets (as in this example), or class probabilities (such as one-hot vectors).

TIP

Shape errors are quite common, especially when getting started, so you should familiarize yourself with the error messages: try removing the `nn.Flatten` module, or try messing with the shape of the inputs and/or labels, and see the errors you get.

But wait! Didn't we say in Chapter 9 that we should use the softmax activation function on the output layer for multiclass classification tasks? Well it turns out that PyTorch's `nn.CrossEntropyLoss` computes the cross-entropy loss directly from the logits (i.e., the class scores, introduced in Chapter 4), rather than from the class probabilities. This bypasses some costly computations during training (e.g., logarithms and exponentials that cancel out), saving both compute and RAM. It's also more numerically stable. However, the downside is that the model must output logits, which means that we will have to call the softmax function manually on the logits whenever we want class probabilities, as we will see shortly.

OTHER CLASSIFICATION LOSSES

For multiclass classification, another option is to add the `nn.LogSoftmax` activation function to the output layer, and then use the `nn.NLLLoss` (negative log-likelihood loss). The model then outputs log probabilities (rather than logits), and the loss computes the cross-entropy based on these log probabilities. Whenever you need actual estimated probabilities, just pass the log probabilities through the exponential function. This approach is a bit slower than using `nn.CrossEntropyLoss`, so it's not used as often, but it can sometimes be useful if you want your model to output log probabilities, or when you wish to tweak the probability distribution before computing the loss.

For binary classification tasks, you must use a single output neuron in the output layer, and use the `nn.BCEWithLogitsLoss` (BCE stands for binary cross-entropy). The model outputs logits, so you must apply the sigmoid function to get estimated probabilities (for the positive class). Alternatively, you can add the `nn.Sigmoid` activation function to the output layer, and use the `nn.BCELoss`: the model will then output estimated probabilities directly (but it's a bit

slower and less numerically stable).

For multilabel binary classification, the only difference is that you must have one neuron per label in the output layer.

Now we can train the model as usual (e.g., using the `train()` function with an `SGD` optimizer). To evaluate the model, we can use the `Accuracy` streaming metric from the `torchmetrics` library, and move it to the GPU:

```
accuracy = torchmetrics.Accuracy(task="multiclass", num_classes=10).to(device)
```

WARNING

Training the model will take a few minutes with a GPU (or much longer without one). Handling images requires significantly more compute and memory than handling low-dimensional data.

The model reaches around 92.8% accuracy on the training set, and 87.6% accuracy on the validation set (the results might differ a bit depending on the hardware accelerator you use). This means there's a little bit of overfitting going on, so you may want to reduce the number of neurons or add some regularization (see Chapter 11).

Now that the model is trained, we can use it to make predictions on new images. As an example, let's make predictions for the first batch in the validation set, and look at the results for the first three images:

```
>>> model.eval()
>>> X_new, y_new = next(iter(valid_loader))
>>> X_new = X_new[:3].to(device)
>>> with torch.no_grad():
...     y_pred_logits = model(X_new)
...
>>> y_pred = y_pred_logits.argmax(dim=1)  # index of the largest logit
>>> y_pred
tensor([7, 4, 2], device='cuda:0')
>>> [train_and_valid_data.classes[index] for index in y_pred]
['Sneaker', 'Coat', 'Pullover']
```

For each image, the predicted class is the one with the highest logit. In this example, all three predictions are correct!

But what if we want the model's estimated probabilities? For this, we need to compute the softmax of the logits manually, since the model does not include the softmax activation function on the output layer, as we discussed earlier. We could create a `nn.Softmax` module and pass it the logits, but we can also just call the `softmax()` function, which is just one of many func-

tions you will find in the `torch.nn.functional` module (by convention, this module is usually imported as `F`). It doesn't make much difference, it just avoids creating a module instance that we don't need:

```
>>> import torch.nn.functional as F
>>> y_proba = F.softmax(y_pred_logits, dim=1)
>>> y_proba.round(decimals=3)
tensor([[0.000, 0.000, 0.000, 0.000, 0.000, 0.002, 0.000, 0.957, 0.000, 0.040],
        [0.000, 0.000, 0.003, 0.000, 0.997, 0.000, 0.000, 0.000, 0.000, 0.000],
        [0.001, 0.000, 0.670, 0.001, 0.144, 0.000, 0.182, 0.000, 0.001, 0.000]],
       device='cuda:0')
```

Just like in Chapter 9, the model is very confident about the first two predictions: 95.7% and 99.7% respectively.[13]

TIP

If you wish to apply label smoothing during training, just set the `label_smoothing` hyperparameter of the `nn.CrossEntropyLoss` to the amount of smoothing you wish, between 0 and 1 (e.g., 0.05).

It can often be useful to get the model's top *k* predictions. For this, we can use the `torch.topk()` function, which returns a tuple containing both the top *k* values and their indices:

```
>>> y_top4_logits, y_top4_indices = torch.topk(y_pred_logits, k=4, dim=1)
>>> y_top4_probas = F.softmax(y_top4_logits, dim=1)
>>> y_top4_probas.round(decimals=3)
tensor([[0.9570, 0.0400, 0.0020, 0.0000],
        [0.9970, 0.0030, 0.0000, 0.0000],
        [0.6720, 0.1830, 0.1440, 0.0010]], device='cuda:0')
>>> y_top4_indices
tensor([[7, 9, 5, 8],
        [4, 2, 6, 3],
        [2, 6, 4, 0]], device='cuda:0')
```

For the first image, the model's best guess is class 7 (Sneaker) with 95.7% confidence, its second best guess is class 9 (Ankle boot) with 4% confidence, and so on.

TIP

The Fashion MNIST dataset is balanced, meaning it has the same number of instances of each class. When dealing with an unbalanced dataset, you should generally give more weight to the rare classes and less weight to the frequent ones, or else your model will be biased toward the more frequent classes. You can do this by setting the `weight` argument of the `nn.CrossEntropyLoss`. For example, if there are three classes with 900, 700, and 400 instances, respectively (i.e., 2000 instances in total), then the respective weights should be 2000/900, 2000/700, and 2000/400. It's preferable to normalize these weights to ensure

they add up to 1, so in this example you would set `weight=torch.tensor([0.2205, 0.2835, 0.4961])`.

Your PyTorch superpowers are growing: you can now build, train, and evaluate both regression and classification neural nets. The next step is to learn how to fine-tune the model hyperparameters.

Fine-Tuning Neural Network Hyperparameters with Optuna

We discussed how to manually pick reasonable values for your model's hyperparameters in Chapter 9, but what if you want to go further and automatically search for good hyperparameter values? One option is to convert your PyTorch model to a Scikit-Learn estimator, either by writing your own custom estimator class or by using a wrapper library such as Skorch (*https://skorch.readthedocs.io*), and then use `GridSearchCV` or `RandomizedSearchCV` to fine-tune the hyperparameters, as you did in Chapter 2. However, you will usually get better results by using a dedicated fine-tuning library such as Optuna (*https://optuna.org*), Ray Tune (*https://docs.ray.io*), or Hyperopt (*https://hyperopt.github.io/hyperopt*). These libraries offer several powerful tuning strategies, and they're highly customizable.

Let's look at an example using Optuna. It is not preinstalled on Colab, so we need to install it using `%pip install optuna` (if you prefer to run the code locally, please follow the installation instructions at *https://homl.info/install-p*). Let's tune the learning rate and the number of neurons in the hidden layers (for simplicity, we will use the same number of neurons in both hidden layers). First, we need to define a function that Optuna will call many times to perform hyperparameter tuning: this function must take a `Trial` object and use it to ask Optuna for hyperparameter values, and then use these hyperparameter values to build and train a model. Finally, the function must evaluate the model (typically on the validation set) and return the metric:

```
import optuna

def objective(trial):
    learning_rate = trial.suggest_float("learning_rate", 1e-5, 1e-1, log=True)
    n_hidden = trial.suggest_int("n_hidden", 20, 300)
    model = ImageClassifier(n_inputs=1 * 28 * 28, n_hidden1=n_hidden,
                            n_hidden2=n_hidden, n_classes=10).to(device)
    optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)
    [...]  # train the model, then evaluate it on the validation set
    return validation_accuracy
```

The `suggest_float()` and `suggest_int()` methods let us ask Optuna for a good hyperparameter value in a given range (Optuna also provides a `suggest_categorical()` method).

For the `learning_rate` hyperparameter, we ask for a value between 10^{-3} and 10^{-1}, and since we don't know what the optimal scale is, we add `log=True`: this will make Optuna sample values from a log distribution, which makes it explore all possible scales. If we used the default uniform distribution instead, Optuna would be very unlikely to explore tiny values.

To start hyperparameter tuning, we create a `Study` object and call its `optimize()` method, passing it the objective function we just defined, as well as the number of trials to run (i.e., the number of times Optuna should call the objective function). Since our objective function returns a score—higher is better—we set `direction="maximize"` when creating the study (by default, Optuna tries to *minimize* the objective). To ensure reproducibility, we also set PyTorch's random seed, as well as the random seed used by Optuna's sampler:

```
torch.manual_seed(42)
sampler = optuna.samplers.TPESampler(seed=42)
study = optuna.create_study(direction="maximize", sampler=sampler)
study.optimize(objective, n_trials=5)
```

By default, Optuna uses the *Tree-structured Parzen Estimator* (TPE) algorithm to optimize the hyperparameters: this is a sequential model-based optimization algorithm, meaning it learns from past results to better select promising hyperparameters. In other words, Optuna starts with random hyperparameter values, but it progressively focuses its search on the most promising regions of the hyperparameter space. This allows Optuna to find much better hyperparameters than random search in the same amount of time.

TIP

You can add more hyperparameters to the search space, such as the batch size, the type of optimizer, the number of hidden layers, or the type of activation function, but remember that the search space will grow exponentially as you add more hyperparameters, so make sure it's worth the extra search time and compute.

Once Optuna is done, you can look at the best hyperparameters it found, as well as the corresponding validation accuracy:

```
>>> study.best_params
{'learning_rate': 0.008471801418819975, 'n_hidden': 188}
>>> study.best_value
0.8547999858856201
```

This is worse than the performance we got earlier, but that's because we set `n_trials=5`, which is way too small. If you bump this value up to 50 or more, you will get much better results, but of course it will take hours to run. You can also just run `optimize()` repeatedly and stop once you are happy with the performance.

Optuna can also run trials in parallel across multiple machines, which can offer a near linear

Optuna can also run trials in parallel across multiple machines, which can offer a near-linear speed boost. For this, you will need to set up a SQL database (e.g., SQLite or PostgreSQL), and set the `storage` parameter of the `create_study()` function to point to that database. You also need to set the study's name via the `study_name` parameter, and set `load_if_exists=True`. After that, you can copy your hyperparameter tuning script to multiple machines, and run it on each one (if you are using random seeds, make sure they are different on each machine). The scripts will work in parallel, reading and writing the trial results to the database. This has the additional benefit of keeping a full log of all your experiment results.

You may have noticed that we assumed that the `objective()` function had direct access to the training set and validation, presumably via global variables. In general, it's much cleaner to pass them as extra arguments to the `objective()` function, for example, like this:

```
def objective(trial, train_loader, valid_loader):
    [...]  # the rest of the function remains the same as above

objective_with_data = lambda trial: objective(
    trial, train_loader=train_loader, valid_loader=valid_loader)
study.optimize(objective_with_data, n_trials=5)
```

To set the extra arguments (the dataset loaders in this case), we just create a lambda function when needed and pass it to the `optimize()` method. Alternatively, you can use the `functools.partial()` function which creates a thin wrapper function around the given callable to provide default values for any number of arguments:

```
from functools import partial

objective_with_data = partial(objective, train_loader=train_loader,
                              valid_loader=valid_loader)
```

It's often possible to quickly tell that a trial is absolutely terrible: for example, when the loss shoots up during the first epoch, or when the model barely improves during the first few epochs. In such a case, it's a good idea to interrupt training early to avoid wasting time and compute. You can simply return the model's current validation accuracy and hope that Optuna will learn to avoid this region of hyperparameter space. Alternatively, you can interrupt training by raising the `optuna.TrialPruned` exception: this tells Optuna to ignore this trial altogether. In many cases, this leads to a more efficient search because it avoids polluting Optuna's search algorithm with many noisy model evaluations.

Optuna comes with several `Pruner` classes that can detect and prune bad trials. For example, the `MedianPruner` will prune trials whose performance is below the median performance, at regular intervals during training. It starts pruning after a given number of trials have completed, controlled by `n_startup_trials` (5 by default). For each trial after that, it lets training start for a few epochs, controlled by `n_warmup_steps` (0 by default); then every few epochs (controlled by `interval_steps`), it ensures that the model's performance is better

than the median performance at the same epoch in past trials. To use this pruner, create an instance and pass it to the `create_study()` method:

```
pruner = optuna.pruners.MedianPruner(n_startup_trials=5, n_warmup_steps=0,
                                     interval_steps=1)
study = optuna.create_study(direction="maximize", sampler=sampler,
                            pruner=pruner)
```

Then in the `objective()` function, add the following code so it runs after each epoch:

```
for epoch in range(n_epochs):
    [...]  # train the model for one epoch
    validation_accuracy = [...]  # evaluate the model's validation accuracy
    trial.report(validation_accuracy, epoch)
    if trial.should_prune():
        raise optuna.TrialPruned()
```

The `report()` method informs Optuna of the current validation accuracy and epoch, so it can determine whether the trial should be pruned. If `trial.should_prune()` returns `True`, we raise a `TrialPruned` exception.

TIP

Optuna has many other features well worth exploring, such as visualization tools, persistence tools for trial results and other artifacts, a dashboard for human-in-the-loop optimization, and many other algorithms for hyperparameter search and trial pruning.

Once you are happy with the hyperparameters, you can train the model on the full training set (i.e., the training set plus the validation set), then evaluate it on the test set. Hopefully, it will perform great! If it does, you will want to save the model, then load it and use it in production: that's the final topic of this chapter.

Saving and Loading PyTorch Models

The simplest way to save a PyTorch model is to use the `torch.save()` method, passing it the model and the file path. The model object is serialized using Python's `pickle` module (which can convert objects into a sequence of bytes), then the result is compressed (zip) and saved to disk. The convention is to use the `.pt` or `.pth` extension for PyTorch files:

```
torch.save(model, "my_fashion_mnist.pt")
```

Simple! Now you can load the model (e.g., in your production code) just as easily:

```
loaded_model = torch.load("my_fashion_mnist.pt", weights_only=False)
```

WARNING

If your model uses any custom functions or classes (e.g., `ImageClassifier`), then `torch.save()` only saves references to them, not the code itself. Therefore you must ensure that any custom code is loaded in the Python environment before calling `torch.load()`. Also make sure to use the same version of the code to avoid any mismatch issues.

Setting `weights_only=False` ensures that the whole model object is loaded rather than just the model parameters. Then you can use the loaded model for inference. Don't forget to switch to evaluation mode first using the `eval()` method:

```
loaded_model.eval()
y_pred_logits = loaded_model(X_new)
```

This is nice and easy, but unfortunately this approach has some very serious drawbacks:

- Firstly, pickle's serialization format is notoriously insecure. While `torch.save()` doesn't save custom code, the pickle format supports it, so a hacker could inject malicious code in a saved PyTorch model: this code would be run automatically by the pickle module when the model is loaded. So always make sure you fully trust the model's source before you load it this way.
- Secondly, pickle is somewhat brittle. It can vary depending on the Python version (e.g., there were big changes between Python 3.7 and 3.8), and it saves specific file paths to locate code, which can break if the loading environment has a different folder structure.

To avoid these issues, it is recommended to save and load the model weights only, rather than the full model object:

```
torch.save(model.state_dict(), "my_fashion_mnist_weights.pt")
```

The state dictionary returned by the `state_dict()` method is just a Python `OrderedDict` containing an entry for each parameter returned by the `named_parameters()` method. It also contains buffers, if the model has any: a buffer is just a regular tensor that was registered with the model (or any of its submodules) using the `register_buffer()` method. Buffers hold extra data that needs to be stored along with the model, but that is not a model parameter. We will see an example in Chapter 11 with the batch-norm layer.

To load these weights, we must first create a model with the exact same structure, then load the weights using `torch.load()` with `weights_only=True`, and finally call the model's `load_state_dict()` method with the loaded weights:

```
new_model = ImageClassifier(n_inputs=1 * 28 * 28, n_hidden1=300, n_hidden2=100,
                            n_classes=10)
loaded_weights = torch.load("my_fashion_mnist_weights.pt", weights_only=True)
new_model.load_state_dict(loaded_weights)
new_model.eval()
```

The saved model contains only data, and the `load()` function makes sure of that, so this is safe, and also much less likely to break between Python versions or to cause any deployment issue. However, it only works if you are able to create the exact same model architecture before loading the state dictionary. For this, you need to know the number of layers, the number of neurons per layer, and so on. It's a good idea to save this information along with the state dictionary:

```
model_data = {
    "model_state_dict": model.state_dict(),
    "model_hyperparameters": {"n_inputs": 1 * 28 * 28, "n_hidden1": 300, [...]}
}
torch.save(model_data, "my_fashion_mnist_model.pt")
```

You can then load this dictionary, construct the model based on the saved hyperparameters, and load the state dictionary into this model:

```
loaded_data = torch.load("my_fashion_mnist_model.pt", weights_only=True)
new_model = ImageClassifier(**loaded_data["model_hyperparameters"])
new_model.load_state_dict(loaded_data["model_state_dict"])
new_model.eval()
```

If you want to be able to continue training where it left off, you will also need to save the optimizer's state dictionary, its hyperparameters, and any other training information you may need, such as the current epoch and the loss history.

TIP

The `safetensors` library by Hugging Face is another popular way to save model weights safely.

There is yet another way to save and load your model: by first converting it to TorchScript. This also makes it possible to speed up your model's inference.

Compiling and Optimizing a PyTorch Model

PyTorch comes with a very nice feature: it can automatically convert your model's code to *TorchScript*, which you can think of as a statically typed subset of Python. There are two main

benefits:

- First, TorchScript code can be compiled and optimized to produce significantly faster models. For example, multiple operations can often be fused into a single, more efficient operation. Operations on constants (e.g., 2 * 3) can be replaced with their result (e.g., 6); this is called *constant folding*. Unused code can be pruned, and so on.
- Secondly, TorchScript can be serialized, saved to disk, and then loaded and executed in Python or in a C++ environment using the LibTorch library. This makes it possible to run PyTorch models on a wide range of devices, including embedded devices.

There are two ways to convert a PyTorch model to TorchScript. The first way is called *tracing*. PyTorch just runs your model with some sample data, logs every operation that takes place, and then converts this log to TorchScript. This is done using the `torch.jit.trace()` function:

```
torchscript_model = torch.jit.trace(model, X_new)
```

This generally works well with static models whose `forward()` method doesn't use conditionals or loops. However, if you try to trace a model that includes an `if` or `match` statement, then only the branch that is actually executed will be captured by TorchScript, which is generally not what you want. Similarly, if you use tracing with a model that contains a loop, then the TorchScript code will contain one copy of the operations within that loop for each iteration that was actually executed. Again, not what you generally want.

For such dynamic models, you will probably want to try another approach named *scripting*. In this case, PyTorch actually parses your Python code directly and converts it to TorchScript. This method supports `if` statements and `while` loops properly, as long as the conditions are tensors. It also supports `for` loops when iterating over tensors. However, it only works on a subset of Python. For example, you cannot use global variables, Python generators (`yield`), complex list comprehensions, variable length function arguments (`*args` or `**kwargs`), or `match` statements. Moreover, types must be fixed (a function cannot return an integer in some cases and a float in others), and you can only call other functions if they also respect these rules, so no standard library, no third-party libraries, etc. (see the documentation for the full list of constraints). This sounds daunting, but for most real-world models, these rules are actually not too hard to respect, and you can save your model like this:

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

Regardless of whether you use tracing or scripting to produce your TorchScript model, you can then further optimize it:

```
optimized_model = torch.jit.optimize_for_inference(torchscript_model)
```

TorchScript models can only be used for inference, not for training, since the TorchScript envi-

ronment doesn't support gradient tracking or parameter updates.

Finally, you can save a TorchScript model using its `save()` method:

```
torchscript_model.save('my_fashion_mnist_torchscript.pt')
```

And then load it using the `torch.jit.load()` function:

```
loaded_torchscript_model = torch.jit.load("my_fashion_mnist_torchscript.pt")
```

One important caveat: TorchScript is no longer under active development—bugs are fixed but no new features are added. It still works fine and it remains one of the best ways to run your PyTorch models in a C++ environment,[14] but since the release of PyTorch 2.0 in March 2023, the PyTorch team has been focusing its efforts on a new set of compilation tools centered around the `torch.compile()` function, which you can use very easily:

```
compiled_model = torch.compile(model)
```

The resulting model can now be used normally, and it will automatically be compiled and optimized when you use it. This is called Just-In-Time (JIT) compilation, as opposed to Ahead-Of-Time (AOT) compilation. Under the hood, `torch.compile()` relies on *TorchDynamo* (or *Dynamo* for short) which hooks directly into Python bytecode to capture the model's computation graph at inference time. Having access to the bytecode allows Dynamo to efficiently and reliably capture the computation graph, properly handling conditionals and loops, while also benefiting from dynamic information that can be used to better optimize the model. The actual compilation and optimization is performed by default by a backend component named *TorchInductor*, which in turn relies on the Triton language to generate highly efficient GPU code (Nvidia only), or on the OpenMP API for CPU optimization. PyTorch 2.x offers a few other optimization backends, including the XLA backend for Google's TPU devices: just set `device="xla"` when calling `torch.compile()`.

With that, you now have all the tools you need to start building and training complex and efficient neural networks. I hope you enjoyed this introduction to PyTorch! We covered a lot, but the adventure is only beginning: in the next chapter we will discuss techniques to train very deep nets. After that, we will dive into other popular neural network architectures: convolutional neural networks for image processing, recurrent neural networks for sequential data, transformers for text (and much more), autoencoders for representation learning, and generative adversarial networks and diffusion models to generate data.[15] Then we will visit reinforcement learning to train autonomous agents, and finally, we will learn more about deploying and optimizing your PyTorch models. Let's go!

Exercises

1. PyTorch is similar to NumPy is many ways, but it offers some extra features. Can you name the most important ones?
2. What is the difference between `torch.exp()` and `torch.exp_()`, or between `torch.relu()` and `torch.relu_()`?
3. What are two ways to create a new tensor on the GPU?
4. What are three ways to perform tensor computations without using autograd?
5. Will the following code cause a `RuntimeError`? What if you replace the second line with `z = t.cos_().exp()`? And what if you replace it with `z = t.exp().cos_()`?

```
t = torch.tensor(2.0, requires_grad=True)
z = t.cos().exp_()
z.backward()
```

   How about the following code, will it cause an error? And what if you replace the third line with `w = v.cos_() * v.sin()`? Will `w` have the same value in both cases?

```
u = torch.tensor(2.0, requires_grad=True)
v = u + 1
w = v.cos() * v.sin_()
w.backward()
```

6. Suppose you create a `Linear(100, 200)` module. How many neurons does it have? What is the shape of is `weight` and `bias` parameters? What input shape does it expect? What output shape does it produce?
7. What are the main steps of a PyTorch training loop?
8. Why is it recommended to create the optimizer *after* the model is moved to the GPU?
9. What `DataLoader` options should you generally set to speed up training when using a GPU?
10. What are the main classification losses provided by PyTorch, and when should you use each of them?
11. Why is it important to call `model.train()` before training and `model.eval()` before evaluation?
12. What is the difference between `torch.jit.trace()` and `torch.jit.script()`?
13. Use autograd to find the gradient vector of $f(x, y) = \sin(x^2 y)$ at the point $(x, y) = (1.2, 3.4)$.
14. Create a custom `Dense` module that replicates the functionality of a `nn.Linear` module followed by a `nn.ReLU` module. Try implementing it first using the `nn.Linear` and `nn.ReLU` modules, and then reimplement it using `nn.Parameter` and the `relu()` function.
15. Build and train a classification MLP on the CoverType dataset:
    a. Load the dataset using `sklearn.datasets.fetch_covtype()` and create a custom PyTorch `Dataset` for this data.
    b. Create data loaders for training, validation, and testing.
    c. Build a custom MLP module to tackle this classification task. You can optionally use the custom `Dense` module from the previous exercise.

custom `Dense` module from the previous exercise.

d. Train this model on the GPU, and try to reach 93% accuracy on the test set. For this, you will likely have to perform hyperparameter search to find the right number of layers and neurons per layer, a good learning rate and batch size, and so on. You can optionally use Optuna for this.

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

[1] To be fair, most of TensorFlow's usability issues were fixed in version 2, and Google also launched JAX, which is well designed and extremely fast, so PyTorch still has some healthy competition. The good news is that the APIs of all these libraries have converged quite a bit, so switching from one to the other is much easier than it used to be.

[2] There are things called tensors in mathematics and physics, but ML tensors are simpler: they're really just multidimensional arrays for numerical computations, plus a few extra features.

[3] CUDA is Nvidia's proprietary platform to run code on their CUDA-compatible GPUs, and cuDNN is a library built on CUDA to accelerate various deep neural network architectures.

[4] The `%timeit` magic command only works in Jupyter notebooks and Colab, as well as in the iPython shell; in a regular Python shell or program, you can use the `timeit.timeit()` function instead.

[5] For example, since the derivative of exp(x) is equal to exp(x), it makes a lot of sense to store the output of this operation in the computation graph during the forward pass, then use this output during the backward pass to get the gradients: no need to store additional data, and no need to recompute exp(x).

[6] For example, the derivative of abs(x) is –1 when $x < 0$ and +1 when $x > 0$. If this operation saved its output in the computation graph, the backward pass would be unable to know whether x was positive or negative (since abs(x) is always positive), so it wouldn't be able to compute the gradients. This is why this operation must save its input instead.

[7] For example, the derivative of floor(x) is always zero (at least for noninteger inputs), so the `floor()` operation just saves the shape of the inputs during the forward pass, then during the backward pass it produces gradients of the same shape but full of zeros. For integer inputs, autograd also returns zeros instead of NaN.

[8] Column vectors (shape [m, 1]) and row vectors (shape [1, m]) are often preferred over 1D vectors (shape [m]) in machine learning, as they avoid ambiguity in some operations, such as matrix multiplication or broadcasting, and they make the code more consistent whether there's just one feature or more.

[9] Just like in NumPy, the `reshape()` method allows you to specify –1 for one of the dimensions. This dimension's size is automatically calculated to ensure the new tensor has the same number of cells as the original.

[10] The mean of the batch MSEs is equal to the overall MSE since all batches have the same size. Well, except the last batch, which is often smaller, but this makes very little difference (the correct RMSE is 0.6477 instead of 0.6476).

[11] Heng-Tze Cheng et al., "Wide & Deep Learning for Recommender Systems", *Proceedings of the First Workshop on Deep Learning for Recommender Systems* (2016): 7–10.

[12] TorchVision includes a `ToTensor` transform which does all this, but it's deprecated so it's recommended to use this pipeline instead.

[13] The `round()` method does not yet support the `decimals` argument on MPS devices, so if you run this code with MPS acceleration, you should move the predictions to the CPU before calling `round()`.

[14] Another popular option is exporting your PyTorch model to the open ONNX standard using `torch.onnx.export()`. The ONNX model can then be used for inference in a wide variety of environments.

[15] A few extra ANN architectures are presented in the online notebook at *https://homl.info/extra-anns*.