2 Pretrained networks

This chapter covers

- Running pretrained image-recognition models
- An introduction to GANs and CycleGAN
- Captioning models that can produce text descriptions of images
- Sharing models through Torch Hub

We closed our first chapter promising to unveil amazing things in this chapter, and now it's time to deliver. Computer vision is certainly one of the fields that have been most impacted by the advent of deep learning, for a variety of reasons. The need to classify or interpret the content of natural images existed, very large datasets became available, and new constructs such as convolutional layers were invented and could be run quickly on GPUs with unprecedented accuracy. All of these factors combined with the internet giants' desire to understand pictures taken by millions of users with their mobile devices and managed on said giants' platforms. Quite the perfect storm.

We are going to learn how to use the work of the best researchers in the field by downloading and running very interesting models that have already been trained on open, large-scale datasets. We can think of a pretrained neural network as similar to

a program that takes inputs and generates outputs. The behavior of such a program is dictated by the architecture of the neural network and by the examples it saw during training, in terms of desired input-output pairs, or desired properties that the output should satisfy. Using an off-the-shelf model can be a quick way to jump-start a deep learning project, since it draws on expertise from the researchers who designed the model, as well as the computation time that went into training the weights.

In this chapter, we will explore three popular pretrained models: a model that can label an image according to its content, another that can fabricate a new image from a real image, and a model that can describe the content of an image using proper English sentences. We will learn how to load and run these pretrained models in PyTorch, and we will introduce PyTorch Hub, a set of tools through which PyTorch models like the pretrained ones we'll discuss can be easily made available through a uniform interface. Along the way, we'll discuss data sources, define terminology like *label*, and attend a zebra rodeo.

If you're coming to PyTorch from another deep learning framework, and you'd rather jump right into learning the nuts and bolts of PyTorch, you can get away with skipping to the next chapter. The things we'll cover in this chapter are more fun than foundational and are somewhat independent of any given deep learning tool. That's not to say they're not important! But if you've worked with pretrained models in other deep learning frameworks, then you already know how powerful a tool they can be. And if you're already familiar with the generative adversarial network (GAN) game, you don't need us to explain it to you.

We hope you keep reading, though, since this chapter hides some important skills under the fun. Learning how to run a pretrained model using PyTorch is a useful skill—full stop. It's especially useful if the model has been trained on a large dataset. We will need to get accustomed to the mechanics of obtaining and running a neural network on real-world data, and then visualizing and evaluating its outputs, whether we trained it or not.

2.1 A pretrained network that recognizes the subject of an image

As our first foray into deep learning, we'll run a state-of-the-art deep neural network that was pretrained on an object-recognition task. There are many pretrained networks that can be accessed through source code repositories. It is common for researchers to publish their source code along with their papers, and often the code comes with weights that were obtained by training a model on a reference dataset. Using one of these models could enable us to, for example, equip our next web service with image-recognition capabilities with very little effort.

The pretrained network we'll explore here was trained on a subset of the ImageNet dataset (http://imagenet.stanford.edu). ImageNet is a very large dataset of over 14 million images maintained by Stanford University. All of the images are labeled with a hierarchy of nouns that come from the WordNet dataset (http://wordnet.princeton.edu), which is in turn a large lexical database of the English language.

The ImageNet dataset, like several other public datasets, has its origin in academic competitions. Competitions have traditionally been some of the main playing fields where researchers at institutions and companies regularly challenge each other. Among others, the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) has gained popularity since its inception in 2010. This particular competition is based on a few tasks, which can vary each year, such as image classification (telling what object categories the image contains), object localization (identifying objects' position in images), object detection (identifying and labeling objects in images), scene classification (classifying a situation in an image), and scene parsing (segmenting an image into regions associated with semantic categories, such as cow, house, cheese, hat). In particular, the image-classification task consists of taking an input image and producing a list of 5 labels out of 1,000 total categories, ranked by confidence, describing the content of the image.

The training set for ILSVRC consists of 1.2 million images labeled with one of 1,000 nouns (for example, "dog"), referred to as the *class* of the image. In this sense, we will use the terms *label* and *class* interchangeably. We can take a peek at images from ImageNet in figure 2.1.

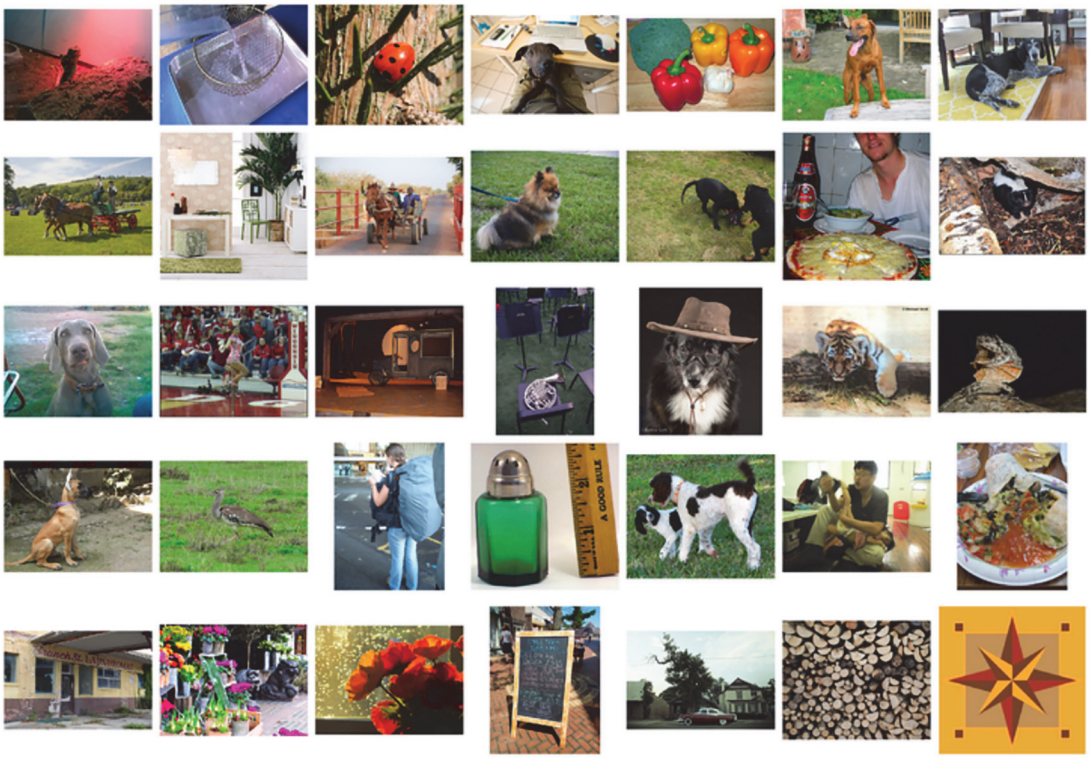

Figure 2.1 A small sample of ImageNet images

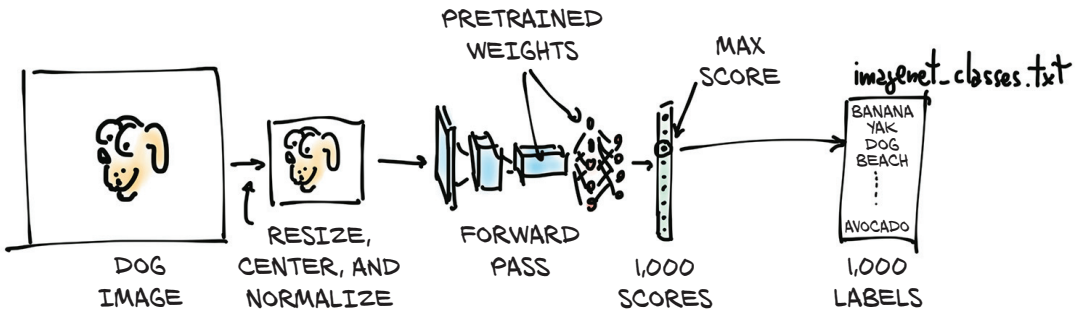

Figure 2.2 The inference process

We are going to end up being able to take our own images and feed them into our pretrained model, as pictured in figure 2.2. This will result in a list of predicted labels for that image, which we can then examine to see what the model thinks our image is. Some images will have predictions that are accurate, and others will not!

The input image will first be preprocessed into an instance of the multidimensional array class `torch.Tensor`. It is an RGB image with height and width, so this tensor will have three dimensions: the three color channels, and two spatial image dimensions of a specific size. (We'll get into the details of what a tensor is in chapter 3, but for now, think of it as being like a vector or matrix of floating-point numbers.) Our model will take that processed input image and pass it into the pretrained network to obtain scores for each class. The highest score corresponds to the most likely class according to the weights. Each class is then mapped one-to-one onto a class label. That output is contained in a `torch.Tensor` with 1,000 elements, each representing the score associated with that class.

Before we can do all that, we'll need to get the network itself, take a peek under the hood to see how it's structured, and learn about how to prepare our data before the model can use it.

2.1.1 Obtaining a pretrained network for image recognition

As discussed, we will now equip ourselves with a network trained on ImageNet. To do so, we'll take a look at the TorchVision project (https://github.com/pytorch/vision), which contains a few of the best-performing neural network architectures for computer vision, such as AlexNet (http://mng.bz/lo6z), ResNet (https://arxiv.org/pdf/1512.03385.pdf), and Inception v3 (https://arxiv.org/pdf/1512.00567.pdf). It also has easy access to datasets like ImageNet and other utilities for getting up to speed with computer vision applications in PyTorch. We'll dive into some of these further along in the book. For now, let's load up and run two networks: first AlexNet, one of the early breakthrough networks for image recognition; and then a residual network, ResNet for short, which won the ImageNet classification, detection, and localization

competitions, among others, in 2015. If you didn't get PyTorch up and running in chapter 1, now is a good time to do that.

The predefined models can be found in `torchvision.models` (code/p1ch2/2_pre_trained_networks.ipynb):

```
# In[1]:
from torchvision import models
```

We can take a look at the actual models:

```
# In[2]:
dir(models)

# Out[2]:
['AlexNet',
 'DenseNet',
 'Inception3',
 'ResNet',
 'SqueezeNet',
 'VGG',
...
 'alexnet',
 'densenet',
 'densenet121',
...
 'resnet',
 'resnet101',
 'resnet152',
...
 ]
```

The capitalized names refer to Python classes that implement a number of popular models. They differ in their architecture—that is, in the arrangement of the operations occurring between the input and the output. The lowercase names are convenience functions that return models instantiated from those classes, sometimes with different parameter sets. For instance, `resnet101` returns an instance of `ResNet` with 101 layers, `resnet18` has 18 layers, and so on. We'll now turn our attention to AlexNet.

2.1.2 AlexNet

The AlexNet architecture won the 2012 ILSVRC by a large margin, with a top-5 test error rate (that is, the correct label must be in the top 5 predictions) of 15.4%. By comparison, the second-best submission, which wasn't based on a deep network, trailed at 26.2%. This was a defining moment in the history of computer vision: the moment when the community started to realize the potential of deep learning for vision tasks. That leap was followed by constant improvement, with more modern architectures and training methods getting top-5 error rates as low as 3%.

By today's standards, AlexNet is a rather small network, compared to state-of-the-art models. But in our case, it's perfect for taking a first peek at a neural network that does something and learning how to run a pretrained version of it on a new image.

We can see the structure of AlexNet in figure 2.3. Not that we have all the elements for understanding it now, but we can anticipate a few aspects. First, each block consists of a bunch of multiplications and additions, plus a sprinkle of other functions in the output that we'll discover in chapter 5. We can think of it as a filter—a function that takes one or more images as input and produces other images as output. The way it does so is determined during training, based on the examples it has *seen* and on the desired outputs for those.

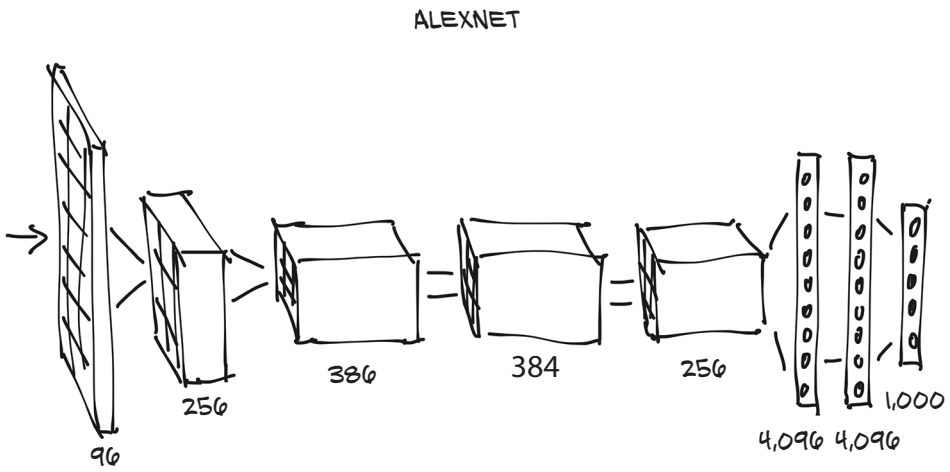

Figure 2.3 The AlexNet architecture

In figure 2.3, input images come in from the left and go through five stacks of filters, each producing a number of output images. After each filter, the images are reduced in size, as annotated. The images produced by the last stack of filters are laid out as a 4,096-element 1D vector and classified to produce 1,000 output probabilities, one for each output class.

In order to run the AlexNet architecture on an input image, we can create an instance of the `AlexNet` class. This is how it's done:

```
# In[3]:
alexnet = models.AlexNet()
```

At this point, `alexnet` is an object that can run the AlexNet architecture. It's not essential for us to understand the details of this architecture for now. For the time being, `AlexNet` is just an opaque object that can be called like a function. By providing

`alexnet` with some precisely sized input data (we'll see shortly what this input data should be), we will run a *forward pass* through the network. That is, the input will run through the first set of neurons, whose outputs will be fed to the next set of neurons, all the way to the final output. Practically speaking, assuming we have an `input` object of the right type, we can run the forward pass with `output = alexnet(input)`.

But if we did that, we would be feeding data through the whole network to produce … garbage! That's because the network is uninitialized: its weights, the numbers by which inputs are added and multiplied, have not been trained on anything—the network itself is a blank (or rather, *random*) slate. We'd need to either train it from scratch or load weights from prior training, which we'll do now.

To this end, let's go back to the `models` module. We learned that the uppercase names correspond to classes that implement popular architectures for computer vision. The lowercase names, on the other hand, are functions that instantiate models with predefined numbers of layers and units and optionally download and load pretrained weights into them. Note that there's nothing essential about using one of these functions: they just make it convenient to instantiate the model with a number of layers and units that matches how the pretrained networks were built.

2.1.3 ResNet

Using the `resnet101` function, we'll now instantiate a 101-layer convolutional neural network. Just to put things in perspective, before the advent of residual networks in 2015, achieving stable training at such depths was considered extremely hard. Residual networks pulled a trick that made it possible, and by doing so, beat several benchmarks in one sweep that year.

Let's create an instance of the network now. We'll pass an argument that will instruct the function to download the weights of `resnet101` trained on the ImageNet dataset, with 1.2 million images and 1,000 categories:

```
# In[4]:
resnet = models.resnet101(pretrained=True)
```

While we're staring at the download progress, we can take a minute to appreciate that `resnet101` sports 44.5 million parameters—that's a lot of parameters to optimize automatically!

2.1.4 Ready, set, almost run

OK, what did we just get? Since we're curious, we'll take a peek at what a `resnet101` looks like. We can do so by printing the value of the returned model. This gives us a textual representation of the same kind of information we saw in 2.3, providing details about the structure of the network. For now, this will be information overload, but as we progress through the book, we'll increase our ability to understand what this code is telling us:

```
# In[5]:
resnet

# Out[5]:
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
                  bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
                     track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
                       ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
...
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

What we are seeing here is `modules`, one per line. Note that they have nothing in common with Python modules: they are individual operations, the building blocks of a neural network. They are also called *layers* in other deep learning frameworks.

If we scroll down, we'll see a lot of `Bottleneck` modules repeating one after the other (101 of them!), containing convolutions and other modules. That's the anatomy of a typical deep neural network for computer vision: a more or less sequential cascade of filters and nonlinear functions, ending with a layer (`fc`) producing scores for each of the 1,000 output classes (`out_features`).

The `resnet` variable can be called like a function, taking as input one or more images and producing an equal number of scores for each of the 1,000 ImageNet classes. Before we can do that, however, we have to preprocess the input images so they are the right size and so that their values (colors) sit roughly in the same numerical range. In order to do that, the `torchvision` module provides `transforms`, which allow us to quickly define pipelines of basic preprocessing functions:

```
# In[6]:
from torchvision import transforms
preprocess = transforms.Compose([
        transforms.Resize(256),
        transforms.CenterCrop(224),
        transforms.ToTensor(),
        transforms.Normalize(
            mean=[0.485, 0.456, 0.406],
            std=[0.229, 0.224, 0.225]
        )])
```

In this case, we defined a `preprocess` function that will scale the input image to 256 × 256, crop the image to 224 × 224 around the center, transform it to a tensor (a PyTorch multidimensional array: in this case, a 3D array with color, height, and

width), and normalize its RGB (red, green, blue) components so that they have defined means and standard deviations. These need to match what was presented to the network during training, if we want the network to produce meaningful answers. We'll go into more depth about transforms when we dive into making our own image-recognition models in section 7.1.3.

We can now grab a picture of our favorite dog (say, bobby.jpg from the GitHub repo), preprocess it, and then see what ResNet thinks of it. We can start by loading an image from the local filesystem using Pillow (https://pillow.readthedocs.io/en/stable), an image-manipulation module for Python:

```
# In[7]:
from PIL import Image
img = Image.open("../data/p1ch2/bobby.jpg")
```

If we were following along from a Jupyter Notebook, we would do the following to see the picture inline (it would be shown where the `<PIL.JpegImagePlugin…` is in the following):

```
# In[8]:
img
# Out[8]:
<PIL.JpegImagePlugin.JpegImageFile image mode=RGB size=1280x720 at
 0x1B1601360B8>
```

Otherwise, we can invoke the `show` method, which will pop up a window with a viewer, to see the image shown in figure 2.4:

```
>>> img.show()
```

Figure 2.4 Bobby, our very special input image

Next, we can pass the image through our preprocessing pipeline:

```
# In[9]:
img_t = preprocess(img)
```

Then we can reshape, crop, and normalize the input tensor in a way that the network expects. We'll understand more of this in the next two chapters; hold tight for now:

```
# In[10]:
import torch
batch_t = torch.unsqueeze(img_t, 0)
```

We're now ready to run our model.

2.1.5 Run!

The process of running a trained model on new data is called *inference* in deep learning circles. In order to do inference, we need to put the network in `eval` mode:

```
# In[11]:
resnet.eval()

# Out[11]:
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
                  bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
                     track_running_stats=True)
  (relu): ReLU(inplace)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
                       ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
...
    )
  )
  (avgpool): AvgPool2d(kernel_size=7, stride=1, padding=0)
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

If we forget to do that, some pretrained models, like *batch normalization* and *dropout*, will not produce meaningful answers, just because of the way they work internally. Now that `eval` has been set, we're ready for inference:

```
# In[12]:
out = resnet(batch_t)
out

# Out[12]:
tensor([[ -3.4803,  -1.6618,  -2.4515,  -3.2662,  -3.2466,  -1.3611,
          -2.0465,  -2.5112,  -1.3043,  -2.8900,  -1.6862,  -1.3055,
...
           2.8674,  -3.7442,   1.5085,  -3.2500,  -2.4894,  -0.3354,
           0.1286,  -1.1355,   3.3969,   4.4584]])
```

A staggering set of operations involving 44.5 million parameters has just happened, producing a vector of 1,000 scores, one per ImageNet class. That didn't take long, did it?

We now need to find out the label of the class that received the highest score. This will tell us what the model saw in the image. If the label matches how a human would describe the image, that's great! It means everything is working. If not, then either something went wrong during training, or the image is so different from what the model expects that the model can't process it properly, or there's some other similar issue.

To see the list of predicted labels, we will load a text file listing the labels in the same order they were presented to the network during training, and then we will pick out the label at the index that produced the highest score from the network. Almost all models meant for image recognition have output in a form similar to what we're about to work with.

Let's load the file containing the 1,000 labels for the ImageNet dataset classes:

```
# In[13]:
with open('../data/p1ch2/imagenet_classes.txt') as f:
    labels = [line.strip() for line in f.readlines()]
```

At this point, we need to determine the index corresponding to the maximum score in the `out` tensor we obtained previously. We can do that using the `max` function in PyTorch, which outputs the maximum value in a tensor as well as the indices where that maximum value occurred:

```
# In[14]:
_, index = torch.max(out, 1)
```

We can now use the index to access the label. Here, `index` is not a plain Python number, but a one-element, one-dimensional tensor (specifically, `tensor([207])`), so we need to get the actual numerical value to use as an index into our `labels` list using `index[0]`. We also use `torch.nn.functional.softmax` (http://mng.bz/BYnq) to normalize our outputs to the range [0, 1], and divide by the sum. That gives us something roughly akin to the confidence that the model has in its prediction. In this case, the model is 96% certain that it knows what it's looking at is a golden retriever:

```
# In[15]:
percentage = torch.nn.functional.softmax(out, dim=1)[0] * 100
labels[index[0]], percentage[index[0]].item()

# Out[15]:
('golden retriever', 96.29334259033203)
```

Uh oh, who's a good boy?

Since the model produced scores, we can also find out what the second best, third best, and so on were. To do this, we can use the `sort` function, which sorts the values in ascending or descending order and also provides the indices of the sorted values in the original array:

```
# In[16]:
_, indices = torch.sort(out, descending=True)
[(labels[idx], percentage[idx].item()) for idx in indices[0][:5]]

# Out[16]:
[('golden retriever', 96.29334259033203),
 ('Labrador retriever', 2.80812406539917),
 ('cocker spaniel, English cocker spaniel, cocker', 0.28267428278923035),
 ('redbone', 0.2086310237646103),
 ('tennis ball', 0.11621569097042084)]
```

We see that the first four are dogs (redbone is a breed; who knew?), after which things start to get funny. The fifth answer, "tennis ball," is probably because there are enough pictures of tennis balls with dogs nearby that the model is essentially saying, "There's a 0.1% chance that I've completely misunderstood what a tennis ball is." This is a great example of the fundamental differences in how humans and neural networks view the world, as well as how easy it is for strange, subtle biases to sneak into our data.

Time to play! We can go ahead and interrogate our network with random images and see what it comes up with. How successful the network will be will largely depend on whether the subjects were well represented in the training set. If we present an image containing a subject outside the training set, it's quite possible that the network will come up with a wrong answer with pretty high confidence. It's useful to experiment and get a feel for how a model reacts to unseen data.

We've just run a network that won an image-classification competition in 2015. It learned to recognize our dog from examples of dogs, together with a ton of other real-world subjects. We'll now see how different architectures can achieve other kinds of tasks, starting with image generation.

2.2 A pretrained model that fakes it until it makes it

Let's suppose, for a moment, that we're career criminals who want to move into selling forgeries of "lost" paintings by famous artists. We're criminals, not painters, so as we paint our fake Rembrandts and Picassos, it quickly becomes apparent that they're amateur imitations rather than the real deal. Even if we spend a bunch of time practicing until we get a canvas that *we* can't tell is fake, trying to pass it off at the local art auction house is going to get us kicked out instantly. Even worse, being told "This is clearly fake; get out," doesn't help us improve! We'd have to randomly try a bunch of things, gauge which ones took *slightly* longer to recognize as forgeries, and emphasize those traits on our future attempts, which would take far too long.

Instead, we need to find an art historian of questionable moral standing to inspect our work and tell us exactly what it was that tipped them off that the painting wasn't legit. With that feedback, we can improve our output in clear, directed ways, until our sketchy scholar can no longer tell our paintings from the real thing.

Soon, we'll have our "Botticelli" in the Louvre, and their Benjamins in our pockets. We'll be rich!

While this scenario is a bit farcical, the underlying technology is sound and will likely have a profound impact on the perceived veracity of digital data in the years to come. The entire concept of "photographic evidence" is likely to become entirely suspect, given how easy it will be to automate the production of convincing, yet fake, images and video. The only key ingredient is data. Let's see how this process works.

2.2.1 The GAN game

In the context of deep learning, what we've just described is known as *the GAN game*, where two networks, one acting as the painter and the other as the art historian, compete to outsmart each other at creating and detecting forgeries. GAN stands for *generative adversarial network*, where *generative* means something is being created (in this case, fake masterpieces), *adversarial* means the two networks are competing to outsmart the other, and well, *network* is pretty obvious. These networks are one of the most original outcomes of recent deep learning research.

Remember that our overarching goal is to produce synthetic examples of a class of images that cannot be recognized as fake. When mixed in with legitimate examples, a skilled examiner would have trouble determining which ones are real and which are our forgeries.

The *generator* network takes the role of the painter in our scenario, tasked with producing realistic-looking images, starting from an arbitrary input. The *discriminator* network is the amoral art inspector, needing to tell whether a given image was fabricated by the generator or belongs in a set of real images. This two-network design is atypical for most deep learning architectures but, when used to implement a GAN game, can lead to incredible results.

Figure 2.5 shows a rough picture of what's going on. The end goal for the generator is to fool the discriminator into mixing up real and fake images. The end goal for the discriminator is to find out when it's being tricked, but it also helps inform the generator about the identifiable mistakes in the generated images. At the start, the generator produces confused, three-eyed monsters that look nothing like a Rembrandt portrait. The discriminator is easily able to distinguish the muddled messes from the real paintings. As training progresses, information flows back from the discriminator, and the generator uses it to improve. By the end of training, the generator is able to produce convincing fakes, and the discriminator no longer is able to tell which is which.

Note that "Discriminator wins" or "Generator wins" shouldn't be taken literally—there's no explicit tournament between the two. However, both networks are trained based on the outcome of the other network, which drives the optimization of the parameters of each network.

This technique has proven itself able to lead to generators that produce realistic images from nothing but noise and a conditioning signal, like an attribute (for example, for faces: young, female, glasses on) or another image. In other words, a well-trained generator learns a plausible model for generating images that look real even when examined by humans.

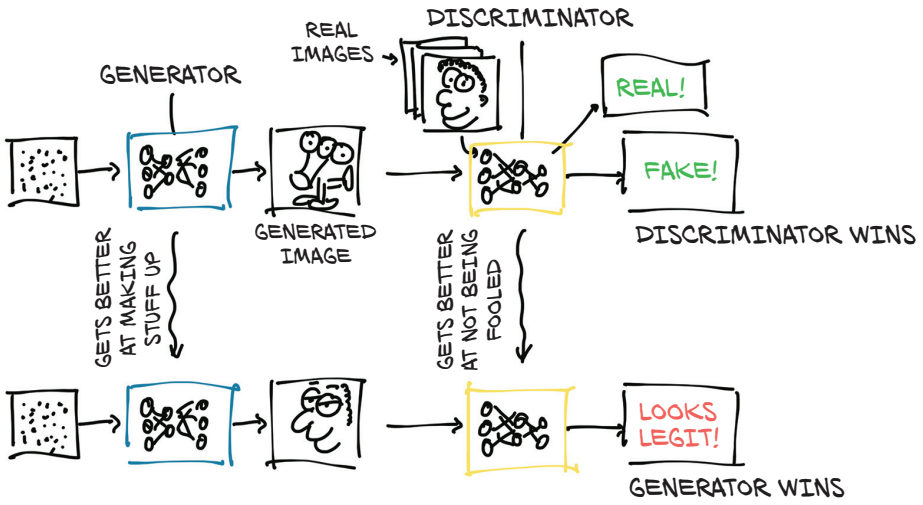

Figure 2.5 Concept of a GAN game

2.2.2 CycleGAN

An interesting evolution of this concept is the CycleGAN. A CycleGAN can turn images of one domain into images of another domain (and back), without the need for us to explicitly provide matching pairs in the training set.

In figure 2.6, we have a CycleGAN workflow for the task of turning a photo of a horse into a zebra, and vice versa. Note that there are two separate generator networks, as well as two distinct discriminators.

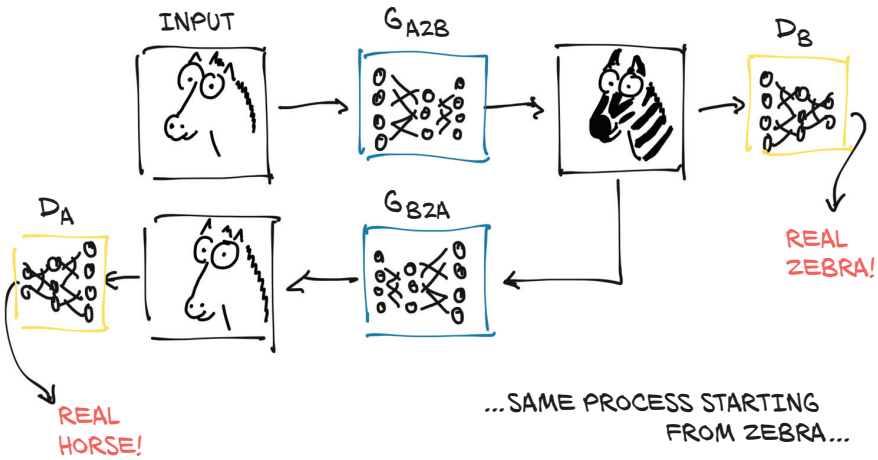

Figure 2.6 A CycleGAN trained to the point that it can fool both discriminator networks

As the figure shows, the first generator learns to produce an image conforming to a target distribution (zebras, in this case) starting from an image belonging to a different distribution (horses), so that the discriminator can't tell if the image produced from a horse photo is actually a genuine picture of a zebra or not. At the same time—and here's where the *Cycle* prefix in the acronym comes in—the resulting fake zebra is sent through a different generator going the other way (zebra to horse, in our case), to be judged by another discriminator on the other side. Creating such a cycle stabilizes the training process considerably, which addresses one of the original issues with GANs.

The fun part is that at this point, we don't need matched horse/zebra pairs as ground truths (good luck getting them to match poses!). It's enough to start from a collection of unrelated horse images and zebra photos for the generators to learn their task, going beyond a purely supervised setting. The implications of this model go even further than this: the generator learns how to selectively change the appearance of objects in the scene without supervision about what's what. There's no signal indicating that manes are manes and legs are legs, but they get translated to something that lines up with the anatomy of the other animal.

2.2.3 A network that turns horses into zebras

We can play with this model right now. The CycleGAN network has been trained on a dataset of (unrelated) horse images and zebra images extracted from the ImageNet dataset. The network learns to take an image of one or more horses and turn them all into zebras, leaving the rest of the image as unmodified as possible. While humankind hasn't held its breath over the last few thousand years for a tool that turn horses into zebras, this task showcases the ability of these architectures to model complex real-world processes with distant supervision. While they have their limits, there are hints that in the near future we won't be able to tell real from fake in a live video feed, which opens a can of worms that we'll duly close right now.

Playing with a pretrained CycleGAN will give us the opportunity to take a step closer and look at how a network—a generator, in this case—is implemented. We'll use our old friend ResNet. We'll define a `ResNetGenerator` class offscreen. The code is in the first cell of the 3_cyclegan.ipynb file, but the implementation isn't relevant right now, and it's too complex to follow until we've gotten a lot more PyTorch experience. Right now, we're focused on *what* it can do, rather than *how* it does it. Let's instantiate the class with default parameters (code/p1ch2/3_cyclegan.ipynb):

```
# In[2]:
netG = ResNetGenerator()
```

The `netG` model has been created, but it contains random weights. We mentioned earlier that we would run a generator model that had been pretrained on the horse2zebra dataset, whose training set contains two sets of 1068 and 1335 images of horses and zebras, respectively. The dataset be found at http://mng.bz/8pKP. The weights of the model have been saved in a .pth file, which is nothing but a `pickle` file of the model's

tensor parameters. We can load those into `ResNetGenerator` using the model's `load_state_dict` method:

```
# In[3]:
model_path = '../data/p1ch2/horse2zebra_0.4.0.pth'
model_data = torch.load(model_path)
netG.load_state_dict(model_data)
```

At this point, `netG` has acquired all the knowledge it achieved during training. Note that this is fully equivalent to what happened when we loaded `resnet101` from `torchvision` in section 2.1.3; but the `torchvision.resnet101` function hid the loading from us.

Let's put the network in `eval` mode, as we did for `resnet101`:

```
# In[4]:
netG.eval()

# Out[4]:
ResNetGenerator(
  (model): Sequential(
...
  )
)
```

Printing out the model as we did earlier, we can appreciate that it's actually pretty condensed, considering what it does. It takes an image, recognizes one or more horses in it by looking at pixels, and individually modifies the values of those pixels so that what comes out looks like a credible zebra. We won't recognize anything zebra-like in the printout (or in the source code, for that matter): that's because there's nothing zebra-like in there. The network is a scaffold—the juice is in the weights.

We're ready to load a random image of a horse and see what our generator produces. First, we need to import `PIL` and `torchvision`:

```
# In[5]:
from PIL import Image
from torchvision import transforms
```

Then we define a few input transformations to make sure data enters the network with the right shape and size:

```
# In[6]:
preprocess = transforms.Compose([transforms.Resize(256),
                                 transforms.ToTensor()])
```

Let's open a horse file (see figure 2.7):

```
# In[7]:
img = Image.open("../data/p1ch2/horse.jpg")
img
```

Figure 2.7 A man riding a horse. The horse is not having it.

OK, there's a dude on the horse. (Not for long, judging by the picture.) Anyhow, let's pass it through preprocessing and turn it into a properly shaped variable:

```
# In[8]:
img_t = preprocess(img)
batch_t = torch.unsqueeze(img_t, 0)
```

We shouldn't worry about the details right now. The important thing is that we follow from a distance. At this point, `batch_t` can be sent to our model:

```
# In[9]:
batch_out = netG(batch_t)
```

`batch_out` is now the output of the generator, which we can convert back to an image:

```
# In[10]:
out_t = (batch_out.data.squeeze() + 1.0) / 2.0
out_img = transforms.ToPILImage()(out_t)
# out_img.save('../data/p1ch2/zebra.jpg')
out_img

# Out[10]:
<PIL.Image.Image image mode=RGB size=316x256 at 0x23B24634F98>
```

Oh, man. Who rides a zebra that way? The resulting image (figure 2.8) is not perfect, but consider that it is a bit unusual for the network to find someone (sort of) riding on top of a horse. It bears repeating that the learning process has not passed through direct supervision, where humans have delineated tens of thousands of horses or manually Photoshopped thousands of zebra stripes. The generator has learned to produce an image that would fool the discriminator into thinking that was a zebra, and there was nothing fishy about the image (clearly the discriminator has never been to a rodeo).

Figure 2.8 A man riding a zebra. The zebra is not having it.

Many other fun generators have been developed using adversarial training or other approaches. Some of them are capable of creating credible human faces of nonexistent individuals; others can translate sketches into real-looking pictures of imaginary landscapes. Generative models are also being explored for producing real-sounding audio, credible text, and enjoyable music. It is likely that these models will be the basis of future tools that support the creative process.

On a serious note, it's hard to overstate the implications of this kind of work. Tools like the one we just downloaded are only going to become higher quality and more ubiquitous. Face-swapping technology, in particular, has gotten considerable media attention. Searching for "deep fakes" will turn up a plethora of example content[1] (though we must note that there is a nontrivial amount of not-safe-for-work content labeled as such; as with everything on the internet, click carefully).

So far, we've had a chance to play with a model that sees into images and a model that generates new images. We'll end our tour with a model that involves one more, fundamental ingredient: natural language.

2.3 A pretrained network that describes scenes

In order to get firsthand experience with a model involving natural language, we will use a pretrained image-captioning model, generously provided by Ruotian Luo.[2] It is an implementation of the NeuralTalk2 model by Andrej Karpathy. When presented with a natural image, this kind of model generates a caption in English that describes the scene, as shown in figure 2.9. The model is trained on a large dataset of images

[1] A relevant example is described in the Vox article "Jordan Peele's simulated Obama PSA is a double-edged warning against fake news," by Aja Romano; http://mng.bz/dxBz (warning: coarse language).

[2] We maintain a clone of the code at https://github.com/deep-learning-with-pytorch/ImageCaptioning.pytorch.

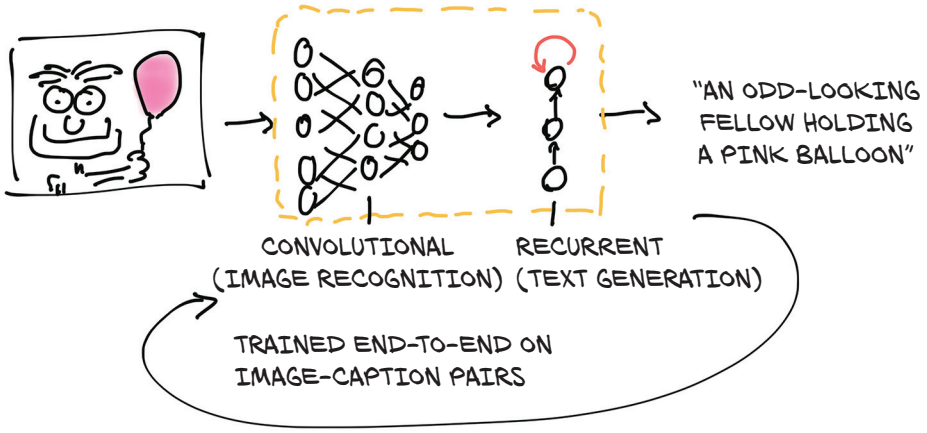

Figure 2.9 Concept of a captioning model

along with a paired sentence description: for example, "A Tabby cat is leaning on a wooden table, with one paw on a laser mouse and the other on a black laptop."[3]

This captioning model has two connected halves. The first half of the model is a network that learns to generate "descriptive" numerical representations of the scene (Tabby cat, laser mouse, paw), which are then taken as input to the second half. That second half is a *recurrent neural network* that generates a coherent sentence by putting those numerical descriptions together. The two halves of the model are trained together on image-caption pairs.

The second half of the model is called *recurrent* because it generates its outputs (individual words) in subsequent forward passes, where the input to each forward pass includes the outputs of the previous forward pass. This generates a dependency of the next word on words that were generated earlier, as we would expect when dealing with sentences or, in general, with sequences.

2.3.1 NeuralTalk2

The NeuralTalk2 model can be found at https://github.com/deep-learning-with-pytorch/ImageCaptioning.pytorch. We can place a set of images in the `data` directory and run the following script:

```
python eval.py --model ./data/FC/fc-model.pth
➥ --infos_path ./data/FC/fc-infos.pkl --image_folder ./data
```

Let's try it with our horse.jpg image. It says, "A person riding a horse on a beach." Quite appropriate.

[3] Andrej Karpathy and Li Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions," https://cs.stanford.edu/people/karpathy/cvpr2015.pdf.

Now, just for fun, let's see if our CycleGAN can also fool this NeuralTalk2 model. Let's add the zebra.jpg image in the data folder and rerun the model: "A group of zebras are standing in a field." Well, it got the animal right, but it saw more than one zebra in the image. Certainly this is not a pose that the network has ever seen a zebra in, nor has it ever seen a rider on a zebra (with some spurious zebra patterns). In addition, it is very likely that zebras are depicted in groups in the training dataset, so there might be some bias that we could investigate. The captioning network hasn't described the rider, either. Again, it's probably for the same reason: the network wasn't shown a rider on a zebra in the training dataset. In any case, this is an impressive feat: we generated a fake image with an impossible situation, and the captioning network was flexible enough to get the subject right.

We'd like to stress that something like this, which would have been extremely hard to achieve before the advent of deep learning, can be obtained with under a thousand lines of code, with a general-purpose architecture that knows nothing about horses or zebras, and a corpus of images and their descriptions (the MS COCO dataset, in this case). No hardcoded criterion or grammar—everything, including the sentence, emerges from patterns in the data.

The network architecture in this last case was, in a way, more complex than the ones we saw earlier, as it includes two networks. One is recurrent, but it was built out of the same building blocks, all of which are provided by PyTorch.

At the time of this writing, models such as these exist more as applied research or novelty projects, rather than something that has a well-defined, concrete use. The results, while promising, just aren't good enough to use … yet. With time (and additional training data), we should expect this class of models to be able to describe the world to people with vision impairment, transcribe scenes from video, and perform other similar tasks.

2.4 Torch Hub

Pretrained models have been published since the early days of deep learning, but until PyTorch 1.0, there was no way to ensure that users would have a uniform interface to get them. TorchVision was a good example of a clean interface, as we saw earlier in this chapter; but other authors, as we have seen for CycleGAN and NeuralTalk2, chose different designs.

PyTorch 1.0 saw the introduction of Torch Hub, which is a mechanism through which authors can publish a model on GitHub, with or without pretrained weights, and expose it through an interface that PyTorch understands. This makes loading a pretrained model from a third party as easy as loading a TorchVision model.

All it takes for an author to publish a model through the Torch Hub mechanism is to place a file named hubconf.py in the root directory of the GitHub repository. The file has a very simple structure:

```
# Optional list of modules the code depends on
dependencies = ['torch', 'math']

def some_entry_fn(*args, **kwargs):
    model = build_some_model(*args, **kwargs)
    return model

def another_entry_fn(*args, **kwargs):
    model = build_another_model(*args, **kwargs)
    return model
```

One or more functions to be exposed to users as entry points for the repository. These functions should initialize models according to the arguments and return them.

In our quest for interesting pretrained models, we can now search for GitHub repositories that include hubconf.py, and we'll know right away that we can load them using the torch.hub module. Let's see how this is done in practice. To do that, we'll go back to TorchVision, because it provides a clean example of how to interact with Torch Hub.

Let's visit https://github.com/pytorch/vision and notice that it contains a hubconf.py file. Great, that checks. The first thing to do is to look in that file to see the entry points for the repo—we'll need to specify them later. In the case of TorchVision, there are two: `resnet18` and `resnet50`. We already know what these do: they return an 18-layer and a 50-layer ResNet model, respectively. We also see that the entry-point functions include a `pretrained` keyword argument. If `True`, the returned models will be initialized with weights learned from ImageNet, as we saw earlier in the chapter.

Now we know the repo, the entry points, and one interesting keyword argument. That's about all we need to load the model using torch.hub, without even cloning the repo. That's right, PyTorch will handle that for us:

```
import torch
from torch import hub

resnet18_model = hub.load('pytorch/vision:master',
                          'resnet18',
                          pretrained=True)
```

Name and branch of the GitHub repo

Name of the entry-point function

Keyword argument

This manages to download a snapshot of the master branch of the pytorch/vision repo, along with the weights, to a local directory (defaults to .torch/hub in our home directory) and run the `resnet18` entry-point function, which returns the instantiated model. Depending on the environment, Python may complain that there's a module missing, like `PIL`. Torch Hub won't install missing dependencies, but it will report them to us so that we can take action.

At this point, we can invoke the returned model with proper arguments to run a forward pass on it, the same way we did earlier. The nice part is that now every model published through this mechanism will be accessible to us using the same modalities, well beyond vision.

Note that entry points are supposed to return models; but, strictly speaking, they are not forced to. For instance, we could have an entry point for transforming inputs and another one for turning the output probabilities into a text label. Or we could have an entry point for just the model, and another that includes the model along with the pre- and postprocessing steps. By leaving these options open, the PyTorch developers have provided the community with just enough standardization and a lot of flexibility. We'll see what patterns will emerge from this opportunity.

Torch Hub is quite new at the time of writing, and there are only a few models published this way. We can get at them by Googling "github.com hubconf.py." Hopefully the list will grow in the future, as more authors share their models through this channel.

2.5 Conclusion

We hope this was a fun chapter. We took some time to play with models created with PyTorch, which were optimized to carry out specific tasks. In fact, the more enterprising of us could already put one of these models behind a web server and start a business, sharing the profits with the original authors![4] Once we learn how these models are built, we will also be able to use the knowledge we gained here to download a pretrained model and quickly fine-tune it on a slightly different task.

We will also see how building models that deal with different problems on different kinds of data can be done using the same building blocks. One thing that PyTorch does particularly right is providing those building blocks in the form of an essential toolset—PyTorch is not a very large library from an API perspective, especially when compared with other deep learning frameworks.

This book does not focus on going through the complete PyTorch API or reviewing deep learning architectures; rather, we will build hands-on knowledge of these building blocks. This way, you will be able to consume the excellent online documentation and repositories on top of a solid foundation.

Starting with the next chapter, we'll embark on a journey that will enable us to teach our computer skills like those described in this chapter from scratch, using PyTorch. We'll also learn that starting from a pretrained network and fine-tuning it on new data, without starting from scratch, is an effective way to solve problems when the data points we have are not particularly numerous. This is one further reason pretrained networks are an important tool for deep learning practitioners to have. Time to learn about the first fundamental building block: tensors.

[4] Contact the publisher for franchise opportunities!

2.6 Exercises

1. Feed the image of the golden retriever into the horse-to-zebra model.
   a. What do you need to do to the image to prepare it?
   b. What does the output look like?
2. Search GitHub for projects that provide a hubconf.py file.
   a. How many repositories are returned?
   b. Find an interesting-looking project with a hubconf.py. Can you understand the purpose of the project from the documentation?
   c. Bookmark the project, and come back after you've finished this book. Can you understand the implementation?

2.7 Summary

- A pretrained network is a model that has already been trained on a dataset. Such networks can typically produce useful results immediately after loading the network parameters.
- By knowing how to use a pretrained model, we can integrate a neural network into a project without having to design or train it.
- AlexNet and ResNet are two deep convolutional networks that set new benchmarks for image recognition in the years they were released.
- Generative adversarial networks (GANs) have two parts—the generator and the discriminator—that work together to produce output indistinguishable from authentic items.
- CycleGAN uses an architecture that supports converting back and forth between two different classes of images.
- NeuralTalk2 uses a hybrid model architecture to consume an image and produce a text description of the image.
- Torch Hub is a standardized way to load models and weights from any project with an appropriate hubconf.py file.