7 Telling birds from airplanes: Learning from images

This chapter covers

- Building a feed-forward neural network
- Loading data using `Dataset` s and `DataLoader` s
- Understanding classification loss

The last chapter gave us the opportunity to dive into the inner mechanics of learning through gradient descent, and the facilities that PyTorch offers to build models and optimize them. We did so using a simple regression model of one input and one output, which allowed us to have everything in plain sight but admittedly was only borderline exciting.

In this chapter, we'll keep moving ahead with building our neural network foundations. This time, we'll turn our attention to images. Image recognition is arguably the task that made the world realize the potential of deep learning.

We will approach a simple image recognition problem step by step, building from a simple neural network like the one we defined in the last chapter. This time, instead of a tiny dataset of numbers, we’ll use a more extensive dataset of tiny images. Let’s download the dataset first and get to work preparing it for use.

A dataset of tiny images

There is nothing like an intuitive understanding of a subject, and there is nothing to achieve that like working on simple data. One of the most basic datasets for image recognition is the handwritten digit-recognition dataset known as MNIST. Here we will use another dataset that is similarly simple and a bit more fun. It’s called CIFAR-10, and, like its sibling CIFAR-100, it has been a computer vision classic for a decade.

CIFAR-10 consists of 60,000 tiny 32 × 32 color (RGB) images, labeled with an integer corresponding to 1 of 10 classes: airplane (0), automobile (1), bird (2), cat (3), deer (4), dog (5), frog (6), horse (7), ship (8), and truck (9).[1] Nowadays, CIFAR-10 is considered too simple for developing or validating new research, but it serves our learning purposes just fine. We will use the `torchvision` module to automatically download the dataset and load it as a collection of PyTorch tensors. Figure 7.1 gives us a taste of CIFAR-10.

1.The images were collected and labeled by Krizhevsky, Nair, and Hinton of the Canadian Institute For Advanced

Research (CIFAR) and were drawn from a larger collection of unlabeled 32 × 32 color images: the “80 million tiny images dataset” from the Computer Science and Artificial Intelligence Laboratory (CSAIL) at the Massachusetts Institute of Technology.

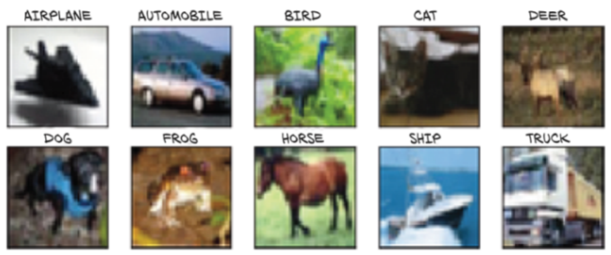


Figure 7.1 Image samples from all CIFAR-10 classes

.1 Downloading CIFAR-10

As we anticipated, let’s import `torchvision` and use the `datasets` module to download the CIFAR-10 data:

```
# In[2]:
from torchvision import datasets
data_path = '../data-unversioned/p1ch7/'
cifar10 = datasets.CIFAR10(data_path, train=True, download=True)         ❶
cifar10_val = datasets.CIFAR10(data_path, train=False, download=True)   ❷
```

❶ Instantiates a dataset for the training data; TorchVision downloads the data if it is not present

❷ With train=False, this gets us a dataset for the validation data, again downloading as necessary.

The first argument we provide to the `CIFAR10` function is the location from which the data will be downloaded; the second specifies whether we're interested in the training set or the validation set; and the third says whether we allow PyTorch to download the data if it is not found in the location specified in the first argument.

Just like `CIFAR10`, the `datasets` submodule gives us precanned access to the most popular computer vision datasets, such as MNIST, Fashion-MNIST, CIFAR-100, SVHN, Coco, and Omniglot. In each case, the dataset is returned as a subclass of `torch.utils.data.Dataset`. We can see that the method-resolution order of our `cifar10` instance includes it as a base class:

```
# In[4]:
type(cifar10).__mro__

# Out[4]:
(torchvision.datasets.cifar.CIFAR10,
 torchvision.datasets.vision.VisionDataset,
 torch.utils.data.dataset.Dataset,
 object)
```

.2 The Dataset class

It's a good time to discover what being a subclass of `torch.utils.data.Dataset` means in practice. Looking at figure 7.2, we see what PyTorch `Dataset` is all about. It is an object that is required to implement two methods:

`__len__` and `__getitem__`. The former should return the number of items in the dataset; the latter should return the item, consisting of a sample and its corresponding label (an integer index).[2]

2.For some advanced uses, PyTorch also provides `IterableDataset`. This can be used in cases like datasets in which random access to the data is prohibitively expensive or does not make sense: for example, because data is generated on the fly.

In practice, when a Python object is equipped with the `__len__` method, we can pass it as an argument to the `len` Python built-in function:

```
# In[5]:
len(cifar10)

# Out[5]:
50000
```

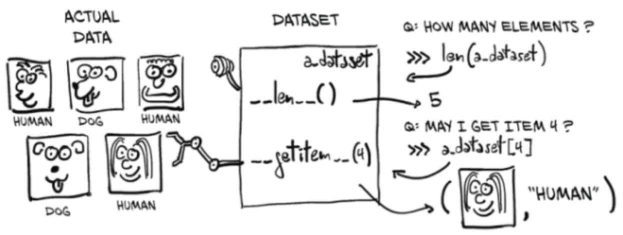


Figure 7.2 Concept of a PyTorch `Dataset` object: it doesn't necessarily hold the data, but it provides uniform access to it through `__len__` and `__getitem__`.

Similarly, since the dataset is equipped with the `__getitem__` method, we can use the standard subscript for indexing tuples and lists to access individual items. Here, we get a `PIL` (Python Imaging Library, the `PIL` package) image with our desired output--an integer with the value `1`, corresponding to “automobile”:

```
# In[6]:
img, label = cifar10[99]
img, label, class_names[label]

# Out[6]:
(<PIL.Image.Image image mode=RGB size=32x32 at 0x7FB383657390>,
 1,
 'automobile')
```

So, the sample in the `data.CIFAR10` dataset is an instance of an RGB PIL image. We can plot it right away:

```
# In[7]:
plt.imshow(img)
plt.show()
```

This produces the output shown in figure 7.3. It’s a red car![3]

---

[3]It doesn’t translate well to print; you’ll have to take our word for it, or check it out in the eBook or the Jupyter Notebook.

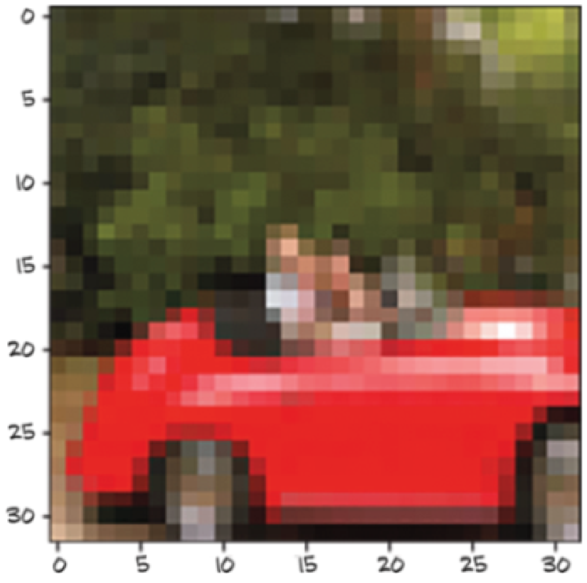


Figure 7.3 The 99th image from the CIFAR-10 dataset: an automobile

.3 Dataset transforms

That's all very nice, but we'll likely need a way to convert the PIL image to a PyTorch tensor before we can do anything with it. That's where `torchvision.transforms` comes in. This module defines a set of composable, function-like objects that can be passed as an argument to a `torchvision` dataset such as `datasets.CIFAR10(...)`, and that perform transformations on the data after it is loaded but before it is returned by `__getitem__`. We can see the list of available objects as follows:

```
# In[8]:
from torchvision import transforms
dir(transforms)

# Out[8]:
['CenterCrop',
 'ColorJitter',
   ...
   'Normalize',
   'Pad',
   'RandomAffine',
   ...
   'RandomResizedCrop',
   'RandomRotation',
   'RandomSizedCrop',
   ...
   'TenCrop',
   'ToPILImage',
   'ToTensor',
   ...
  ]
```

Among those transforms, we can spot `ToTensor`, which turns NumPy arrays and PIL images to tensors. It also takes care to lay out the dimensions of the output tensor as $C \times H \times W$ (channel, height, width; just as we covered in chapter 4).

Let's try out the `ToTensor` transform. Once instantiated, it can be called like a function with the PIL image as the argument, returning a tensor as output:

```
# In[9]:


to_tensor = transforms.ToTensor()
img_t = to_tensor(img)
img_t.shape

# Out[9]:
torch.Size([3, 32, 32])
```

The image has been turned into a 3 × 32 × 32 tensor and therefore a 3-channel

(RGB) 32 × 32 image. Note that nothing has happened to `label`; it is still an integer.

As we anticipated, we can pass the transform directly as an argument to `dataset .CIFAR10`:

```
# In[10]:
tensor_cifar10 = datasets.CIFAR10(data_path, train=True, download=False,
                                  transform=transforms.ToTensor())
```

At this point, accessing an element of the dataset will return a tensor, rather than a PIL image:

```
# In[11]:
img_t, _ = tensor_cifar10[99]
type(img_t)

# Out[11]:
torch.Tensor
```

As expected, the shape has the channel as the first dimension, while the scalar type is `float32`:

```
# In[12]:
img_t.shape, img_t.dtype

# Out[12]:
(torch.Size([3, 32, 32]), torch.float32)
```

Whereas the values in the original PIL image ranged from 0 to 255 (8 bits per channel), the `ToTensor` transform turns the data into a 32-bit floating-point per channel, scaling the values down from 0.0 to 1.0. Let’s verify that:

```
# In[13]:
img_t.min(), img_t.max()

# Out[13]:
(tensor(0.), tensor(1.))
```

And let’s verify that we’re getting the same image out:

```
# In[14]:
plt.imshow(img_t.permute(1, 2, 0))    ❶
plt.show()

# Out[14]:
<Figure size 432x288 with 1 Axes>
```

❶ Changes the order of the axes from C × H × W to H × W × C

As we can see in figure 7.4, we get the same output as before.

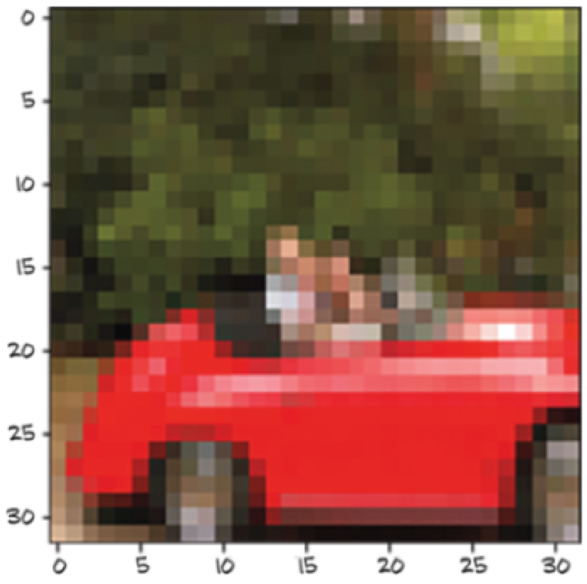


Figure 7.4 We've seen this one already.

It checks. Note how we have to use `per-mute` to change the order of the axes from C × H × W to H × W × C to match what Matplotlib expects.

.4 Normalizing data

Transforms are really handy because we can chain them using `transforms.Com-pose`, and they can handle normalization and data augmentation transparently, directly in the data loader. For instance, it's good practice to normalize the dataset so that each channel has zero mean and unitary standard deviation. We mentioned this in chapter 4, but now, after going through chapter 5, we also have an intuition for why: by choosing activation functions that are linear around 0 plus or minus 1 (or 2), keeping the data in the same range means it's more likely that neurons have nonzero gradients and, hence, will learn sooner. Also, normalizing each channel so that it has the same distribution will ensure that channel information can be mixed and updated through gradient descent using the same learning rate. This is just like the situation in section 5.4.4 when we rescaled the weight to be of the same magnitude as the bias in our temperature-conversion model.

In order to make it so that each channel has zero mean and unitary standard deviation, we can compute the mean value and the standard deviation of each channel across the dataset and apply the following transform: `v_n[c] = (v[c] - mean[c]) / stdev[c]`. This is what `transforms.Normalize` does. The values of `mean` and `stdev` must be computed offline (they are not computed by the transform). Let's compute them for the CIFAR-10 training set.

Since the CIFAR-10 dataset is small, we'll be able to manipulate it entirely in memory. Let's stack all the tensors returned by the dataset along an extra dimension:

```
# In[15]:
imgs = torch.stack([img_t for img_t, _ in tensor_cifar10], dim=3)
imgs.shape

# Out[15]:
torch.Size([3, 32, 32, 50000])
```

Now we can easily compute the mean per channel:

```
# In[16]:
imgs.view(3, -1).mean(dim=1)      ❶

# Out[16]:
tensor([0.4915, 0.4823, 0.4468])
```

❶ Recall that view(3, -1) keeps the three channels and merges all the remaining dimensions into one, figuring out the appropriate size. Here our 3 × 32 × 32 image is transformed into a 3 × 1,024 vector, and then the mean is taken over the 1,024 elements of each channel.

Computing the standard deviation is similar:

```
# In[17]:
imgs.view(3, -1).std(dim=1)

# Out[17]:
tensor([0.2470, 0.2435, 0.2616])
```

With these numbers in our hands, we can initialize the `Normalize` transform

```
# In[18]:
transforms.Normalize((0.4915, 0.4823, 0.4468), (0.2470, 0.2435, 0.2616))

# Out[18]:
Normalize(mean=(0.4915, 0.4823, 0.4468), std=(0.247, 0.2435, 0.2616))
```

and concatenate it after the `ToTensor` transform:

```
# In[19]:
transformed_cifar10 = datasets.CIFAR10(
    data_path, train=True, download=False,
    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.4915, 0.4823, 0.4468),
                             (0.2470, 0.2435, 0.2616))
    ]))
```

Note that, at this point, plotting an image drawn from the dataset won't provide us with a faithful representation of the actual image:

```
# In[21]:
img_t, _ = transformed_cifar10[99]

plt.imshow(img_t.permute(1, 2, 0))
plt.show()
```

The renormalized red car we get is shown in figure 7.5. This is because normalization has shifted the RGB levels outside the 0.0 to 1.0 range and changed the overall magnitudes of the channels. All of the data is still there; it's just that Matplotlib renders it as black. We'll keep this in mind for the future.

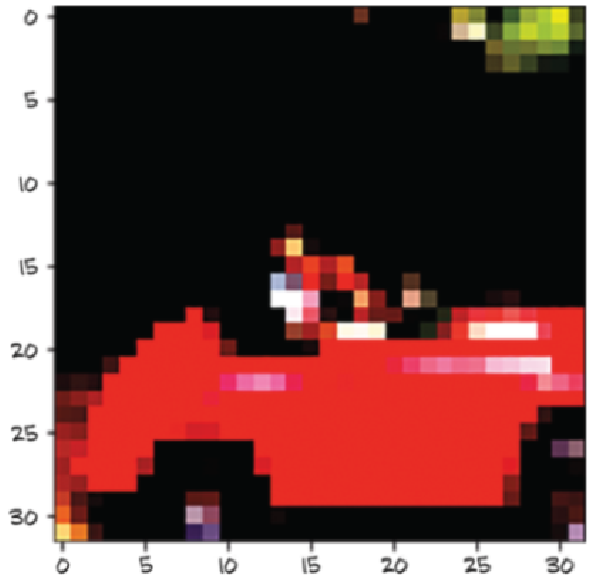


Figure 7.5 Our random CIFAR-10 image after normalization

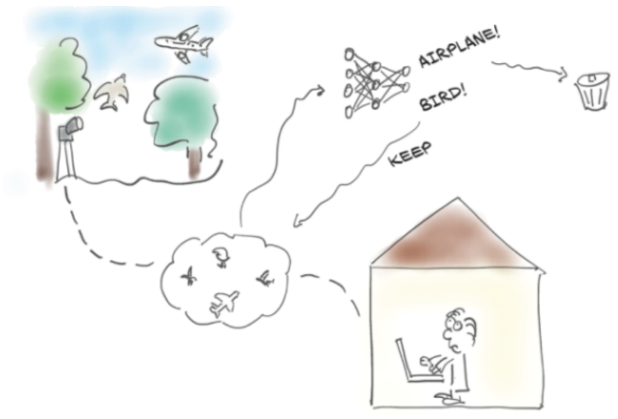


Figure 7.6 The problem at hand: we're going to help our friend tell birds from airplanes for her blog, by training a neural network to do the job.

Still, we have a fancy dataset loaded that contains tens of thousands of images! That's quite convenient, because we were going to need something exactly like it.

Distinguishing birds from airplanes

Jane, our friend at the bird-watching club, has set up a fleet of cameras in the woods south of the airport. The cameras are supposed to save a shot when something enters the frame and upload it to the club's real-time bird-watching blog. The problem is that a lot of planes coming and going from the airport end up triggering the camera, so Jane spends a lot of time deleting pictures of airplanes from the blog. What she needs is an automated system like that shown in figure 7.6. Instead of manually deleting, she needs a neural network--an AI if we're into fancy marketing speak--to throw away the airplanes right away.

No worries! We'll take care of that, no problem--we just got the perfect dataset for it (what a coincidence, right?). We'll pick out all the birds and airplanes from our CIFAR-10 dataset and build a neural network that can tell birds and airplanes apart.

.1 Building the dataset

The first step is to get the data in the right shape. We could create a `Dataset` subclass that only includes birds and airplanes. However, the dataset is small, and we only need indexing and `len` to work on our dataset. It doesn't actually have to be a subclass of `torch.utils.-data.dataset.Dataset` ! Well, why not take a shortcut and just filter the data in `cifar10` and remap the labels so they are contiguous? Here's how:

```
# In[5]:
label_map = {0: 0, 2: 1}
class_names = ['airplane', 'bird']
cifar2 = [(img, label_map[label])
          for img, label in cifar10
          if label in [0, 2]]
cifar2_val = [(img, label_map[label])
              for img, label in cifar10_val
              if label in [0, 2]]
```

The `cifar2` object satisfies the basic requirements for a `Dataset`--that is, `__len__` and `__getitem__` are defined--so we're going to use that. We should be aware, however, that this is a clever shortcut and we might wish to implement a proper `Dataset` if we hit limitations with it.[4]

---

[4].Here, we built the new dataset manually and also wanted to remap the classes. In some cases, it may be enough to take a subset of the indices of a given dataset. This can be accomplished using the `torch.utils .data.Subset` class. Similarly, there is `ConcatDataset` to join datasets (of compatible items) into a larger one. For iterable datasets, `ChainDataset` gives a larger, iterable dataset.

We have a dataset! Next, we need a model to feed our data to.

.2 A fully connected model

We learned how to build a neural network in chapter 5. We know that it’s a tensor of features in, a tensor of features out. After all, an image is just a set of numbers laid out in a spatial configuration. OK, we don’t know how to handle the spatial configuration part just yet, but in theory if we just take the image pixels and straighten them into a long 1D vector, we could consider those numbers as input features, right? This is what figure 7.7 illustrates.

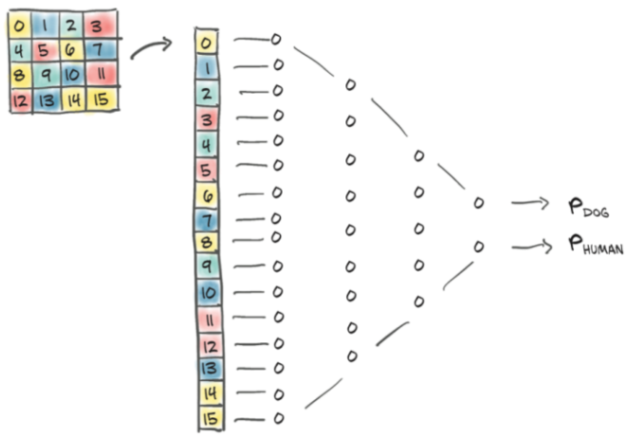


Figure 7.7 Treating our image as a 1D vector of values and training a fully connected classifier on it

Let’s try that. How many features per sample? Well, 32 × 32 × 3: that is, 3,072 input features per sample. Starting from the model we built in chapter 5, our new model would be an `nn.Linear` with 3,072 input features and some number of hidden features, followed by an activation, and then another `nn.Linear` that tapers the network down to an appropriate output number of features (2, for this use case):

```
# In[6]:
import torch.nn as nn

n_out = 2

model = nn.Sequential(
            nn.Linear(
                3072,    ❶
                512,     ❷
            ),
            nn.Tanh(),
            nn.Linear(
                512,     ❷
                n_out,   ❸
            )
        )
```

❶ Input features

❷ Hidden layer size

❸ Output classes

We somewhat arbitrarily pick 512 hidden features. A neural network needs at least one hidden layer (of activations, so two modules) with a nonlinearity in between in order to be able to learn arbitrary functions in the way we discussed in section 6.3--otherwise, it would just be a linear model. The hidden features represent (learned) relations between the inputs encoded through the weight matrix. As such, the model might learn to "compare” vector elements 176 and 208, but it does not a priori focus on them because it is structurally unaware that these are, indeed (row 5, pixel 16) and (row 6, pixel 16), and thus adjacent.

So we have a model. Next we’ll discuss what our model output should be.

.3 Output of a classifier

In chapter 6, the network produced the predicted temperature (a number with a quantitative meaning) as output. We could do something similar here: make our network output a single scalar value (so `n_out = 1`), cast the labels to floats (0.0 for airplane and 1.0 for bird), and use those as a target for `MSELoss` (the average of squared differences in the batch). Doing so, we would cast the problem into a regression problem. However, looking more closely, we are now dealing with something a bit different in nature.[5]

---

[5] Using distance on the “probability” vectors would already have been much better than using `MSELoss` with the class numbers--which, recalling our discussion of types of values in the sidebar “Continuous, ordinal, and categorical values” from chapter 4, does not make sense for categories and does not work at all in practice. Still, `MSELoss` is not very well suited to classification problems.

We need to recognize that the output is categorical: it's either a bird or an airplane (or something else if we had all 10 of the original classes). As we learned in chapter 4, when we have to represent a categorical variable, we should switch to a one-hot-encoding representation of that variable, such as `[1, 0]` for airplane or `[0, 1]` for bird (the order is arbitrary). This will still work if we have 10 classes, as in the full CIFAR-10 dataset; we'll just have a vector of length 10.[6]

---

[6].For the special binary classification case, using two values here is redundant, as one is always 1 minus the other. And indeed PyTorch lets us output only a single probability using the `nn.Sigmoid` activation at the end of the model to get a probability and the binary cross-entropy loss function `nn.BCELoss`. There also is an `nn.BCELossWithLogits` merging these two steps.

In the ideal case, the network would output `torch.tensor([1.0, 0.0])` for an airplane and `torch.tensor([0.0, 1.0])` for a bird. Practically speaking, since our classifier will not be perfect, we can expect the network to output something in between. The key realization in this case is that we can interpret our output as probabilities: the first entry is the probability of "airplane," and the second is the probability of "bird."

Casting the problem in terms of probabilities imposes a few extra constraints on the outputs of our network:

- Each element of the output must be in the `[0.0, 1.0]` range (a probability of an outcome cannot be less than 0 or greater than 1).
- The elements of the output must add up to 1.0 (we're certain that one of the two outcomes will occur).

It sounds like a tough constraint to enforce in a differentiable way on a vector of numbers. Yet there's a very smart trick that does exactly that, and it's differentiable: it's called *softmax*.

.4 Representing the output as probabilities

Softmax is a function that takes a vector of values and produces another vector of the same dimension, where the values satisfy the constraints we just listed to represent probabilities. The expression for softmax is shown in figure 7.8.

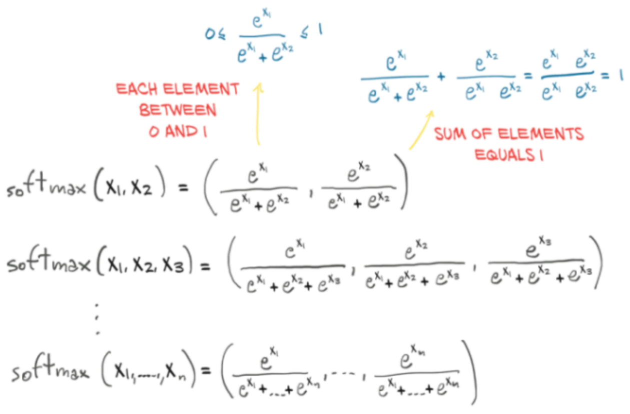


Figure 7.8 Handwritten softmax

That is, we take the elements of the vector, compute the elementwise exponential, and divide each element by the sum of exponentials. In code, it's something like this:

```
# In[7]:
def softmax(x):
    return torch.exp(x) / torch.exp(x).sum()
```

Let's test it on an input vector:

```
# In[8]:
x = torch.tensor([1.0, 2.0, 3.0])

softmax(x)

# Out[8]:
tensor([0.0900, 0.2447, 0.6652])
```

As expected, it satisfies the constraints on probability:

```
# In[9]:
softmax(x).sum()

# Out[9]:
tensor(1.)
```

Softmax is a monotone function, in that lower values in the input will correspond to lower values in the output. However, it's not *scale invariant*, in that the ratio between values is not preserved. In fact, the ratio between the first and second elements of the input is 0.5, while the ratio between the same elements in the output is 0.3678. This is not a real issue, since the learning process will drive the parameters of the model in a way that values have appropriate ratios.

The `nn` module makes softmax available as a module. Since, as usual, input tensors may have an additional batch 0th dimension, or have dimensions along which they encode probabilities and others in which they don't, `nn.Softmax` requires us to specify the dimension along which the softmax function is applied:

```
# In[10]:
softmax = nn.Softmax(dim=1)

x = torch.tensor([[1.0, 2.0, 3.0],
                  [1.0, 2.0, 3.0]])

softmax(x)

# Out[10]:
tensor([[0.0900, 0.2447, 0.6652],
        [0.0900, 0.2447, 0.6652]])
```

In this case, we have two input vectors in two rows (just like when we work with batches), so we initialize `nn.Softmax` to operate along dimension 1.

Excellent! We can now add a softmax at the end of our model, and our network will be equipped to produce probabilities:

```
# In[11]:
model = nn.Sequential(
            nn.Linear(3072, 512),
            nn.Tanh(),
            nn.Linear(512, 2),
            nn.Softmax(dim=1))
```

We can actually try running the model before even training it. Let’s do it, just to see what comes out. We first build a batch of one image, our bird (figure 7.9):

```
# In[12]:
img, _ = cifar2[0]

plt.imshow(img.permute(1, 2, 0))
plt.show()
```

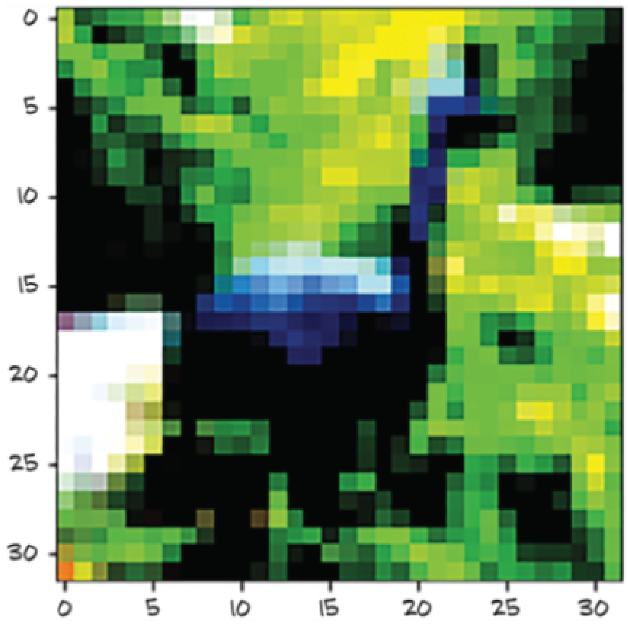


Figure 7.9 A random bird from the CIFAR-10 dataset (after normalization)

Oh, hello there. In order to call the model, we need to make the input have the right dimensions. We recall that our model expects 3,072 features in the input, and that `nn` works with data organized into batches along the zeroth dimension. So we need to turn our 3 × 32 × 32 image into a 1D tensor and then add an extra dimension in the zeroth position. We learned how to do this in chapter 3:

```
# In[13]:
img_batch = img.view(-1).unsqueeze(0)
```

Now we're ready to invoke our model:

```
# In[14]:
out = model(img_batch)
out

# Out[14]:
tensor([[0.4784, 0.5216]], grad_fn=<SoftmaxBackward>)
```

So, we got probabilities! Well, we know we shouldn't get too excited: the weights and biases of our linear layers have not been trained at all. Their elements are initialized randomly by PyTorch between -1.0 and 1.0. Interestingly, we also see `grad_fn` for the output, which is the tip of the backward computation graph (it will be used as soon as we need to backpropagate).[7]

---

7. While it is, in principle, possible to say that here the model is uncertain (because it assigns 48% and 52% probabilities to the two classes), it will turn out that typical training results in highly overconfident models. Bayesian neural networks

can provide some remedy, but they are beyond the scope of this book.

In addition, while we know which output probability is supposed to be which (recall our `class_names`), our network has no indication of that. Is the first entry “airplane” and the second “bird,” or the other way around? The network can’t even tell that at this point. It’s the loss function that associates a meaning with these two numbers, after backpropagation. If the labels are provided as index 0 for “airplane” and index 1 for “bird,” then that’s the order the outputs will be induced to take. Thus, after training, we will be able to get the label as an index by computing the *argmax* of the output probabilities: that is, the index at which we get the maximum probability. Conveniently, when supplied with a dimension, `torch.max` returns the maximum element along that dimension as well as the index at which that value occurs. In our case, we need to take the max along the probability vector (not across batches), therefore, dimension 1:

```
# In[15]:
_, index = torch.max(out, dim=1)

index

# Out[15]:
tensor([1])
```

It says the image is a bird. Pure luck. But we have adapted our model output to the classification task at hand by getting it to output probabilities. We also have now run our model against an input image and verified that our plumbing works. Time to get training. As in the previous two chapters, we need a loss to minimize during training.

.5 A loss for classifying

We just mentioned that the loss is what gives probabilities meaning. In chapters 5 and 6, we used mean square error (MSE) as our loss. We could still use MSE and make our output probabilities converge to `[0.0, 1.0]` and `[1.0, 0.0]`. However, thinking about it, we're not really interested in reproducing these values exactly. Looking back at the argmax operation we used to extract the index of the predicted class, what we're really interested in is that the first probability is higher than the second for airplanes and vice versa for birds. In other words, we want to penalize misclassifications rather than painstakingly penalize everything that doesn't look exactly like a 0.0 or 1.0.

What we need to maximize in this case is the probability associated with the correct class, `out[class_index]`, where `out` is the output of softmax and `class_index` is a vector containing 0 for "airplane" and 1 for "bird" for each sample. This quantity--that is, the probability associated with the correct class--is referred to as the *likelihood* (of our model's parameters, given the data).[8] In other words, we want a loss function that is very high when the likelihood is low: so low that the alternatives have a higher probability. Conversely, the loss should be low when the likelihood is higher than the alternatives, and we're not really fixated on driving the probability up to 1.

8.For a succinct definition of the terminology, refer to David MacKay's *Information Theory, Inference, and Learning Algorithms* (Cambridge University Press, 2003), section 2.3.

There's a loss function that behaves that way, and it's called *negative log likelihood* (NLL). It has the expression `NLL = - sum(log(out_i[c_i]))`, where the sum is taken over *N* samples and `c_i` is the correct class for sample *i*. Let's take a look at figure 7.10, which shows the NLL as a function of predicted probability.

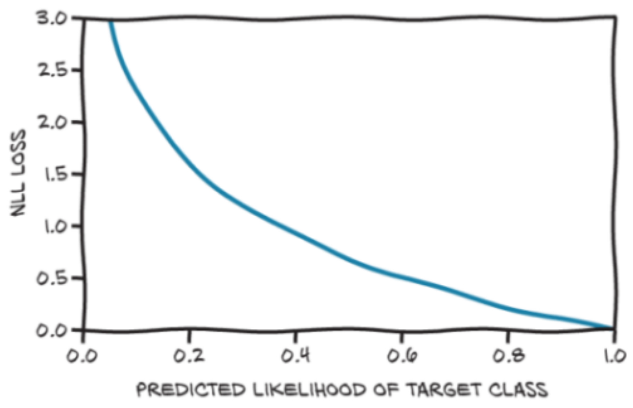


Figure 7.10 The NLL loss as a function of the predicted probabilities

The figure shows that when low probabilities are assigned to the data, the NLL grows to infinity, whereas it decreases at a rather shallow rate when probabilities are greater than 0.5. Remember that the NLL takes probabilities as input; so, as the likelihood grows, the other probabilities will necessarily decrease.

Summing up, our loss for classification can be computed as follows. For each sample in the batch:

1. Run the forward pass, and obtain the output values from the last (linear) layer.
2. Compute their softmax, and obtain probabilities.
3. Take the predicted probability corresponding to the correct class (the likelihood of the parameters). Note that we know what the correct class is because it's a supervised problem--it's our ground truth.
4. Compute its logarithm, slap a minus sign in front of it, and add it to the loss.

So, how do we do this in PyTorch? PyTorch has an `nn.NLLLoss` class. However (gotcha ahead), as opposed to what you might expect, it does not take probabilities but rather takes a tensor of log probabilities as input. It then computes the NLL of our model given the batch of data. There's a good reason behind the input convention: taking the logarithm of a probability is tricky when the probability gets close to zero. The workaround is to use `nn.LogSoftmax` instead of `nn.Softmax`, which takes care to make the calculation numerically stable.

We can now modify our model to use `nn.LogSoftmax` as the output module:

```
model = nn.Sequential(
             nn.Linear(3072, 512),
             nn.Tanh(),
             nn.Linear(512, 2),
             nn.LogSoftmax(dim=1))
```

Then we instantiate our NLL loss:

```
loss = nn.NLLLoss()
```

The loss takes the output of `nn.LogSoftmax` for a batch as the first argument and a tensor of class indices (zeros and ones, in our case) as the second argument. We can now test it with our birdie:

```
img, label = cifar2[0]
```

```
out = model(img.view(-1).unsqueeze(0))

loss(out, torch.tensor([label]))

tensor(0.6509, grad_fn=<NllLossBackward>)
```

Ending our investigation of losses, we can look at how using cross-entropy loss improves over MSE. In figure 7.11, we see that the cross-entropy loss has some slope when the prediction is off target (in the low-loss corner, the correct class is assigned a predicted probability of 99.97%), while the MSE we dismissed at the beginning saturates much earlier and--crucially--also for very wrong predictions. The underlying reason is that the slope of the MSE is too low to compensate for the flatness of the softmax function for wrong predictions. This is why the MSE for probabilities is not a good fit for classification work.

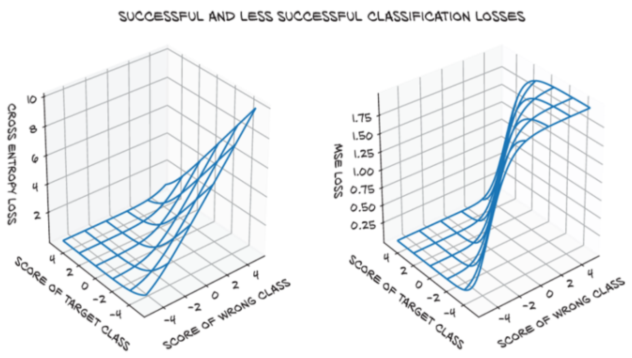


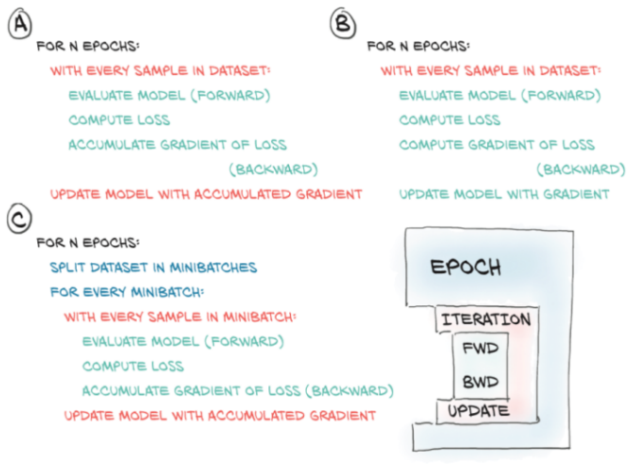
Figure 7.11 The cross entropy (left) and MSE between predicted probabilities and the target probability vector (right) as functions of the predicted scores--that is, before the (log-) softmax

.6 Training the classifier

All right! We're ready to bring back the training loop we wrote in chapter 5 and see how it trains (the process is illustrated in figure 7.12):

```
import torch
import torch.nn as nn

model = nn.Sequential(
            nn.Linear(3072, 512),
            nn.Tanh(),
            nn.Linear(512, 2),
            nn.LogSoftmax(dim=1))

learning_rate = 1e-2

optimizer = optim.SGD(model.parameters(), lr=learning_rate)

loss_fn = nn.NLLLoss()

n_epochs = 100

for epoch in range(n_epochs):
    for img, label in cifar2:
        out = model(img.view(-1).unsqueeze(0))
        loss = loss_fn(out, torch.tensor([label]))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    print("Epoch: %d, Loss: %f" % (epoch, float(loss)))    ❶
```

❶ Prints the loss for the last image. In the next chapter, we will improve our output to give an average over the entire epoch.



```
(A)
FOR N EPOCHS:
  WITH EVERY SAMPLE IN DATASET:
    EVALUATE MODEL (FORWARD)
    COMPUTE LOSS
    ACCUMULATE GRADIENT OF LOSS
                        (BACKWARD)
  UPDATE MODEL WITH ACCUMULATED GRADIENT
```

```
(B)
FOR N EPOCHS:
  WITH EVERY SAMPLE IN DATASET:
    EVALUATE MODEL (FORWARD)
    COMPUTE LOSS
    COMPUTE GRADIENT OF LOSS
                        (BACKWARD)
    UPDATE MODEL WITH GRADIENT
```

```
(C)
FOR N EPOCHS:
  SPLIT DATASET IN MINIBATCHES
  FOR EVERY MINIBATCH:
    WITH EVERY SAMPLE IN MINIBATCH:
      EVALUATE MODEL (FORWARD)
      COMPUTE LOSS
      ACCUMULATE GRADIENT OF LOSS (BACKWARD)
    UPDATE MODEL WITH ACCUMULATED GRADIENT
```

Figure 7.12 Training loops: (A) averaging updates over the whole dataset; (B) updating the model at each sample; (C) averaging updates over minibatches

Looking more closely, we made a small change to the training loop. In chapter 5, we had just one loop: over the epochs (recall that an epoch ends when all samples in the training set have been evaluated). We figured that evaluating all 10,000 images in a single batch would be too much, so we decided to have an inner loop where we evaluate one sample at a time and backpropagate over that single sample.

While in the first case the gradient is accumulated over all samples before being applied, in this case we apply changes to parameters based on a very partial estimation of the gradient on a single sample. However, what is a good direction for reducing the loss based on one sample might not be a good direction for others. By shuffling samples at each epoch and estimating the gradient on one or (preferably, for stability) a few samples at a time, we are effectively introducing randomness in our gradient descent. Remember SGD? It stands for *stochastic gradient descent*, and this is what the *S* is about: working on small batches (aka minibatches) of shuffled data. It turns out that following gradients estimated over minibatches, which are poorer approximations of gradients estimated across the whole dataset, helps convergence and prevents the optimization process from getting stuck in local minima it encounters along the way. As depicted in figure 7.13, gradients from minibatches are randomly off the ideal trajectory, which is part of the reason why we want to use a reasonably small learning rate. Shuffling the dataset at each epoch helps ensure that the sequence of gradients estimated over minibatches is representative of the gradients computed across the full dataset.

Typically, minibatches are a constant size that we need to set prior to training, just like the learning rate. These are called *hyperparameters*, to distinguish them from the parameters of a model.

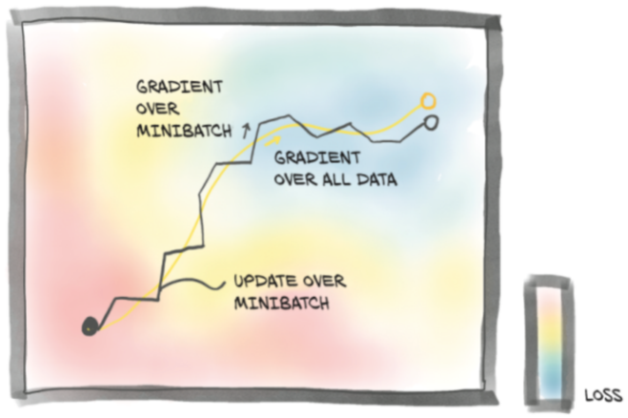


Figure 7.13 Gradient descent averaged over the whole dataset (light path) versus stochastic gradient descent, where the gradient is estimated on randomly picked minibatches

In our training code, we chose minibatches of size 1 by picking one item at a time from the dataset. The `torch.utils.data` module has a class that helps with shuffling and organizing the data in minibatches: `DataLoader`. The job of a data loader is to sample minibatches from a dataset, giving us the flexibility to choose from different sampling strategies. A very common strategy is uniform sampling after shuffling the data at each epoch. Figure 7.14 shows the data loader shuffling the indices it gets from the `Dataset`.

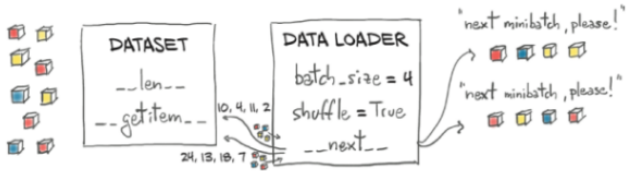


Figure 7.14 A data loader dispensing minibatches by using a dataset to sample individual data items

Let's see how this is done. At a minimum, the `DataLoader` constructor takes a `Dataset` object as input, along with `batch_size` and a `shuffle` Boolean that indicates whether the data needs to be shuffled at the beginning of each epoch:

```
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)
```

A `DataLoader` can be iterated over, so we can use it directly in the inner loop of our new training code:

```
import torch
import torch.nn as nn

train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,
                                           shuffle=True)

model = nn.Sequential(
            nn.Linear(3072, 512),
            nn.Tanh(),
            nn.Linear(512, 2),
            nn.LogSoftmax(dim=1))

learning_rate = 1e-2

optimizer = optim.SGD(model.parameters(), lr=learning_rate)

loss_fn = nn.NLLLoss()

n_epochs = 100

for epoch in range(n_epochs):
    for imgs, labels in train_loader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
        loss = loss_fn(outputs, labels)
```

```
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        print("Epoch: %d, Loss: %f" % (epoch, float(loss)))    ❶
```

❶ Due to the shuffling, this now prints the loss for a random batch--clearly something we want to improve in chapter 8

At each inner iteration, `imgs` is a tensor of size 64 × 3 × 32 × 32--that is, a minibatch of 64 (32 × 32) RGB images--while `labels` is a tensor of size 64 containing label indices.

Let's run our training:

```
Epoch: 0, Loss: 0.523478
Epoch: 1, Loss: 0.391083
Epoch: 2, Loss: 0.407412
Epoch: 3, Loss: 0.364203
...
Epoch: 96, Loss: 0.019537
Epoch: 97, Loss: 0.008973
Epoch: 98, Loss: 0.002607
Epoch: 99, Loss: 0.026200
```

We see that the loss decreases somehow, but we have no idea whether it's low enough. Since our goal here is to correctly assign classes to images, and preferably do that on an independent dataset, we can compute the accuracy of our model on the validation set in terms of

the number of correct classifications
over the total:

```
val_loader = torch.utils.data.DataLoader(cifar2_val, batch_size=64,
                                         shuffle=False)

correct = 0
total = 0

with torch.no_grad():
    for imgs, labels in val_loader:
        batch_size = imgs.shape[0]
        outputs = model(imgs.view(batch_size, -1))
        _, predicted = torch.max(outputs, dim=1)
        total += labels.shape[0]
        correct += int((predicted == labels).sum())

print("Accuracy: %f", correct / total)

Accuracy: 0.794000
```

Not a great performance, but quite a lot better than random. In our defense, our model was quite a shallow classifier; it's a miracle that it worked at all. It did because our dataset is really simple--a lot of the samples in the two classes likely have systematic differences (such as the color of the background) that help the model tell birds from airplanes, based on a few pixels.

We can certainly add some bling to our model by including more layers, which will increase the model's depth and capacity. One rather arbitrary possibility is

```
model = nn.Sequential(
            nn.Linear(3072, 1024),
            nn.Tanh(),
            nn.Linear(1024, 512),
            nn.Tanh(),
            nn.Linear(512, 128),
            nn.Tanh(),
            nn.Linear(128, 2),
            nn.LogSoftmax(dim=1))
```

Here we are trying to taper the number of features more gently toward the output, in the hope that intermediate layers will do a better job of squeezing information in increasingly shorter intermediate outputs.

The combination of `nn.LogSoftmax` and `nn.NLLLoss` is equivalent to using `nn.CrossEntropyLoss`. This terminology is a particularity of PyTorch, as the `nn.NLLoss` computes, in fact, the cross entropy but with log probability predictions as inputs where `nn.CrossEntropyLoss` takes scores (sometimes called *logits*). Technically, `nn.NLLLoss` is the cross entropy between the Dirac distribution, putting all mass on the target, and the predicted distribution given by the log probability inputs.

To add to the confusion, in information theory, up to normalization by sample size, this cross entropy can be interpreted as a negative log likelihood of the predicted distribution under the target distribution as an outcome. So both losses are the negative log likelihood of the model parameters given the data when our model predicts the (softmax-applied) probabilities. In this book, we won't rely on these details, but don't let the PyTorch naming confuse you when you see the terms used in the literature.

It is quite common to drop the last `nn.LogSoftmax` layer from the network and use `nn.CrossEntropyLoss` as a loss. Let us try that:

```
model = nn.Sequential(
            nn.Linear(3072, 1024),
            nn.Tanh(),
            nn.Linear(1024, 512),
            nn.Tanh(),
            nn.Linear(512, 128),
            nn.Tanh(),
            nn.Linear(128, 2))

loss_fn = nn.CrossEntropyLoss()
```

Note that the numbers will be *exactly* the same as with `nn.LogSoftmax` and `nn.NLLLoss`. It's just more convenient to do it all in one pass, with the only gotcha being that the output of our mod-el will not be interpretable as probabili-ties (or log probabilities). We'll need to explicitly pass the output through a soft-max to obtain those.

Training this model and evaluating the accuracy on the validation set (0.802000) lets us appreciate that a larger model bought us an increase in accuracy, but not that much. The accuracy on the train-ing set is practically perfect (0.998100). What is this telling us? That we are overfitting our model in both cases. Our fully connected model is finding a way to discriminate birds and airplanes on the training set by memorizing the training set, but performance on the validation set is not all that great, even if we choose a larger model.

PyTorch offers a quick way to determine how many parameters a model has through the `parameters()` method of `nn.Model` (the same method we use to provide the parameters to the optimizer). To find out how many elements are in each tensor instance, we can call the `numel` method. Summing those gives us our total count. Depending on our use case, counting parameters might require us to check whether a parameter has `requires_grad` set to `True`, as well. We might want to differentiate the number of *trainable* parameters from the overall model size. Let's take a look at what we have right now:

```
# In[7]:
numel_list = [p.numel()
              for p in connected_model.parameters()
              if p.requires_grad == True]
sum(numel_list), numel_list

# Out[7]:
(3737474, [3145728, 1024, 524288, 512, 65536, 128, 256, 2])
```

Wow, 3.7 million parameters! Not a small network for such a small input image, is it? Even our first network was pretty large:

```
# In[9]:
numel_list = [p.numel() for p in first_model.parameters()]
sum(numel_list), numel_list

# Out[9]:
(1574402, [1572864, 512, 1024, 2])
```

The number of parameters in our first model is roughly half that in our latest model. Well, from the list of individual parameter sizes, we start having an idea what's responsible: the first module, which has 1.5 million parameters. In our full network, we had 1,024 output features, which led the first linear module to have 3 million parameters. This shouldn't be unexpected: we know that a linear layer computes `y = weight * x + bias`, and if `x` has length 3,072 (disregarding the batch dimension for simplicity) and `y` must have length 1,024, then the `weight` tensor needs to be of size 1,024 × 3,072 and the `bias` size must be 1,024. And 1,024 * 3,072 + 1,024 = 3,146,752, as we found earlier. We can verify these quantities directly:

```
# In[10]:
linear = nn.Linear(3072, 1024)

linear.weight.shape, linear.bias.shape

# Out[10]:
(torch.Size([1024, 3072]), torch.Size([1024]))
```

What is this telling us? That our neural network won't scale very well with the number of pixels. What if we had a 1,024 × 1,024 RGB image? That's 3.1 million input values. Even abruptly going to 1,024 hidden features (which is not going to work for our classifier), we would have over 3 *billion* parameters. Using 32-bit floats, we're already at 12 GB of RAM, and we haven't even hit the second layer, much less computed and stored the gradients. That's just not going to fit on most present-day GPUs.

.7 The limits of going fully connected

Let's reason about what using a linear module on a 1D view of our image entails--figure 7.15 shows what is going on. It's like taking every single input value--that is, every single component in our RGB image--and computing a linear combination of it with all the other values for every output feature. On one hand, we are allowing for the combination of any pixel with every other pixel in the image being potentially relevant for our task. On the other hand, we aren't utilizing the relative position of neighboring or faraway pixels, since we are treating the image as one big vector of numbers.

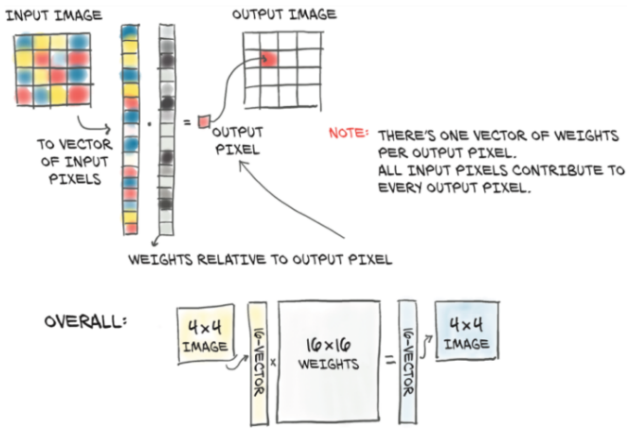

INPUT IMAGE
OUTPUT IMAGE
TO VECTOR OF INPUT PIXELS
OUTPUT PIXEL
NOTE: THERE'S ONE VECTOR OF WEIGHTS PER OUTPUT PIXEL.
ALL INPUT PIXELS CONTRIBUTE TO EVERY OUTPUT PIXEL.
WEIGHTS RELATIVE TO OUTPUT PIXEL
OVERALL:
4×4 IMAGE
16-VECTOR
16×16 WEIGHTS
16-VECTOR
4×4 IMAGE

Figure 7.15 Using a fully connected module with an input image: every input pixel is combined with every other to produce each element in the output.

An airplane flying in the sky captured in a 32 × 32 image will be very roughly similar to a dark, cross-like shape on a blue background. A fully connected network as in figure 7.15 would need to learn that when pixel 0,1 is dark, pixel 1,1 is also dark, and so on, that's a good indication of an airplane. This is illustrated in the top half of figure 7.16. However, shift the same airplane by one pixel or more as in the bottom half of the figure, and the relationships between pixels will have to be relearned from scratch: this time, an airplane is likely when pixel 0,2 is dark, pixel 1,2 is dark, and so on. In more technical terms, a fully connected network is not *translation invariant*. This means a network that has been trained to recognize a Spitfire starting at position 4,4 will not be able to recognize the *exact same* Spitfire starting at position 8,8. We would then have to *augment* the dataset--that is, apply random translations to images during training--so the network would have a chance to see Spitfires all over the image, and we would need to do this for every image in the dataset (for the record, we could concatenate a transform from `torchvision.transforms` to do this transparently). However, this *data augmentation* strategy comes at a cost: the number of hidden features--that is, of parameters--must be large enough to store the information about all of these translated replicas.

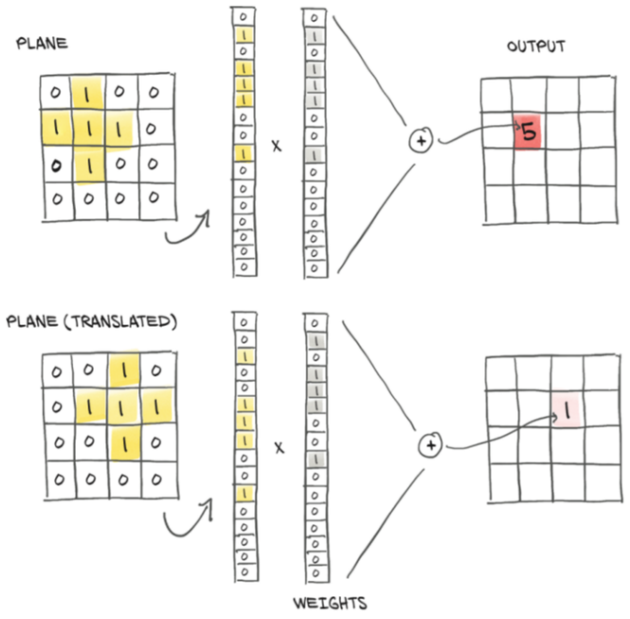


Figure 7.16 Translation invariance, or the lack thereof, with fully connected layers

So, at the end of this chapter, we have a dataset, a model, and a training loop, and our model learns. However, due to a mismatch between our problem and our network structure, we end up overfitting our training data, rather than learning the generalized features of what we want the model to detect.

We've created a model that allows for relating every pixel to every other pixel in the image, regardless of their spatial arrangement. We have a reasonable assumption that pixels that are closer together are in theory a lot more related, though. This means we are training a classifier that is not translation-invariant, so we're forced to use a lot of capacity for learning translated replicas if we want to hope to do well on the validation set. There has to be a better way, right?

Of course, most such questions in a book like this are rhetorical. The solution to our current set of problems is to change our model to use convolutional layers. We'll cover what that means in the next chapter.

Conclusion

In this chapter, we have solved a simple classification problem from dataset, to model, to minimizing an appropriate loss in a training loop. All of these things will be standard tools for your PyTorch toolbelt, and the skills needed to use them will be useful throughout your PyTorch tenure.

We've also found a severe shortcoming of our model: we have been treating 2D images as 1D data. Also, we do not have a natural way to incorporate the translation invariance of our problem. In the next chapter, you'll learn how to exploit the 2D nature of image data to get much better results.[9]

9. The same caveat about translation invariance also applies to purely 1D data: an audio classifier should likely produce the same output even if the sound to be classified starts a tenth of a second earlier or later.

We could use what we have learned right away to process data without this translation invariance. For example, using it on tabular data or the time-series data we met in chapter 4, we can probably do great things already. To some extent, it would also be possible to use it on text data that is appropriately represented.[10]

10. *Bag-of-words models*, which just average over word embeddings, can be processed with the network design from this chapter. More contemporary models take the positions of the words into account and need more advanced models.

Exercises

1. Use `torchvision` to implement random cropping of the data.
    1. How are the resulting images different from the uncropped originals?
    2. What happens when you request the same image a second time?
    3. What is the result of training using randomly cropped images?

2. Switch loss functions (perhaps MSE).
    1. Does the training behavior change?
    2. Is it possible to reduce the capacity of the network enough that it stops overfitting?
    3. How does the model perform on the validation set when doing so?

ummary

- Computer vision is one of the most extensive applications of deep learning.
- Several datasets of annotated images are publicly available; many of them can be accessed via `torchvision`.
- `Dataset`s and `DataLoader`s provide a simple yet effective abstraction for loading and sampling datasets.
- For a classification task, using the softmax function on the output of a network produces values that satisfy the requirements for being interpreted as probabilities. The ideal loss function for classification in this case is obtained by using the output of softmax as the input of a non-negative log likelihood function. The combination of softmax and such loss is called cross entropy in PyTorch.
- Nothing prevents us from treating images as vectors of pixel values, dealing with them using a fully connected network, just like any other numerical data. However, doing so makes it much harder to take advantage of the spatial relationships in the data.

- Simple models can be created using `nn.Sequential`.
