Part 2

Learning from images in the real world: Early detection of lung cancer

Part 2 is structured differently than part 1; it's almost a book within a book. We'll take a single use case and explore it in depth over the course of several chapters, starting with the basic building blocks we learned in part 1, and building out a more complete project than we've seen so far. Our first attempts are going to be incomplete and inaccurate, and we'll explore how to diagnose those problems and then fix them. We'll also identify various other improvements to our solution, implement them, and measure their impact. In order to train the models we'll develop in part 2, you will need access to a GPU with at least 8 GB of RAM as well as several hundred gigabytes of free disk space to store the training data.

Chapter 9 introduces the project, environment, and data we will consume and the structure of the project we'll implement. Chapter 10 shows how we can turn our data into a PyTorch dataset, and chapters 11 and 12 introduce our classification model: the metrics we need to gauge how well the dataset is training, and implement solutions to problems preventing the model from training well. In chapter 13, we'll shift gears to the beginning of the end-to-end project by creating a segmentation model that produces a heatmap rather than a single classification. That heatmap will be used to generate locations to classify. Finally, in chapter 14, we'll combine our segmentation and classification models to perform a final diagnosis.

9 Using PyTorch to fight cancer

This chapter covers

- Breaking a large problem into smaller, easier ones
- Exploring the constraints of an intricate deep learning problem, and deciding on a structure and approach
- Downloading the training data

We have two main goals for this chapter. We'll start by covering the overall plan for part 2 of the book so that we have a solid idea of the larger scope the following individual chapters will be building toward. In chapter 10, we will begin to build out the data-parsing and data-manipulation routines that will produce data to be consumed in chapter 11 while training our first model. In order to do what's needed for those upcoming chapters well, we'll also use this chapter to cover some of the context in which our project will be operating: we'll go over data formats, data sources, and exploring the constraints that our problem domain places on us. Get used to performing these tasks, since you'll have to do them for any serious deep learning project!

9.1 Introduction to the use case

Our goal for this part of the book is to give you the tools to deal with situations where things aren't working, which is a far more common state of affairs than part 1 might have led you to believe. We can't predict every failure case or cover every debugging technique, but hopefully we'll give you enough to not feel stuck when you encounter a new roadblock. Similarly, we want to help you avoid situations with your own projects where you have no idea what you could do next when your projects are under-performing. Instead, we hope your ideas list will be so long that the challenge will be to prioritize!

In order to present these ideas and techniques, we need a context with some nuance and a fair bit of heft to it. We've chosen automatic detection of malignant tumors in the lungs using only a CT scan of a patient's chest as input. We'll be focusing on the technical challenges rather than the human impact, but make no mistake—even from just an engineering perspective, part 2 will require a more serious, structured approach than we needed in part 1 in order to have the project succeed.

NOTE CT scans are essentially 3D X-rays, represented as a 3D array of single-channel data. We'll cover them in more detail soon.

As you might have guessed, the title of this chapter is more eye-catching, implied hyperbole than anything approaching a serious statement of intent. Let us be precise: our project in this part of the book will take three-dimensional CT scans of human torsos as input and produce as output the location of suspected malignant tumors, if any exist.

Detecting lung cancer early has a huge impact on survival rate, but is difficult to do manually, especially in any comprehensive, whole-population sense. Currently, the work of reviewing the data must be performed by highly trained specialists, requires painstaking attention to detail, and it is dominated by cases where no cancer exists.

Doing that job well is akin to being placed in front of 100 haystacks and being told, "Determine which of these, if any, contain a needle." Searching this way results in the potential for missed warning signs, particularly in the early stages when the hints are more subtle. The human brain just isn't built well for that kind of monotonous work. And that, of course, is where deep learning comes in.

Automating this process is going to give us experience working in an uncooperative environment where we have to do more work from scratch, and there are fewer easy answers to problems that we might run into. Together, we'll get there, though! Once you're finished reading part 2, we think you'll be ready to start working on a real-world, unsolved problem of your own choosing.

We chose this problem of lung tumor detection for a few reasons. The primary reason is that the problem itself is unsolved! This is important, because we want to make it clear that you can use PyTorch to tackle cutting-edge projects effectively. We hope that increases your confidence in PyTorch as a framework, as well as in yourself as a developer. Another nice aspect of this problem space is that while it's unsolved, a lot of teams have been paying attention to it recently and have seen promising results. That means this challenge is probably right at the edge of our collective ability to solve; we won't be wasting our time on a problem that's actually decades away from

reasonable solutions. That attention on the problem has also resulted in a lot of high-quality papers and open source projects, which are a great source of inspiration and ideas. This will be a huge help once we conclude part 2 of the book, if you are interested in continuing to improve on the solution we create. We'll provide some links to additional information in chapter 14.

This part of the book will remain focused on the problem of detecting lung tumors, but the skills we'll teach are general. Learning how to investigate, preprocess, and present your data for training is important no matter what project you're working on. While we'll be covering preprocessing in the specific context of lung tumors, the general idea is that *this is what you should be prepared to do* for your project to succeed. Similarly, setting up a training loop, getting the right performance metrics, and tying the project's models together into a final application are all general skills that we'll employ as we go through chapters 9 through 14.

> **NOTE** While the end result of part 2 will work, the output will not be accurate enough to use clinically. We're focusing on using this as a motivating example for *teaching PyTorch*, not on employing every last trick to solve the problem.

9.2 Preparing for a large-scale project

This project will build off of the foundational skills learned in part 1. In particular, the content covering model construction from chapter 8 will be directly relevant. Repeated convolutional layers followed by a resolution-reducing downsampling layer will still make up the majority of our model. We will use 3D data as input to our model, however. This is conceptually similar to the 2D image data used in the last few chapters of part 1, but we will not be able to rely on all of the 2D-specific tools available in the PyTorch ecosystem.

The main differences between the work we did with convolutional models in chapter 8 and what we'll do in part 2 are related to how much effort we put into things outside the model itself. In chapter 8, we used a provided, off-the-shelf dataset and did little data manipulation before feeding the data into a model for classification. Almost all of our time and attention were spent building the model itself, whereas now we're not even going to begin designing the first of our two model architectures until chapter 11. That is a direct consequence of having nonstandard data without prebuilt libraries ready to hand us training samples suitable to plug into a model. We'll have to learn about our data and implement quite a bit ourselves.

Even when that's done, this will not end up being a case where we convert the CT to a tensor, feed it into a neural network, and have the answer pop out the other side. As is common for real-world use cases such as this, a workable approach will be more complicated to account for confounding factors such as limited data availability, finite computational resources, and limitations on our ability to design effective models. Please keep that in mind as we build to a high-level explanation of our project architecture.

Speaking of finite computational resources, part 2 will require access to a GPU to achieve reasonable training speeds, preferably one with at least 8 GB of RAM. Trying

to train the models we will build on CPU could take weeks![1] If you don't have a GPU handy, we provide pretrained models in chapter 14; the nodule analysis script there can probably be run overnight. While we don't want to tie the book to proprietary services if we don't have to, we should note that at the time of writing, Colaboratory (https://colab.research.google.com) provides free GPU instances that might be of use. PyTorch even comes preinstalled! You will also need to have at least 220 GB of free disk space to store the raw training data, cached data, and trained models.

> **NOTE** Many of the code examples presented in part 2 have complicating details omitted. Rather than clutter the examples with logging, error handling, and edge cases, the text of this book contains only code that expresses the core idea under discussion. Full working code samples can be found on the book's website (www.manning.com/books/deep-learning-with-pytorch) and GitHub (https://github.com/deep-learning-with-pytorch/dlwpt-code).

OK, we've established that this is a hard, multifaceted problem, but what are we going to do about it? Instead of looking at an entire CT scan for signs of tumors or their potential malignancy, we're going to solve a series of simpler problems that will combine to provide the end-to-end result we're interested in. Like a factory assembly line, each step will take raw materials (data) and/or output from previous steps, perform some processing, and hand off the result to the next station down the line. Not every problem needs to be solved this way, but breaking off chunks of the problem to solve in isolation is often a great way to start. Even if it turns out to be the wrong approach for a given project, it's likely we'll have learned enough while working on the individual chunks that we'll have a good idea how to restructure our approach into something successful.

Before we get into the details of how we'll break down our problem, we need to learn some details about the medical domain. While the code listings will tell you *what* we're doing, learning about radiation oncology will explain *why*. Learning about the problem space is crucial, no matter what domain it is. Deep learning is powerful, but it's not magic, and trying to apply it blindly to nontrivial problems will likely fail. Instead, we have to combine insights into the space with intuition about neural network behavior. From there, disciplined experimentation and refinement should give us enough information to close in on a workable solution.

9.3 What is a CT scan, exactly?

Before we get too far into the project, we need to take a moment to explain what a CT scan is. We will be using data from CT scans extensively as the main data format for our project, so having a working understanding of the data format's strengths, weaknesses, and fundamental nature will be crucial to utilizing it well. The key point we noted earlier is this: CT scans are essentially 3D X-rays, represented as a 3D array of

[1] We presume—we haven't tried it, much less timed it.

single-channel data. As we might recall from chapter 4, this is like a stacked set of grayscale PNG images.

Voxel

A *voxel* is the 3D equivalent to the familiar two-dimensional pixel. It encloses a volume of space (hence, "volumetric pixel"), rather than an area, and is typically arranged in a 3D grid to represent a field of data. Each of those dimensions will have a measurable distance associated with it. Often, voxels are cubic, but for this chapter, we will be dealing with voxels that are rectangular prisms.

In addition to medical data, we can see similar voxel data in fluid simulations, 3D scene reconstructions from 2D images, light detection and ranging (LIDAR) data for self-driving cars, and many other problem spaces. Those spaces all have their individual quirks and subtleties, and while the APIs that we're going to cover here apply generally, we must also be aware of the nature of the data we're using with those APIs if we want to be effective.

Each voxel of a CT scan has a numeric value that roughly corresponds to the average mass density of the matter contained inside. Most visualizations of that data show high-density material like bones and metal implants as white, low-density air and lung tissue as black, and fat and tissue as various shades of gray. Again, this ends up looking somewhat similar to an X-ray, with some key differences.

The primary difference between CT scans and X-rays is that whereas an X-ray is a projection of 3D intensity (in this case, tissue and bone density) onto a 2D plane, a CT scan retains the third dimension of the data. This allows us to render the data in a variety of ways: for example, as a grayscale solid, which we can see in figure 9.1.

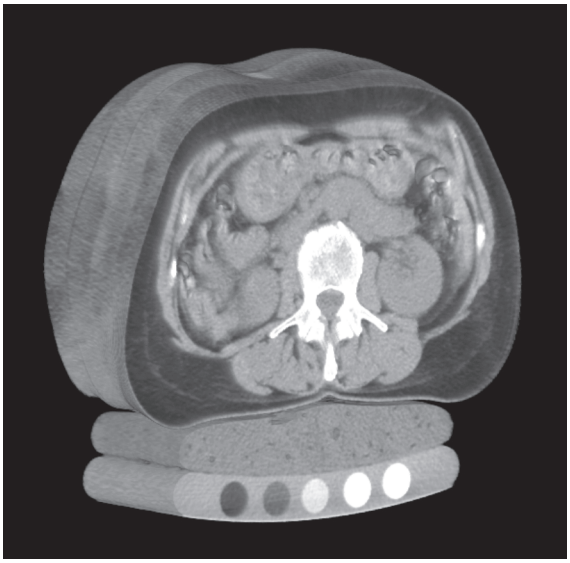

Figure 9.1 A CT scan of a human torso showing, from the top, skin, organs, spine, and patient support bed. Source: http://mng.bz/04r6; Mindways CT Software / CC BY-SA 3.0 (https://creativecommons.org/licenses/by-sa/3.0/deed.en).

NOTE CT scans actually measure radiodensity, which is a function of both mass density and atomic number of the material under examination. For our purposes here, the distinction isn't relevant, since the model will consume and learn from the CT data no matter what the exact units of the input happen to be.

This 3D representation also allows us to "see inside" the subject by hiding tissue types we are not interested in. For example, we can render the data in 3D and restrict visibility to only bone and lung tissue, as in figure 9.2.

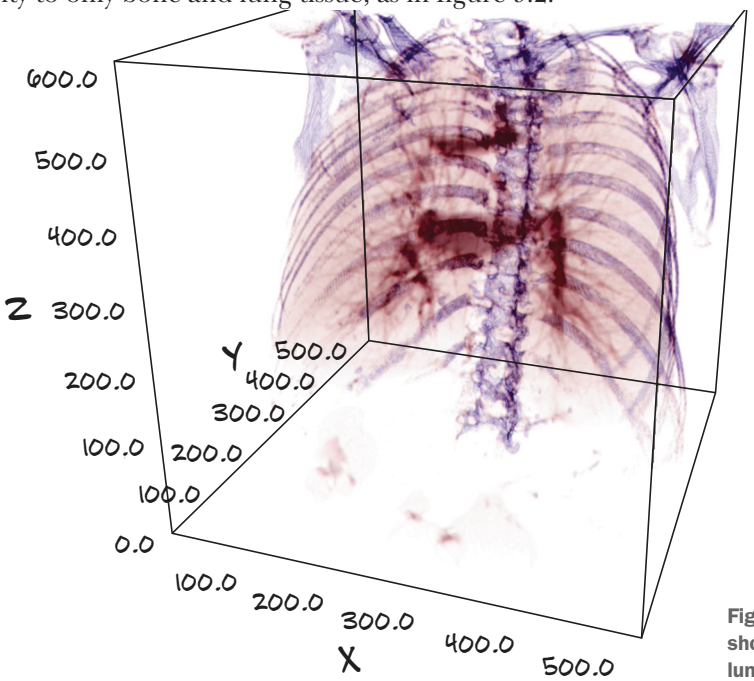

Figure 9.2 A CT scan showing ribs, spine, and lung structures

CT scans are much more difficult to acquire than X-rays, because doing so requires a machine like the one shown in figure 9.3 that typically costs upward of a million dollars new and requires trained staff to operate it. Most hospitals and some well-equipped clinics have a CT scanner, but they aren't nearly as ubiquitous as X-ray machines. This, combined with patient privacy regulations, can make it somewhat difficult to get CT scans unless someone has already done the work of gathering and organizing a collection of them.

Figure 9.3 also shows an example bounding box for the area contained in the CT scan. The bed the patient is resting on moves back and forth, allowing the scanner to image multiple slices of the patient and hence fill the bounding box. The scanner's darker, central ring is where the actual imaging equipment is located.

A final difference between a CT scan and an X-ray is that the data is a digital-only format. *CT* stands for *computed tomography* (https://en.wikipedia.org/wiki/CT_scan#Process).

Figure 9.3 A patient inside a CT scanner, with the CT scan's bounding box overlaid. Other than in stock photos, patients don't typically wear street clothes while in the machine.

The raw output of the scanning process doesn't look particularly meaningful to the human eye and must be properly reinterpreted by a computer into something we can understand. The settings of the CT scanner when the scan is taken can have a large impact on the resulting data.

While this information might not seem particularly relevant, we have actually learned something that is: from figure 9.3, we can see that the way the CT scanner measures distance along the head-to-foot axis is different than the other two axes. The patient actually moves along that axis! This explains (or at least is a strong hint as to) why our voxels might not be cubic, and also ties into how we approach massaging our data in chapter 12. This is a good example of why we need to understand our problem space if we're going to make effective choices about how to solve our problem. When starting to work on your own projects, be sure you do the same investigation into the details of your data.

9.4 The project: An end-to-end detector for lung cancer

Now that we've got our heads wrapped around the basics of CT scans, let's discuss the structure of our project. Most of the bytes on disk will be devoted to storing the CT scans' 3D arrays containing density information, and our models will primarily consume various subslices of those 3D arrays. We're going to use five main steps to go from examining a whole-chest CT scan to giving the patient a lung cancer diagnosis.

Our full, end-to-end solution shown in figure 9.4 will load CT data files to produce a `Ct` instance that contains the full 3D scan, combine that with a module that performs *segmentation* (flagging voxels of interest), and then group the interesting voxels into small lumps in the search for candidate *nodules*.

The nodule locations are combined back with the CT voxel data to produce nodule candidates, which can then be examined by our nodule classification model to determine whether they are actually nodules in the first place and, eventually, whether

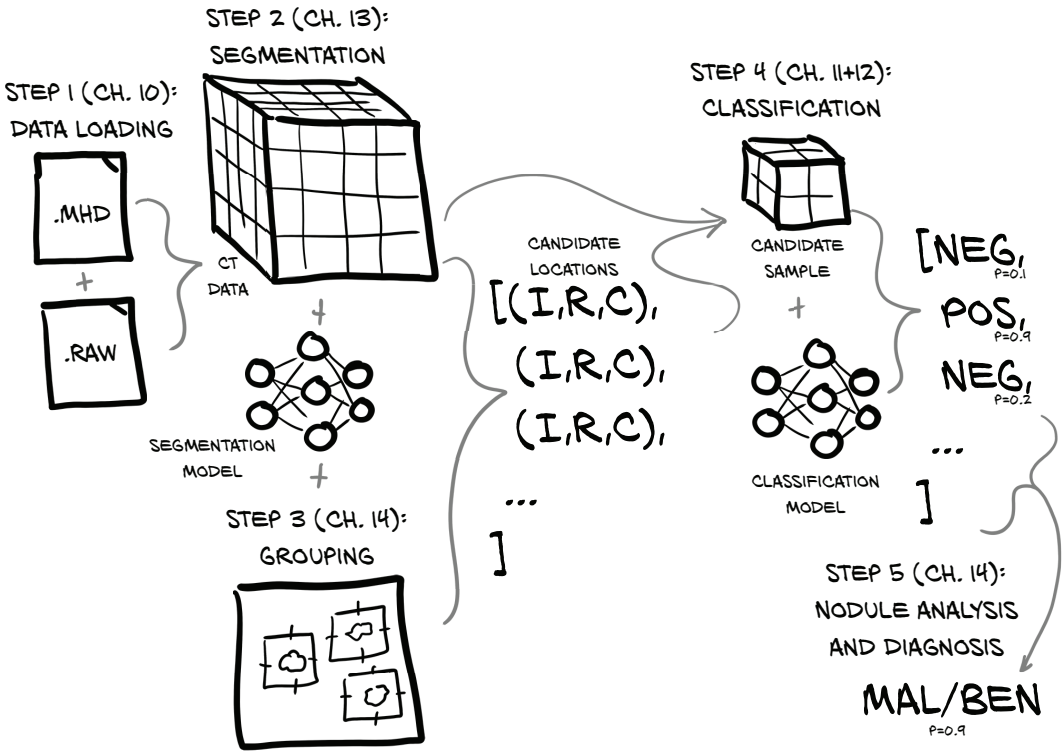

Figure 9.4 The end-to-end process of taking a full-chest CT scan and determining whether the patent has a malignant tumor

Nodules

A mass of tissue made of proliferating cells in the lung is a *tumor*. A tumor can be *benign* or it can be *malignant*, in which case it is also referred to as *cancer*. A small tumor in the lung (just a few millimeters wide) is called a *nodule*. About 40% of lung nodules turn out to be malignant—small cancers. It is very important to catch those as early as possible, and this depends on medical imaging of the kind we are looking at here.

they're malignant. This latter task is particularly difficult because malignancy might not be apparent from CT imaging alone, but we'll see how far we get. Last, each of those individual, per-nodule classifications can then be combined into a whole-patient diagnosis.

In more detail, we will do the following:

1. Load our raw CT scan data into a form that we can use with PyTorch. Putting raw data into a form usable by PyTorch will be the first step in any project you face. The process is somewhat less complicated with 2D image data and simpler still with non-image data.

2. Identify the voxels of potential tumors in the lungs using PyTorch to implement a technique known as *segmentation*. This is roughly akin to producing a heatmap of areas that should be fed into our classifier in step 3. This will allow us to focus on potential tumors inside the lungs and ignore huge swaths of uninteresting anatomy (a person can't have lung cancer in the stomach, for example).

   Generally, being able to focus on a single, small task is best while learning. With experience, there are some situations where more complicated model structures can yield superlative results (for example, the GAN game we saw in chapter 2), but designing those from scratch requires extensive mastery of the basic building blocks first. Gotta walk before you run, and all that.

3. Group interesting voxels into lumps: that is, candidate nodules (see figure 9.5 for more information on nodules). Here, we will find the rough center of each hotspot on our heatmap.

   Each nodule can be located by the index, row, and column of its center point. We do this to present a simple, constrained problem to the final classifier. Grouping voxels will not involve PyTorch directly, which is why we've pulled this out into a separate step. Often, when working with multistep solutions, there will be non-deep-learning glue steps between the larger, deep-learning-powered portions of the project.

4. Classify candidate nodules as actual nodules or non-nodules using 3D convolution.

   This will be similar in concept to the 2D convolution we covered in chapter 8. The features that determine the nature of a tumor from a candidate structure are local to the tumor in question, so this approach should provide a good balance between limiting input data size and excluding relevant information. Making scope-limiting decisions like this can keep each individual task constrained, which can help limit the amount of things to examine when troubleshooting.

5. Diagnose the patient using the combined per-nodule classifications.

   Similar to the nodule classifier in the previous step, we will attempt to determine whether the nodule is benign or malignant based on imaging data alone. We will take a simple maximum of the per-tumor malignancy predictions, as only one tumor needs to be malignant for a patient to have cancer. Other projects might want to use different ways of aggregating the per-instance predictions into a file score. Here, we are asking, "Is there anything suspicious?" so maximum is a good fit for aggregation. If we were looking for quantitative information like "the ratio of type A tissue to type B tissue," we might take an appropriate mean instead.

Figure 9.4 only depicts the final path through the system once we've built and trained all of the requisite models. The actual work required to train the relevant models will be detailed as we get closer to implementing each step.

The data we'll use for training provides human-annotated output for both steps 3 and 4. This allows us to treat steps 2 and 3 (identifying voxels and grouping them into nodule candidates) as almost a separate project from step 4 (nodule candidate

On the shoulders of giants

We are *standing on the shoulders of giants* when deciding on this five-step approach. We'll discuss these giants and their work more in chapter 14. There isn't any particular reason why we should know in advance that this project structure will work well for this problem; instead, we're relying on others who have actually implemented similar things and reported success when doing so. Expect to have to experiment to find workable approaches when transitioning to a different domain, but always try to learn from earlier efforts in the space and from those who have worked in similar areas and have discovered things that might transfer well. Go out there, look for what others have done, and use that as a benchmark. At the same time, avoid getting code and running it *blindly*, because you need to fully understand the code you're running in order to use the results to make progress for yourself.

classification). Human experts have annotated the data with nodule locations, so we can work on either steps 2 and 3 or step 4 in whatever order we prefer.

We will first work on step 1 (data loading), and then jump to step 4 before we come back and implement steps 2 and 3, since step 4 (classification) requires an approach similar to what we used in chapter 8, using multiple convolutional and pooling layers to aggregate spatial information before feeding it into a linear classifier. Once we've got a handle on our classification model, we can start working on step 2 (segmentation). Since segmentation is the more complicated topic, we want to tackle it without having to learn both segmentation and the fundamentals of CT scans and malignant tumors at the same time. Instead, we'll explore the cancer-detection space while working on a more familiar classification problem.

This approach of starting in the middle of the problem and working our way out probably seems odd. Starting at step 1 and working our way forward would make more intuitive sense. Being able to carve up the problem and work on steps independently is useful, however, since it can encourage more modular solutions; in addition, it's easier to partition the workload between members of a small team. Also, actual clinical users would likely prefer a system that flags suspicious nodules for review rather than provides a single binary diagnosis. Adapting our modular solution to different use cases will probably be easier than if we'd done a monolithic, from-the-top system.

As we work our way through implementing each step, we'll be going into a fair bit of detail about lung tumors, as well as presenting a lot of fine-grained detail about CT scans. While that might seem off-topic for a book that's focused on PyTorch, we're doing so specifically so that you begin to develop an intuition about the problem space. That's crucial to have, because the space of all possible solutions and approaches is too large to effectively code, train, and evaluate.

If we were working on a different project (say, the one you tackle after finishing this book), we'd still need to do an investigation to understand the data and problem space. Perhaps you're interested in satellite mapping, and your next project needs to consume pictures of our planet taken from orbit. You'd need to ask questions about the wavelengths being collected—do you get only normal RGB, or something more

exotic? What about infrared or ultraviolet? In addition, there might be impacts on the images based on time of day, or if the imaged location isn't directly under the satellite, skewing the image. Will the image need correction?

Even if your hypothetical *third* project's data type remains the same, it's probable that the domain you'll be working in will change things, possibly drastically. Processing camera output for self-driving cars still involves 2D images, but the complications and caveats are wildly different. For example, it's much less likely that a mapping satellite will need to worry about the sun shining into the camera, or getting mud on the lens!

We must be able to use our intuition to guide our investigation into potential optimizations and improvements. That's true of deep learning projects in general, and we'll practice using our intuition as we go through part 2. So, let's do that. Take a quick step back, and do a gut check. What does your intuition say about this approach? Does it seem overcomplicated to you?

9.4.1 Why can't we just throw data at a neural network until it works?

After reading the last section, we couldn't blame you for thinking, "This is nothing like chapter 8!" You might be wondering why we've got two separate model architectures or why the overall data flow is so complicated. Well, our approach is different from that in chapter 8 for a reason. It's a hard task to automate, and people haven't fully figured it out yet. That difficulty translates to complexity; once we as a society have solved this problem definitively, there will probably be an off-the-shelf library package we can grab to have it Just Work, but we're not there just yet.

Why so difficult, though?

Well, for starters, the majority of a CT scan is fundamentally uninteresting with regard to answering the question, "Does this patient have a malignant tumor?" This makes intuitive sense, since the vast majority of the patient's body will consist of healthy cells. In the cases where there is a malignant tumor, up to 99.9999% of the voxels in the CT still won't be cancer. That ratio is equivalent to a two-pixel blob of incorrectly tinted color somewhere on a high-definition television, or a single misspelled word out of a shelf of novels.

Can you identify the white dot in the three views of figure 9.5 that has been flagged as a nodule?[2]

If you need a hint, the index, row, and column values can be used to help find the relevant blob of dense tissue. Do you think you could figure out the relevant properties of tumors given only images (and that means *only* the images—no index, row, and column information!) like these? What if you were given the entire 3D scan, not just three slices that intersect the interesting part of the scan?

> **NOTE** Don't fret if you can't locate the tumor! We're trying to illustrate just how subtle this data can be—the fact that it is hard to identify visually is the entire point of this example.

[2]The `series_uid` of this sample is `1.3.6.1.4.1.14519.5.2.1.6279.6001.126264578931778258890371755354`, which can be useful if you'd like to look at it in detail later.

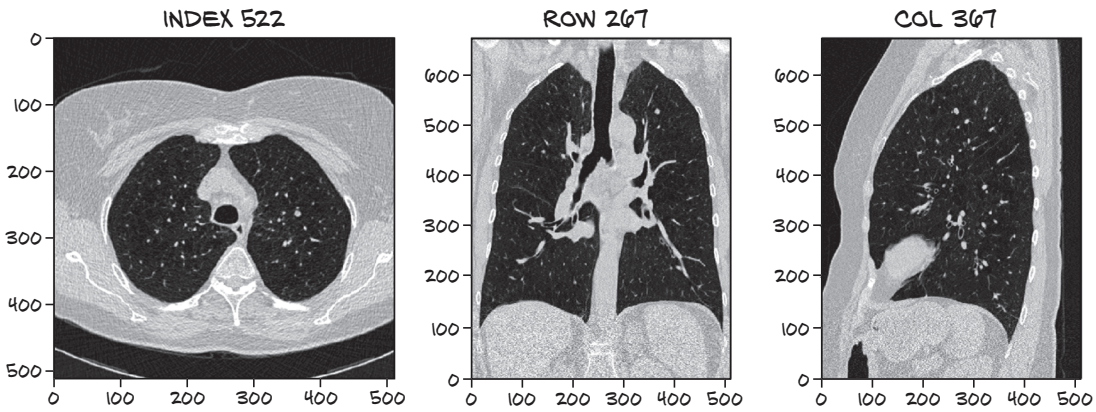

Figure 9.5 A CT scan with approximately 1,000 structures that look like tumors to the untrained eye. Exactly one has been identified as a nodule when reviewed by a human specialist. The rest are normal anatomical structures like blood vessels, lesions, and other non-problematic lumps.

You might have seen elsewhere that end-to-end approaches for detection and classification of objects are very successful in general vision tasks. TorchVision includes end-to-end models like Fast R-CNN/Mask R-CNN, but these are typically trained on hundreds of thousands of images, and those datasets aren't constrained by the number of samples from rare classes. The project architecture we will use has the benefit of working well with a more modest amount of data. So while it's certainly theoretically possible to just throw an arbitrarily large amount of data at a neural network until it learns the specifics of the proverbial lost needle, as well as how to ignore the hay, it's going to be practically prohibitive to collect enough data and wait for a long enough time to train the network properly. That won't be the *best* approach since the results are poor, and most readers won't have access to the compute resources to pull it off at all.

To come up with the best solution, we could investigate proven model designs that can better integrate data in an end-to-end manner.[3] These complicated designs are capable of producing high-quality results, but they're not the *best* because understanding the design decisions behind them requires having mastered fundamental concepts first. That makes these advanced models poor candidates to use while teaching those same fundamentals!

That's not to say that our multistep design is the *best* approach, either, but that's because "best" is only relative to the criteria we chose to evaluate approaches. There are *many* "best" approaches, just as there are many goals we could have in mind as we work on a project. Our self-contained, multistep approach has some disadvantages as well.

Recall the GAN game from chapter 2. There, we had two networks cooperating to produce convincing forgeries of old master artists. The artist would produce a candidate work, and the scholar would critique it, giving the artist feedback on how to

[3]For example, Retina U-Net (https://arxiv.org/pdf/1811.08661.pdf) and FishNet (http://mng.bz/K240).

improve. Put in technical terms, the structure of the model allowed gradients to backpropagate from the final classifier (fake or real) to the earliest parts of the project (the artist).

Our approach for solving the problem won't use end-to-end gradient backpropagation to directly optimize for our end goal. Instead, we'll optimize discrete chunks of the problem individually, since our segmentation model and classification model won't be trained in tandem with each other. That might limit the top-end effectiveness of our solution, but we feel that this will make for a much better learning experience.

We feel that being able to focus on a single step at a time allows us to zoom in and concentrate on the smaller number of new skills we're learning. Each of our two models will be focused on performing exactly one task. Similar to a human radiologist as they review slice after slice of CT, the job gets much easier to train for if the scope is well contained. We also want to provide tools that allow for rich manipulation of the data. Being able to zoom in and focus on the detail of a particular location will have a huge impact on overall productivity while training the model compared to having to look at the entire image at once. Our segmentation model is forced to consume the entire image, but we will structure things so that our classification model gets a zoomed-in view of the areas of interest.

Step 3 (grouping) will produce and step 4 (classification) will consume data similar to the image in figure 9.6 containing sequential transverse slices of a tumor. This image is a close-up view of a (potentially malignant, or at least indeterminate) tumor, and it is what we're going to train the step 4 model to identify, and the step 5 model to classify as either benign or malignant. While this lump may seem nondescript to an untrained eye (or untrained convolutional network), identifying the warning signs of malignancy in this sample is at least a far more constrained problem than having to consume the entire CT we saw earlier. Our code for the next chapter will provide routines to produce zoomed-in nodule images like figure 9.6.

We will perform the step 1 data-loading work in chapter 10, and chapters 11 and 12 will focus on solving the problem of classifying these nodules. After that, we'll back up to work on step 2 (using segmentation to find the candidate tumors) in chapter 13, and then we'll close out part 2 of the book in chapter 14 by implementing the end-to-end project with step 3 (grouping) and step 5 (nodule analysis and diagnosis).

NOTE Standard rendering of CTs places the superior at the top of the image (basically, the head goes up), but CTs order their slices such that the first slice is the inferior (toward the feet). So, Matplotlib renders the images upside down unless we take care to flip them. Since that flip doesn't really matter to our model, we won't complicate the code paths between our raw data and the model, but we will add a flip to our rendering code to get the images right-side up. For more information about CT coordinate systems, see section 10.4.

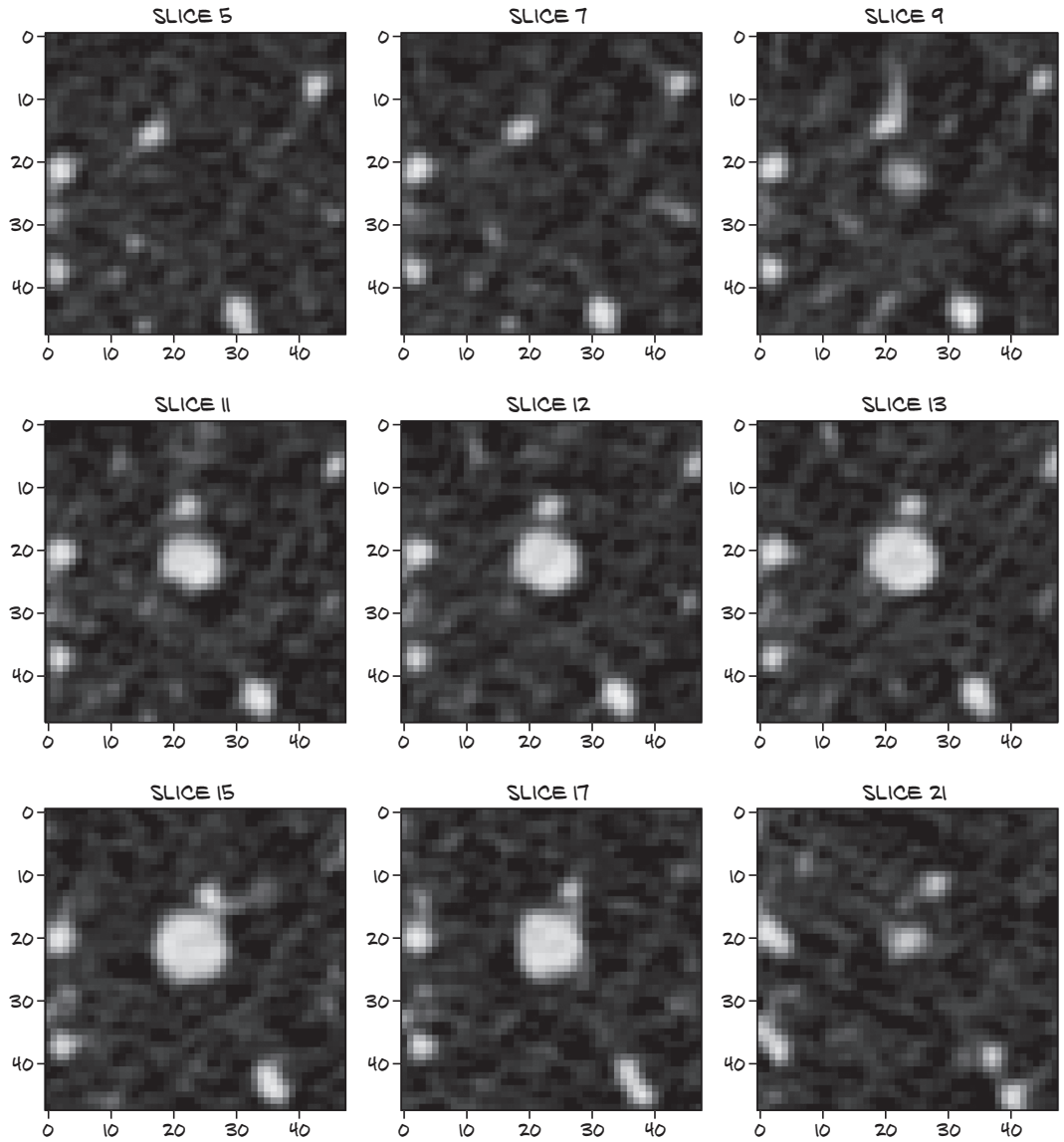

Figure 9.6 A close-up, multislice crop of the tumor from the CT scan in figure 9.5

Let's repeat our high-level overview in figure 9.7.

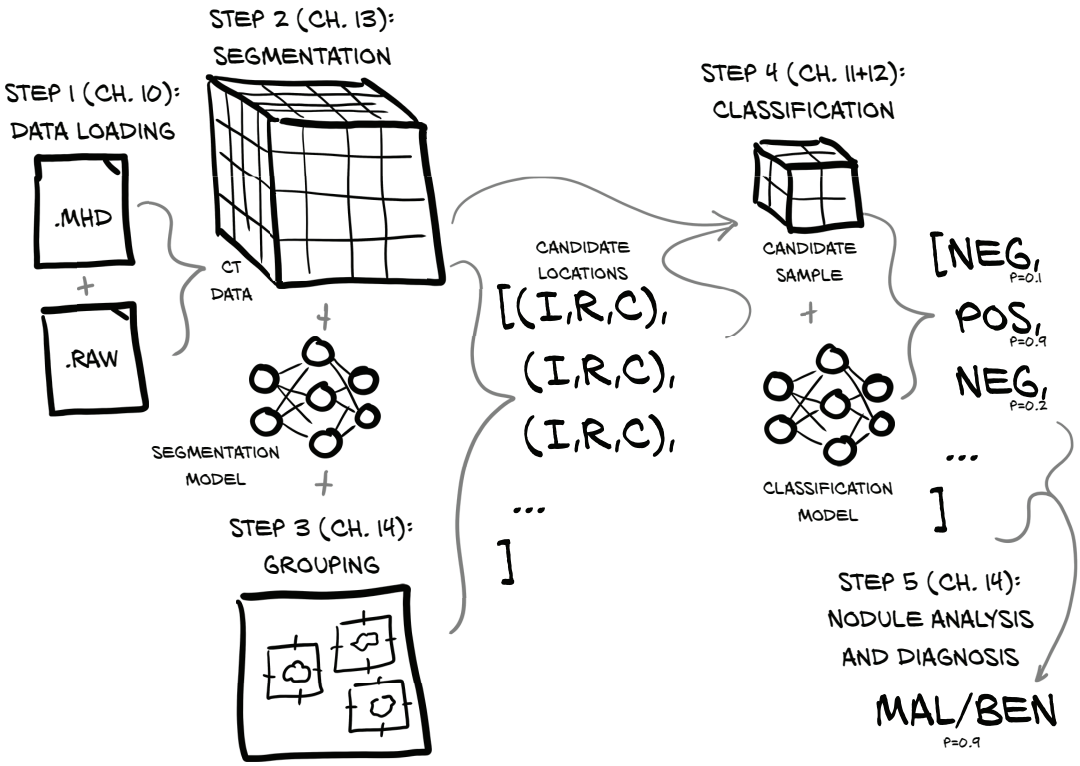

Figure 9.7 The end-to-end process of taking a full-chest CT scan and determining whether the patient has a malignant tumor

9.4.2 What is a nodule?

As we've said, in order to understand our data well enough to use it effectively, we need to learn some specifics about cancer and radiation oncology. One last key thing we need to understand is what a *nodule* is. Simply put, a nodule is any of the myriad lumps and bumps that might appear inside someone's lungs. Some are problematic from a health-of-the-patient perspective; some are not. The precise definition[4] limits the size of a nodule to 3 cm or less, with a larger lump being a *lung mass*; but we're going to use *nodule* interchangeably for all such anatomical structures, since it's a somewhat arbitrary cutoff and we're going to deal with lumps on both sides of 3 cm using the same code paths. A nodule—a small mass in the lung—can turn out to be benign or a malignant tumor (also referred to as *cancer*). From a radiological perspective, a nodule is really similar to other lumps that have a wide variety of causes: infection, inflammation, blood-supply issues, malformed blood vessels, and diseases other than tumors.

[4] Eric J. Olson, "Lung nodules: Can they be cancerous?" Mayo Clinic, http://mng.bz/yyge.

The key part is this: the cancers that we are trying to detect will *always* be nodules, either suspended in the very non-dense tissue of the lung or attached to the lung wall. That means we can limit our classifier to only nodules, rather than have it examine all tissue. Being able to restrict the scope of expected inputs will help our classifier learn the task at hand.

This is another example of how the underlying deep learning techniques we'll use are universal, but they can't be applied blindly.[5] We'll need to understand the field we're working in to make choices that will serve us well.

In figure 9.8, we can see a stereotypical example of a malignant nodule. The smallest nodules we'll be concerned with are only a few millimeters across, though the one in figure 9.8 is larger. As we discussed earlier in the chapter, this makes the smallest nodules approximately a million times smaller than the CT scan as a whole. More than half of the nodules detected in patients are not malignant.[6]

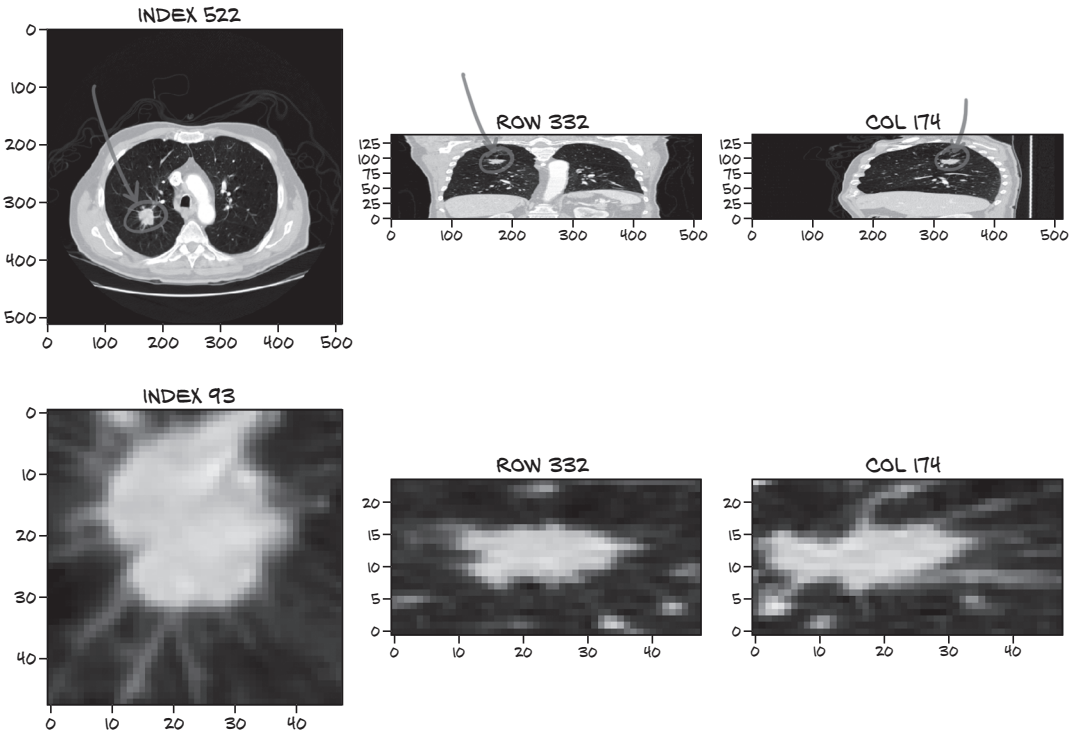

Figure 9.8 A CT scan with a malignant nodule displaying a visual discrepancy from other nodules

[5]Not if we want decent results, at least.

[6]According to the National Cancer Institute Dictionary of Cancer Terms: http://mng.bz/jgBP.

9.4.3 Our data source: The LUNA Grand Challenge

The CT scans we were just looking at come from the LUNA (LUng Nodule Analysis) Grand Challenge. The LUNA Grand Challenge is the combination of an open dataset with high-quality labels of patient CT scans (many with lung nodules) and a public ranking of classifiers against the data. There is something of a culture of publicly sharing medical datasets for research and analysis; open access to such data allows researchers to use, combine, and perform novel work on this data without having to enter into formal research agreements between institutions (obviously, some data is kept private as well). The goal of the LUNA Grand Challenge is to encourage improvements in nodule detection by making it easy for teams to compete for high positions on the leader board. A project team can test the efficacy of their detection methods against standardized criteria (the dataset provided). To be included in the public ranking, a team must provide a scientific paper describing the project architecture, training methods, and so on. This makes for a great resource to provide further ideas and inspiration for project improvements.

> **NOTE** Many CT scans "in the wild" are incredibly messy, in terms of idiosyncrasies between various scanners and processing programs. For example, some scanners indicate areas of the CT scan that are outside of the scanner's field of view by setting the density of those voxels to something negative. CT scans can also be acquired with a variety of settings on the CT scanner, which can change the resulting image in ways ranging from subtly to wildly different. Although the LUNA data is generally clean, be sure to check your assumptions if you incorporate other data sources.

We will be using the LUNA 2016 dataset. The LUNA site (https://luna16.grand-challenge.org/Description) describes two tracks for the challenge: the first track, "Nodule detection (NDET)," roughly corresponds to our step 1 (segmentation); and the second track, "False positive reduction (FPRED)," is similar to our step 3 (classification). When the site discusses "locations of possible nodules," it is talking about a process similar to what we'll cover in chapter 13.

9.4.4 Downloading the LUNA data

Before we go any further into the nuts and bolts of our project, we'll cover how to get the data we'll be using. It's about 60 GB of data compressed, so depending on your internet connection, it might take a while to download. Once uncompressed, it takes up about 120 GB of space; and we'll need another 100 GB or so of cache space to store smaller chunks of data so that we can access it more quickly than reading in the whole CT.[7]

[7]The cache space required is per chapter, but once you're done with a chapter, you can delete the cache to free up space.

Navigate to https://luna16.grand-challenge.org/download and either register using email or use the Google OAuth login. Once logged in, you should see two download links to Zenodo data, as well as a link to Academic Torrents. The data should be the same from either.

> **TIP** The luna.grand-challenge.org domain does not have links to the data download page as of this writing. If you are having issues finding the download page, double-check the domain for luna16., not `luna.`, and reenter the URL if needed.

The data we will be using comes in 10 subsets, aptly named `subset0` through `subset9`. Unzip each of them so you have separate subdirectories like code/data-unversioned/part2/luna/subset0, and so on. On Linux, you'll need the `7z` decompression utility (Ubuntu provides this via the `p7zip-full` package). Windows users can get an extractor from the 7-Zip website (www.7-zip.org). Some decompression utilities will not be able to open the archives; make sure you have the full version of the extractor if you get an error.

In addition, you need the candidates.csv and annotations.csv files. We've included these files on the book's website and in the GitHub repository for convenience, so they should already be present in code/data/part2/luna/*.csv. They can also be downloaded from the same location as the data subsets.

> **NOTE** If you do not have easy access to ~220 GB of free disk space, it's possible to run the examples using only 1 or 2 of the 10 subsets of data. The smaller training set will result in the model performing much more poorly, but that's better than not being able to run the examples at all.

Once you have the candidates file and at least one subset downloaded, uncompressed, and put in the correct location, you should be able to start running the examples in this chapter. If you want to jump ahead, you can use the code/p2ch09_explore_data.ipynb Jupyter Notebook to get started. Otherwise, we'll return to the notebook in more depth later in the chapter. Hopefully your downloads will finish before you start reading the next chapter!

9.5 Conclusion

We've made major strides toward finishing our project! You might have the feeling that we haven't accomplished much; after all, we haven't implemented a single line of code yet. But keep in mind that you'll need to do research and preparation as we have here when you tackle projects on your own.

In this chapter, we set out to do two things:

1. Understand the larger context around our lung cancer-detection project
2. Sketch out the direction and structure of our project for part 2

If you still feel that we haven't made real progress, please recognize that mindset as a trap—understanding the space your project is working in is crucial, and the design

work we've done will pay off handsomely as we move forward. We'll see those dividends shortly, once we start implementing our data-loading routines in chapter 10.

Since this chapter has been informational only, without any code, we'll skip the exercises for now.

9.6 Summary

- Our approach to detecting cancerous nodules will have five rough steps: data loading, segmentation, grouping, classification, and nodule analysis and diagnosis.
- Breaking down our project into smaller, semi-independent subprojects makes teaching each subproject easier. Other approaches might make more sense for future projects with different goals than the ones for this book.
- A CT scan is a 3D array of intensity data with approximately 32 million voxels, which is around a million times larger than the nodules we want to recognize. Focusing the model on a crop of the CT scan relevant to the task at hand will make it easier to get reasonable results from training.
- Understanding our data will make it easier to write processing routines for our data that don't distort or destroy important aspects of the data. The array of CT scan data typically will not have cubic voxels; mapping location information in real-world units to array indexes requires conversion. The intensity of a CT scan corresponds roughly to mass density but uses unique units.
- Identifying the key concepts of a project and making sure they are well represented in our design can be crucial. Most aspects of our project will revolve around nodules, which are small masses in the lungs and can be spotted on a CT along with many other structures that have a similar appearance.
- We are using the LUNA Grand Challenge data to train our model. The LUNA data contains CT scans, as well as human-annotated outputs for classification and grouping. Having high-quality data has a major impact on a project's success.

10 Combining data sources into a unified dataset

This chapter covers

- Loading and processing raw data files
- Implementing a Python class to represent our data
- Converting our data into a format usable by PyTorch
- Visualizing the training and validation data

Now that we've discussed the high-level goals for part 2, as well as outlined how the data will flow through our system, let's get into specifics of what we're going to do in this chapter. It's time to implement basic data-loading and data-processing routines for our raw data. Basically, every significant project you work on will need something analogous to what we cover here.[1] Figure 10.1 shows the high-level map of our project from chapter 9. We'll focus on step 1, data loading, for the rest of this chapter.

Our goal is to be able to produce a training sample given our inputs of raw CT scan data and a list of annotations for those CTs. This might sound simple, but quite a bit needs to happen before we can load, process, and extract the data we're

[1] To the rare researcher who has all of their data well prepared for them in advance: lucky you! The rest of us will be busy writing code for loading and parsing.

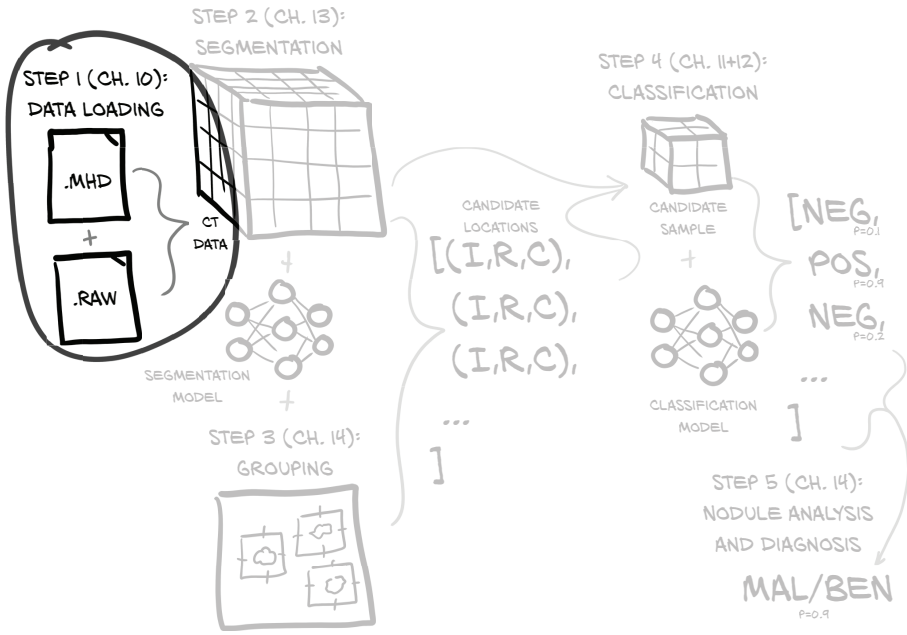

Figure 10.1 Our end-to-end lung cancer detection project, with a focus on this chapter's topic: step 1, data loading

interested in. Figure 10.2 shows what we'll need to do to turn our raw data into a training sample. Luckily, we got a head start on *understanding* our data in the last chapter, but we have more work to do on that front as well.

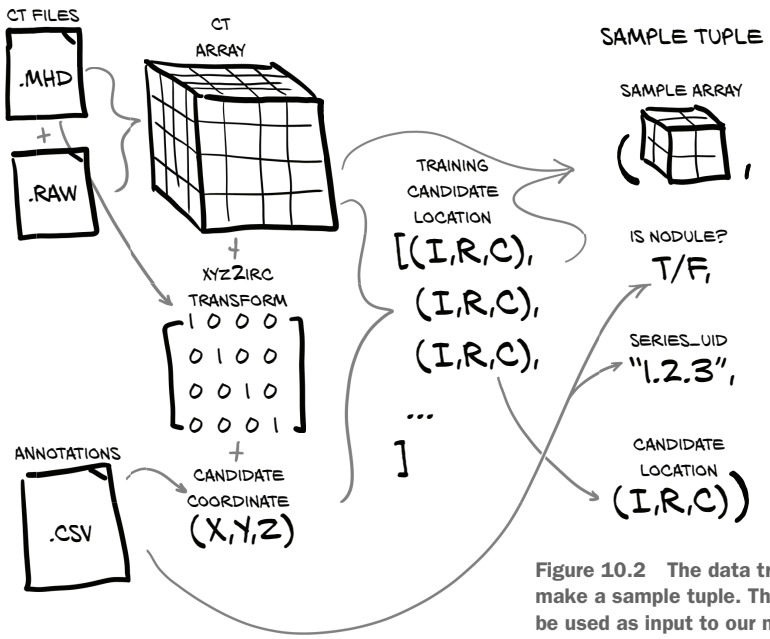

Figure 10.2 The data transforms required to make a sample tuple. These sample tuples will be used as input to our model training routine.

This is a crucial moment, when we begin to transmute the leaden raw data, if not into gold, then at least into the stuff that our neural network will spin *into* gold. We first discussed the mechanics of this transformation in chapter 4.

10.1 Raw CT data files

Our CT data comes in two files: a .mhd file containing metadata header information, and a .raw file containing the raw bytes that make up the 3D array. Each file's name starts with a unique identifier called the *series UID* (the name comes from the Digital Imaging and Communications in Medicine [DICOM] nomenclature) for the CT scan in question. For example, for series UID 1.2.3, there would be two files: 1.2.3.mhd and 1.2.3.raw.

Our `Ct` class will consume those two files and produce the 3D array, as well as the transformation matrix to convert from the patient coordinate system (which we will discuss in more detail in section 10.6) to the index, row, column coordinates needed by the array (these coordinates are shown as (I,R,C) in the figures and are denoted with `_irc` variable suffixes in the code). Don't sweat the details of all this right now; just remember that we've got some coordinate system conversion to do before we can apply these coordinates to our CT data. We'll explore the details as we need them.

We will also load the annotation data provided by LUNA, which will give us a list of nodule coordinates, each with a malignancy flag, along with the series UID of the relevant CT scan. By combining the nodule coordinate with coordinate system transformation information, we get the index, row, and column of the voxel at the center of our nodule.

Using the (I,R,C) coordinates, we can crop a small 3D slice of our CT data to use as the input to our model. Along with this 3D sample array, we must construct the rest of our training sample tuple, which will have the sample array, nodule status flag, series UID, and the index of this sample in the CT list of nodule candidates. This sample tuple is exactly what PyTorch expects from our `Dataset` subclass and represents the last section of our bridge from our original raw data to the standard structure of PyTorch tensors.

Limiting or cropping our data so as not to drown our model in noise is important, as is making sure we're not so aggressive that our signal gets cropped out of our input. We want to make sure the range of our data is well behaved, especially after normalization. Clamping our data to remove outliers can be useful, especially if our data is prone to extreme outliers. We can also create handcrafted, algorithmic transformations of our input; this is known as *feature engineering;* and we discussed it briefly in chapter 1. We'll usually want to let the model do most of the heavy lifting; feature engineering has its uses, but we won't use it here in part 2.

10.2 Parsing LUNA's annotation data

The first thing we need to do is begin loading our data. When working on a new project, that's often a good place to start. Making sure we know how to work with the raw input is required no matter what, and knowing how our data will look after it loads

can help inform the structure of our early experiments. We could try loading individual CT scans, but we think it makes sense to parse the CSV files that LUNA provides, which contain information about the points of interest in each CT scan. As we can see in figure 10.3, we expect to get some coordinate information, an indication of whether the coordinate is a nodule, and a unique identifier for the CT scan. Since there are fewer types of information in the CSV files, and they're easier to parse, we're hoping they will give us some clues about what to look for once we start loading CTs.

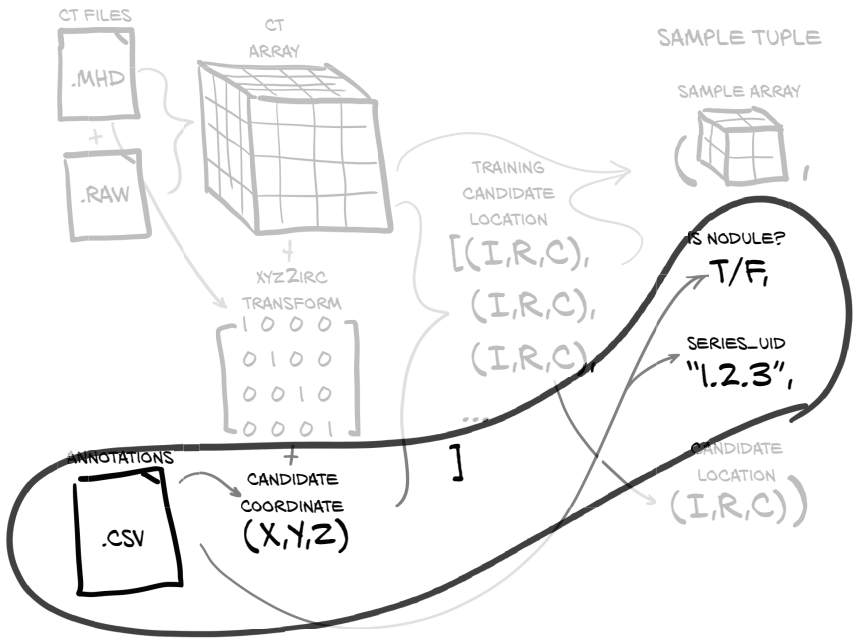

Figure 10.3 The LUNA annotations in candidates.csv contain the CT series, the nodule candidate's position, and a flag indicating if the candidate is actually a nodule or not.

The candidates.csv file contains information about all lumps that potentially look like nodules, whether those lumps are malignant, benign tumors, or something else altogether. We'll use this as the basis for building a complete list of candidates that can then be split into our training and validation datasets. The following Bash shell session shows what the file contains:

```
$ wc -l candidates.csv
551066 candidates.csv

$ head data/part2/luna/candidates.csv
seriesuid,coordX,coordY,coordZ,class
1.3...6860,-56.08,-67.85,-311.92,0
1.3...6860,53.21,-244.41,-245.17,0
1.3...6860,103.66,-121.8,-286.62,0
```

Counts the number of lines in the file

Prints the first few lines of the file

The first line of the .csv file defines the column headers.

```
1.3...6860,-33.66,-72.75,-308.41,0
...

$ grep ',1$' candidates.csv | wc -l
1351
```

Counts the number of lines that end with 1, which indicates malignancy

NOTE The values in the `seriesuid` column have been elided to better fit the printed page.

So we have 551,000 lines, each with a `seriesuid` (which we'll call `series_uid` in the code), some (X,Y,Z) coordinates, and a `class` column that corresponds to the nodule status (it's a Boolean value: 0 for a candidate that is not an actual nodule, and 1 for a candidate that is a nodule, either malignant or benign). We have 1,351 candidates flagged as actual nodules.

The annotations.csv file contains information about some of the candidates that have been flagged as nodules. We are interested in the `diameter_mm` information in particular:

This is a different number than in the candidates.csv file.

```
$ wc -l annotations.csv
1187 annotations.csv
```

The last column is also different.

```
$ head data/part2/luna/annotations.csv
seriesuid,coordX,coordY,coordZ,diameter_mm
1.3.6...6860,-128.6994211,-175.3192718,-298.3875064,5.651470635
1.3.6...6860,103.7836509,-211.9251487,-227.12125,4.224708481
1.3.6...5208,69.63901724,-140.9445859,876.3744957,5.786347814
1.3.6...0405,-24.0138242,192.1024053,-391.0812764,8.143261683
...
```

We have size information for about 1,200 nodules. This is useful, since we can use it to make sure our training and validation data includes a representative spread of nodule sizes. Without this, it's possible that our validation set could end up with only extreme values, making it seem as though our model is underperforming.

10.2.1 Training and validation sets

For any standard supervised learning task (classification is the prototypical example), we'll split our data into training and validation sets. We want to make sure both sets are *representative* of the range of real-world input data we're expecting to see and handle normally. If either set is meaningfully different from our real-world use cases, it's pretty likely that our model will behave differently than we expect—all of the training and statistics we collect won't be predictive once we transfer over to production use! We're not trying to make this an exact science, but you should keep an eye out in future projects for hints that you are training and testing on data that doesn't make sense for your operating environment.

Let's get back to our nodules. We're going to sort them by size and take every *N*th one for our validation set. That should give us the representative spread we're looking

for. Unfortunately, the location information provided in annotations.csv doesn't always precisely line up with the coordinates in candidates.csv:

```
$ grep 100225287222365663678666836860 annotations.csv
1.3.6...6860,-128.6994211,-175.3192718,-298.3875064,5.651470635
1.3.6...6860,103.7836509,-211.9251487,-227.12125,4.224708481

$ grep '100225287222365663678666836860.*,1$' candidates.csv
1.3.6...6860,104.16480444,-211.685591018,-227.011363746,1
1.3.6...6860,-128.94,-175.04,-297.87,1
```

These two coordinates are very close to each other.

If we truncate the corresponding coordinates from each file, we end up with (–128.70, –175.32,–298.39) versus (–128.94,–175.04,–297.87). Since the nodule in question has a diameter of 5 mm, both of these points are clearly meant to be the "center" of the nodule, but they don't line up exactly. It would be a perfectly valid response to decide that dealing with this data mismatch isn't worth it, and to ignore the file. We are going to do the legwork to make things line up, though, since real-world datasets are often imperfect this way, and this is a good example of the kind of work you will need to do to assemble data from disparate data sources.

10.2.2 Unifying our annotation and candidate data

Now that we know what our raw data files look like, let's build a `getCandidateInfoList` function that will stitch it all together. We'll use a named tuple that is defined at the top of the file to hold the information for each nodule.

Listing 10.1 dsets.py:7

```
from collections import namedtuple
# ... line 27
CandidateInfoTuple = namedtuple(
  'CandidateInfoTuple',
  'isNodule_bool, diameter_mm, series_uid, center_xyz',
)
```

These tuples are *not* our training samples, as they're missing the chunks of CT data we need. Instead, these represent a sanitized, cleaned, unified interface to the human-annotated data we're using. It's very important to isolate having to deal with messy data from model training. Otherwise, your training loop can get cluttered quickly, because you have to keep dealing with special cases and other distractions in the middle of code that should be focused on training.

> **TIP** Clearly separate the code that's responsible for data sanitization from the rest of your project. Don't be afraid to rewrite your data once and save it to disk if needed.

Our list of candidate information will have the nodule status (what we're going to be training the model to classify), diameter (useful for getting a good spread in training,

since large and small nodules will not have the same features), series (to locate the correct CT scan), and candidate center (to find the candidate in the larger CT). The function that will build a list of these `NoduleInfoTuple` instances starts by using an in-memory caching decorator, followed by getting the list of files present on disk.

Listing 10.2 dsets.py:32

Standard library in-memory caching

requireOnDisk_bool defaults to screening out series from data subsets that aren't in place yet.

```
@functools.lru_cache(1)
def getCandidateInfoList(requireOnDisk_bool=True):
  mhd_list = glob.glob('data-unversioned/part2/luna/subset*/*.mhd')
  presentOnDisk_set = {os.path.split(p)[-1][:-4] for p in mhd_list}
```

Since parsing some of the data files can be slow, we'll cache the results of this function call in memory. This will come in handy later, because we'll be calling this function more often in future chapters. Speeding up our data pipeline by carefully applying in-memory or on-disk caching can result in some pretty impressive gains in training speed. Keep an eye out for these opportunities as you work on your projects.

Earlier we said that we'll support running our training program with less than the full set of training data, due to the long download times and high disk space requirements. The `requireOnDisk_bool` parameter is what makes good on that promise; we're detecting which LUNA series UIDs are actually present and ready to be loaded from disk, and we'll use that information to limit which entries we use from the CSV files we're about to parse. Being able to run a subset of our data through the training loop can be useful to verify that the code is working as intended. Often a model's training results are bad to useless when doing so, but exercising our logging, metrics, model check-pointing, and similar functionality is beneficial.

After we get our candidate information, we want to merge in the diameter information from annotations.csv. First we need to group our annotations by `series_uid`, as that's the first key we'll use to cross-reference each row from the two files.

Listing 10.3 dsets.py:40, `def getCandidateInfoList`

```
diameter_dict = {}
with open('data/part2/luna/annotations.csv', "r") as f:
  for row in list(csv.reader(f))[1:]:
    series_uid = row[0]
    annotationCenter_xyz = tuple([float(x) for x in row[1:4]])
    annotationDiameter_mm = float(row[4])

    diameter_dict.setdefault(series_uid, []).append(
      (annotationCenter_xyz, annotationDiameter_mm)
    )
```

Now we'll build our full list of candidates using the information in the candidates.csv file.

Listing 10.4 dsets.py:51, `def getCandidateInfoList`

```
candidateInfo_list = []
with open('data/part2/luna/candidates.csv', "r") as f:
  for row in list(csv.reader(f))[1:]:
    series_uid = row[0]

    if series_uid not in presentOnDisk_set and requireOnDisk_bool:
      continue

    isNodule_bool = bool(int(row[4]))
    candidateCenter_xyz = tuple([float(x) for x in row[1:4]])

    candidateDiameter_mm = 0.0
    for annotation_tup in diameter_dict.get(series_uid, []):
      annotationCenter_xyz, annotationDiameter_mm = annotation_tup
      for i in range(3):
        delta_mm = abs(candidateCenter_xyz[i] - annotationCenter_xyz[i])
        if delta_mm > annotationDiameter_mm / 4:
          break
        else:
          candidateDiameter_mm = annotationDiameter_mm
          break

    candidateInfo_list.append(CandidateInfoTuple(
      isNodule_bool,
      candidateDiameter_mm,
      series_uid,
      candidateCenter_xyz,
    ))
```

If a series_uid isn't present, it's in a subset we don't have on disk, so we should skip it.

Divides the diameter by 2 to get the radius, and divides the radius by 2 to require that the two nodule center points not be too far apart relative to the size of the nodule. (This results in a bounding-box check, not a true distance check.)

For each of the candidate entries for a given `series_uid`, we loop through the annotations we collected earlier for the same `series_uid` and see if the two coordinates are close enough to consider them the same nodule. If they are, great! Now we have diameter information for that nodule. If we don't find a match, that's fine; we'll just treat the nodule as having a 0.0 diameter. Since we're only using this information to get a good spread of nodule sizes in our training and validation sets, having incorrect diameter sizes for some nodules shouldn't be a problem, but we should remember we're doing this in case our assumption here is wrong.

That's a lot of somewhat fiddly code just to merge in our nodule diameter. Unfortunately, having to do this kind of manipulation and fuzzy matching can be fairly common, depending on your raw data. Once we get to this point, however, we just need to sort the data and return it.

Listing 10.5 dsets.py:80, def getCandidateInfoList

```
candidateInfo_list.sort(reverse=True)
return candidateInfo_list
```

This means we have all of the actual nodule samples starting with the largest first, followed by all of the non-nodule samples (which don't have nodule size information).

The ordering of the tuple members in `noduleInfo_list` is driven by this sort. We're using this sorting approach to help ensure that when we take a slice of the data, that slice gets a representative chunk of the actual nodules with a good spread of nodule diameters. We'll discuss this more in section 10.5.3.

10.3 Loading individual CT scans

Next up, we need to be able to take our CT data from a pile of bits on disk and turn it into a Python object from which we can extract 3D nodule density data. We can see this path from the .mhd and .raw files to `Ct` objects in figure 10.4. Our nodule annotation information acts like a map to the interesting parts of our raw data. Before we can follow that map to our data of interest, we need to get the data into an addressable form.

TIP Having a large amount of raw data, most of which is uninteresting, is a common situation; look for ways to limit your scope to only the relevant data when working on your own projects.

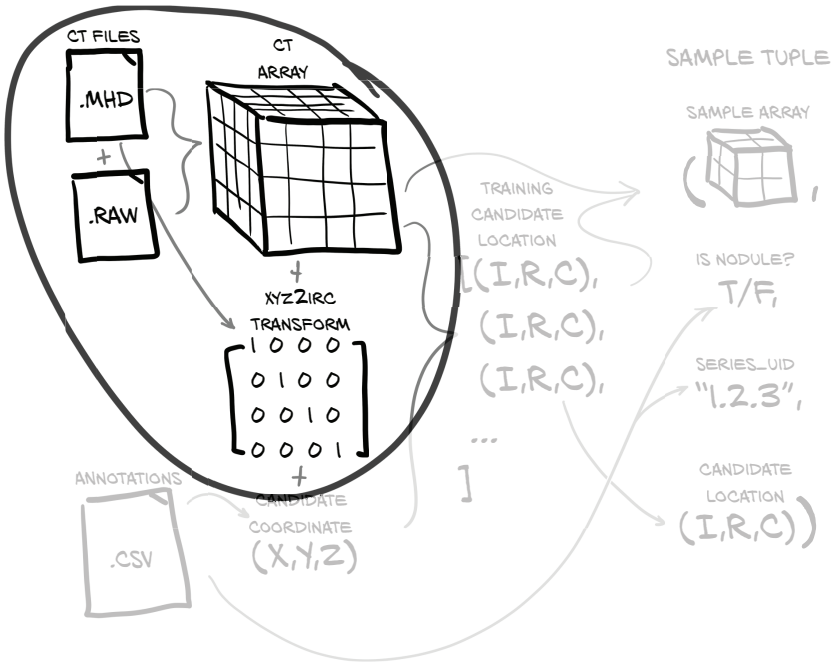

Figure 10.4 Loading a CT scan produces a voxel array and a transformation from patient coordinates to array indices.

The native file format for CT scans is DICOM (www.dicomstandard.org). The first version of the DICOM standard was authored in 1984, and as we might expect from anything computing-related that comes from that time period, it's a bit of a mess (for example, whole sections that are now retired were devoted to the data link layer protocol to use, since Ethernet hadn't won yet).

> **NOTE** We've done the legwork of finding the right library to parse these raw data files, but for other formats you've never heard of, you'll have to find a parser yourself. We recommend taking the time to do so! The Python ecosystem has parsers for just about every file format under the sun, and your time is almost certainly better spent working on the novel parts of your project than writing parsers for esoteric data formats.

Happily, LUNA has converted the data we're going to be using for this chapter into the MetaIO format, which is quite a bit easier to use (https://itk.org/Wiki/MetaIO/Documentation#Quick_Start). Don't worry if you've never heard of the format before! We can treat the format of the data files as a black box and use `SimpleITK` to load them into more familiar NumPy arrays.

Listing 10.6 dsets.py:9

```
import SimpleITK as sitk
# ... line 83
class Ct:
  def __init__(self, series_uid):
    mhd_path = glob.glob(
     'data-unversioned/part2/luna/subset*/{}.mhd'.format(series_uid)
     )[0]

    ct_mhd = sitk.ReadImage(mhd_path)
    ct_a = np.array(sitk.GetArrayFromImage(ct_mhd), dtype=np.float32)
```

We don't care to track which subset a given series_uid is in, so we wildcard the subset.

sitk.ReadImage implicitly consumes the .raw file in addition to the passed-in .mhd file.

Recreates an np.array since we want to convert the value type to np.float3

For real projects, you'll want to understand what types of information are contained in your raw data, but it's perfectly fine to rely on third-party code like `SimpleITK` to parse the bits on disk. Finding the right balance of knowing everything about your inputs versus blindly accepting whatever your data-loading library hands you will probably take some experience. Just remember that we're mostly concerned about *data*, not *bits*. It's the information that matters, not how it's represented.

Being able to uniquely identify a given sample of our data can be useful. For example, clearly communicating which sample is causing a problem or is getting poor classification results can drastically improve our ability to isolate and debug the issue. Depending on the nature of our samples, sometimes that unique identifier is an atom, like a number or a string, and sometimes it's more complicated, like a tuple.

We identify specific CT scans using the *series instance UID* (`series_uid`) assigned when the CT scan was created. DICOM makes heavy use of unique identifiers (UIDs)

for individual DICOM files, groups of files, courses of treatment, and so on. These identifiers are similar in concept to UUIDs (https://docs.python.org/3.6/library/uuid.html), but they have a different creation process and are formatted differently. For our purposes, we can treat them as opaque ASCII strings that serve as unique keys to reference the various CT scans. Officially, only the characters 0 through 9 and the period (.) are valid characters in a DICOM UID, but some DICOM files in the wild have been anonymized with routines that replace the UIDs with hexadecimal (0–9 and a–f) or other technically out-of-spec values (these out-of-spec values typically aren't flagged or cleaned by DICOM parsers; as we said before, it's a bit of a mess).

The 10 subsets we discussed earlier have about 90 CT scans each (888 in total), with every CT scan represented as two files: one with a .mhd extension and one with a .raw extension. The data being split between multiple files is hidden behind the `sitk` routines, however, and is not something we need to be directly concerned with.

At this point, `ct_a` is a three-dimensional array. All three dimensions are spatial, and the single intensity channel is implicit. As we saw in chapter 4, in a PyTorch tensor, the channel information is represented as a fourth dimension with size 1.

10.3.1 Hounsfield Units

Recall that earlier, we said that we need to understand our *data*, not the *bits* that store it. Here, we have a perfect example of that in action. Without understanding the nuances of our data's values and range, we'll end up feeding values into our model that will hinder its ability to learn what we want it to.

Continuing the `__init__` method, we need to do a bit of cleanup on the `ct_a` values. CT scan voxels are expressed in Hounsfield units (HU; https://en.wikipedia.org/wiki/Hounsfield_scale), which are odd units; air is –1,000 HU (close enough to 0 g/cc [grams per cubic centimeter] for our purposes), water is 0 HU (1 g/cc), and bone is at least +1,000 HU (2–3 g/cc).

> **NOTE** HU values are typically stored on disk as signed 12-bit integers (shoved into 16-bit integers), which fits well with the level of precision CT scanners can provide. While this is perhaps interesting, it's not particularly relevant to the project.

Some CT scanners use HU values that correspond to negative densities to indicate that those voxels are outside of the CT scanner's field of view. For our purposes, everything outside of the patient should be air, so we discard that field-of-view information by setting a lower bound of the values to –1,000 HU. Similarly, the exact densities of bones, metal implants, and so on are not relevant to our use case, so we cap density at roughly 2 g/cc (1,000 HU) even though that's not biologically accurate in most cases.

Listing 10.7 dsets.py:96, `Ct.__init__`

```
ct_a.clip(-1000, 1000, ct_a)
```

Values above 0 HU don't scale perfectly with density, but the tumors we're interested in are typically around 1 g/cc (0 HU), so we're going to ignore that HU doesn't map perfectly to common units like g/cc. That's fine, since our model will be trained to consume HU directly.

We want to remove all of these outlier values from our data: they aren't directly relevant to our goal, and having those outliers can make the model's job harder. This can happen in many ways, but a common example is when batch normalization is fed these outlier values and the statistics about how to best normalize the data are skewed. Always be on the lookout for ways to clean your data.

All of the values we've built are now assigned to `self`.

Listing 10.8 dsets.py:98, `Ct.__init__`

```
self.series_uid = series_uid
self.hu_a = ct_a
```

It's important to know that our data uses the range of –1,000 to +1,000, since in chapter 13 we end up adding channels of information to our samples. If we don't account for the disparity between HU and our additional data, those new channels can easily be overshadowed by the raw HU values. We won't add more channels of data for the classification step of our project, so we don't need to implement special handling right now.

10.4 Locating a nodule using the patient coordinate system

Deep learning models typically need fixed-size inputs,[2] due to having a fixed number of input neurons. We need to be able to produce a fixed-size array containing the candidate so that we can use it as input to our classifier. We'd like to train our model using a crop of the CT scan that has a candidate nicely centered, since then our model doesn't have to learn how to notice nodules tucked away in the corner of the input. By reducing the variation in expected inputs, we make the model's job easier.

10.4.1 The patient coordinate system

Unfortunately, all of the candidate center data we loaded in section 10.2 is expressed in millimeters, not voxels! We can't just plug locations in millimeters into an array index and expect everything to work out the way we want. As we can see in figure 10.5, we need to transform our coordinates from the millimeter-based coordinate system (X,Y,Z) they're expressed in, to the voxel-address-based coordinate system (I,R,C) used to take array slices from our CT scan data. This is a classic example of how it's important to handle units consistently!

As we have mentioned previously, when dealing with CT scans, we refer to the array dimensions as *index, row, and column,* because a separate meaning exists for X, Y, and Z,

[2] There are exceptions, but they're not relevant right now.

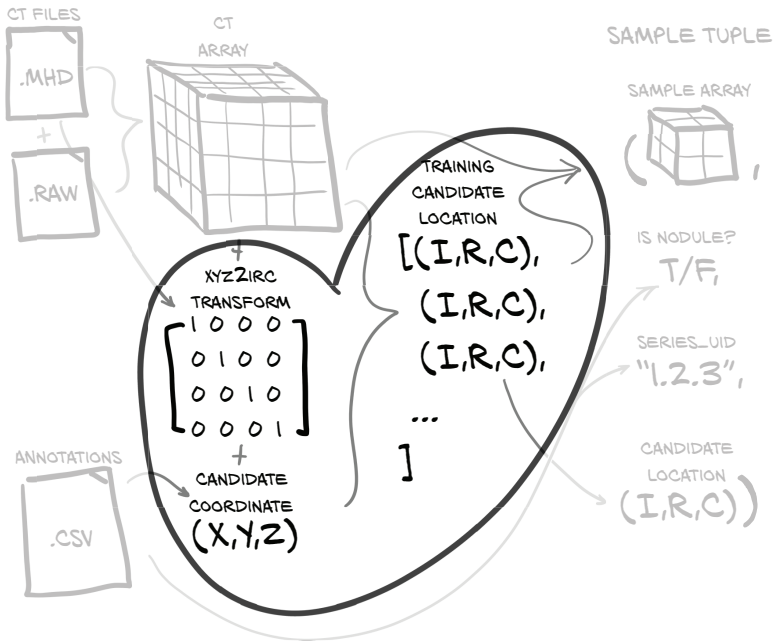

Figure 10.5 Using the transformation information to convert a nodule center coordinate in patient coordinates (X,Y,Z) to an array index (Index,Row,Column).

as illustrated in figure 10.6. The *patient coordinate system* defines positive X to be patient-left (*left*), positive Y to be patient-behind (*posterior*), and positive Z to be toward-patient-head (*superior*). Left-posterior-superior is sometimes abbreviated *LPS*.

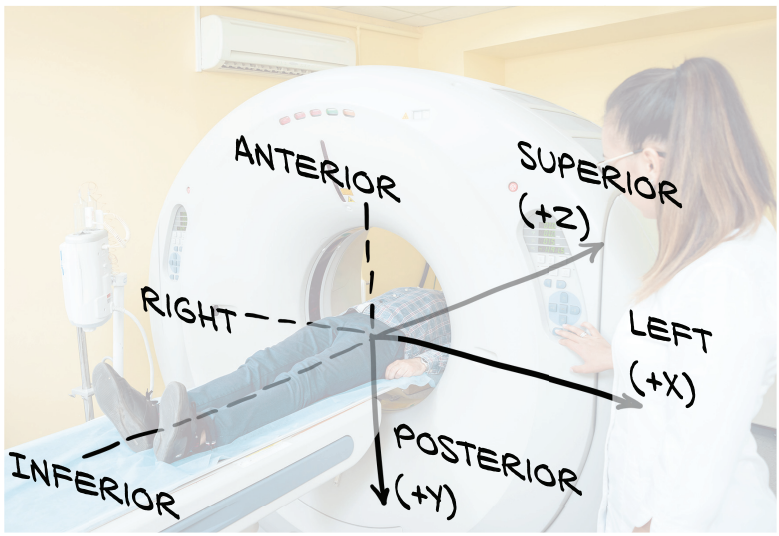

Figure 10.6 Our inappropriately clothed patient demonstrating the axes of the patient coordinate system

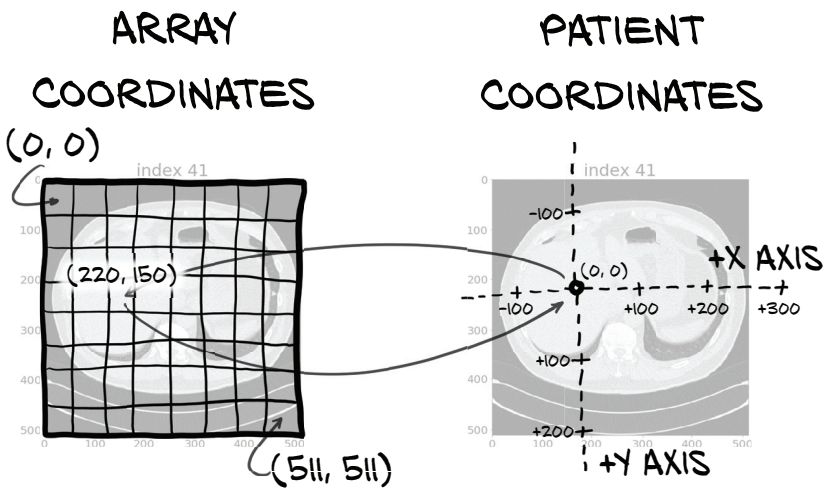

Figure 10.7 Array coordinates and patient coordinates have different origins and scaling.

The patient coordinate system is measured in millimeters and has an arbitrarily positioned origin that does not correspond to the origin of the CT voxel array, as shown in figure 10.7.

The patient coordinate system is often used to specify the locations of interesting anatomy in a way that is independent of any particular scan. The metadata that defines the relationship between the CT array and the patient coordinate system is stored in the header of DICOM files, and that meta-image format preserves the data in its header as well. This metadata allows us to construct the transformation from (X,Y,Z) to (I,R,C) that we saw in figure 10.5. The raw data contains many other fields of similar metadata, but since we don't have a use for them right now, those unneeded fields will be ignored.

10.4.2 CT scan shape and voxel sizes

One of the most common variations between CT scans is the size of the voxels; typically, they are not cubes. Instead, they can be 1.125 mm × 1.125 mm × 2.5 mm or similar. Usually the row and column dimensions have voxel sizes that are the same, and the index dimension has a larger value, but other ratios can exist.

When plotted using square pixels, the non-cubic voxels can end up looking somewhat distorted, similar to the distortion near the north and south poles when using a Mercator projection map. That's an imperfect analogy, since in this case the distortion is uniform and linear—the patient looks far more squat or barrel-chested in figure 10.8 than they would in reality. We will need to apply a scaling factor if we want the images to depict realistic proportions.

Knowing these kinds of details can help when trying to interpret our results visually. Without this information, it would be easy to assume that something was wrong with our data loading: we might think the data looked so squat because we were skipping half of

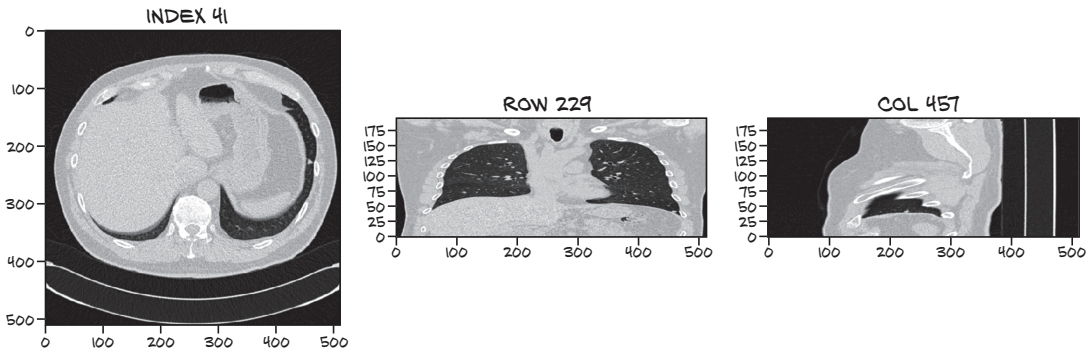

Figure 10.8 A CT scan with non-cubic voxels along the index-axis. Note how compressed the lungs are from top to bottom.

the slices by accident, or something along those lines. It can be easy to waste a lot of time debugging something that's been working all along, and being familiar with your data can help prevent that.

CTs are commonly 512 rows by 512 columns, with the index dimension ranging from around 100 total slices up to perhaps 250 slices (250 slices times 2.5 millimeters is typically enough to contain the anatomical region of interest). This results in a lower bound of approximately 2^{25} voxels, or about 32 million data points. Each CT specifies the voxel size in millimeters as part of the file metadata; for example, we'll call `ct_mhd.GetSpacing()` in listing 10.10.

10.4.3 Converting between millimeters and voxel addresses

We will define some utility code to assist with the conversion between patient coordinates in millimeters (which we will denote in the code with an `_xyz` suffix on variables and the like) and (I,R,C) array coordinates (which we will denote in code with an `_irc` suffix).

You might wonder whether the `SimpleITK` library comes with utility functions to convert these. And indeed, an `Image` instance does feature two methods—`TransformIndexToPhysicalPoint` and `TransformPhysicalPointToIndex`—to do just that (except shuffling from CRI [column,row,index] IRC). However, we want to be able to do this computation without keeping the `Image` object around, so we'll perform the math manually here.

Flipping the axes (and potentially a rotation or other transforms) is encoded in a 3 × 3 matrix returned as a tuple from `ct_mhd.GetDirections()`. To go from voxel indices to coordinates, we need to follow these four steps in order:

1. Flip the coordinates from IRC to CRI, to align with XYZ.
2. Scale the indices with the voxel sizes.
3. Matrix-multiply with the directions matrix, using `@` in Python.
4. Add the offset for the origin.

To go back from XYZ to IRC, we need to perform the inverse of each step in the reverse order.

We keep the voxel sizes in named tuples, so we convert these into arrays.

Listing 10.9 util.py:16

```
IrcTuple = collections.namedtuple('IrcTuple', ['index', 'row', 'col'])
XyzTuple = collections.namedtuple('XyzTuple', ['x', 'y', 'z'])

def irc2xyz(coord_irc, origin_xyz, vxSize_xyz, direction_a):
  cri_a = np.array(coord_irc)[::-1]
  origin_a = np.array(origin_xyz)
  vxSize_a = np.array(vxSize_xyz)
  coords_xyz = (direction_a @ (cri_a * vxSize_a)) + origin_a
  return XyzTuple(*coords_xyz)

def xyz2irc(coord_xyz, origin_xyz, vxSize_xyz, direction_a):
  origin_a = np.array(origin_xyz)
  vxSize_a = np.array(vxSize_xyz)
  coord_a = np.array(coord_xyz)
  cri_a = ((coord_a - origin_a) @ np.linalg.inv(direction_a)) / vxSize_a
  cri_a = np.round(cri_a)
  return IrcTuple(int(cri_a[2]), int(cri_a[1]), int(cri_a[0]))
```

Swaps the order while we convert to a NumPy array

The bottom three steps of our plan, all in one line

Inverse of the last three steps

Sneaks in proper rounding before converting to integers

Shuffles and converts to integers

Phew. If that was a bit heavy, don't worry. Just remember that we need to convert and use the functions as a black box. The metadata we need to convert from patient coordinates (`_xyz`) to array coordinates (`_irc`) is contained in the MetaIO file alongside the CT data itself. We pull the voxel sizing and positioning metadata out of the .mhd file at the same time we get the `ct_a`.

Listing 10.10 dsets.py:72, `class Ct`

```
class Ct:
  def __init__(self, series_uid):
    mhd_path = glob.glob('data-
     unversioned/part2/luna/subset*/{}.mhd'.format(series_uid))[0]

        ct_mhd = sitk.ReadImage(mhd_path)
        # ... line 91
        self.origin_xyz = XyzTuple(*ct_mhd.GetOrigin())
        self.vxSize_xyz = XyzTuple(*ct_mhd.GetSpacing())
        self.direction_a = np.array(ct_mhd.GetDirection()).reshape(3, 3)
```

Converts the directions to an array, and reshapes the nine-element array to its proper 3 × 3 matrix shape

These are the inputs we need to pass into our `xyz2irc` conversion function, in addition to the individual point to covert. With these attributes, our CT object implementation

now has all the data needed to convert a candidate center from patient coordinates to array coordinates.

10.4.4 Extracting a nodule from a CT scan

As we mentioned in chapter 9, up to 99.9999% of the voxels in a CT scan of a patient with a lung nodule won't be part of the actual nodule (or cancer, for that matter). Again, that ratio is equivalent to a two-pixel blob of incorrectly tinted color somewhere on a high-definition television, or a single misspelled word out of a shelf of novels. Forcing our model to examine such huge swaths of data looking for the hints of the nodules we want it to focus on is going to work about as well as asking you to find a single misspelled word from a set of novels written in a language you don't know![3]

Instead, as we can see in figure 10.9, we will extract an area around each candidate and let the model focus on one candidate at a time. This is akin to letting you read individual paragraphs in that foreign language: still not an easy task, but far less daunting! Looking for ways to reduce the scope of the problem for our model can help, especially in the early stages of a project when we're trying to get our first working implementation up and running.

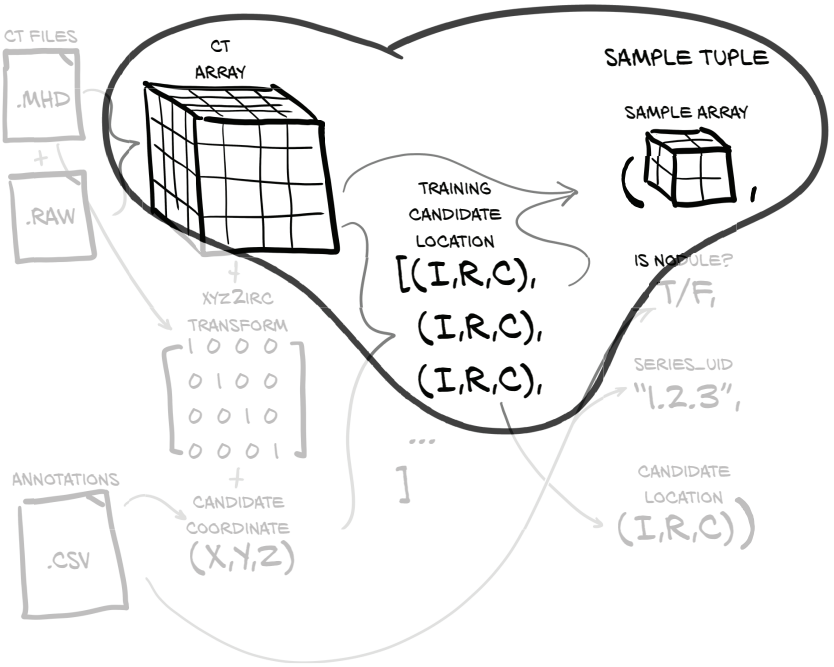

Figure 10.9 Cropping a candidate sample out of the larger CT voxel array using the candidate center's array coordinate information (Index,Row,Column)

[3] Have you found a misspelled word in this book yet? ;)

The `getRawNodule` function takes the center expressed in the patient coordinate system (X,Y,Z), just as it's specified in the LUNA CSV data, as well as a width in voxels. It returns a cubic chunk of CT, as well as the center of the candidate converted to array coordinates.

Listing 10.11 dsets.py:105, `Ct.getRawCandidate`

```
def getRawCandidate(self, center_xyz, width_irc):
  center_irc = xyz2irc(
    center_xyz,
    self.origin_xyz,
    self.vxSize_xyz,
    self.direction_a,
  )

  slice_list = []
  for axis, center_val in enumerate(center_irc):
    start_ndx = int(round(center_val - width_irc[axis]/2))
    end_ndx = int(start_ndx + width_irc[axis])
    slice_list.append(slice(start_ndx, end_ndx))

  ct_chunk = self.hu_a[tuple(slice_list)]

  return ct_chunk, center_irc
```

The actual implementation will need to deal with situations where the combination of center and width puts the edges of the cropped areas outside of the array. But as noted earlier, we will skip complications that obscure the larger intent of the function. The full implementation can be found on the book's website (www.manning.com/books/deep-learning-with-pytorch?query=pytorch) and in the GitHub repository (https://github.com/deep-learning-with-pytorch/dlwpt-code).

10.5 A straightforward dataset implementation

We first saw PyTorch `Dataset` instances in chapter 7, but this will be the first time we've implemented one ourselves. By subclassing `Dataset`, we will take our arbitrary data and plug it into the rest of the PyTorch ecosystem. Each `Ct` instance represents hundreds of different samples that we can use to train our model or validate its effectiveness. Our `LunaDataset` class will normalize those samples, flattening each CT's nodules into a single collection from which samples can be retrieved without regard for which `Ct` instance the sample originates from. This flattening is often how we want to process data, although as we'll see in chapter 12, in some situations a simple flattening of the data isn't enough to train a model well.

In terms of implementation, we are going to start with the requirements imposed from subclassing `Dataset` and work backward. This is different from the datasets we've worked with earlier; there we were using classes provided by external libraries, whereas here we need to implement and instantiate the class ourselves. Once we have done so, we can use it similarly to those earlier examples. Luckily, the implementation

of our custom subclass will not be too difficult, as the PyTorch API only requires that any `Dataset` subclasses we want to implement must provide these two functions:

- An implementation of `__len__` that must return a single, constant value after initialization (the value ends up being cached in some use cases)
- The `__getitem__` method, which takes an index and returns a tuple with sample data to be used for training (or validation, as the case may be)

First, let's see what the function signatures and return values of those functions look like.

Listing 10.12 dsets.py:176, `LunaDataset.__len__`

```
def __len__(self):
  return len(self.candidateInfo_list)

def __getitem__(self, ndx):
  # ... line 200
  return (
    candidate_t,  1((CO10-1))
    pos_t,  1((CO10-2))
    candidateInfo_tup.series_uid,
    torch.tensor(center_irc),
  )
```

This is our training sample.

Our `__len__` implementation is straightforward: we have a list of candidates, each candidate is a sample, and our dataset is as large as the number of samples we have. We don't have to make the implementation as simple as it is here; in later chapters, we'll see this change![4] The only rule is that if `__len__` returns a value of N, then `__getitem__` needs to return something valid for all inputs 0 to $N-1$.

For `__getitem__`, we take `ndx` (typically an integer, given the rule about supporting inputs 0 to $N-1$) and return the four-item sample tuple as depicted in figure 10.2. Building this tuple is a bit more complicated than getting the length of our dataset, however, so let's take a look.

The first part of this method implies that we need to construct `self.candidateInfo_list` as well as provide the `getCtRawNodule` function.

Listing 10.13 dsets.py:179, `LunaDataset.__getitem__`

```
def __getitem__(self, ndx):
  candidateInfo_tup = self.candidateInfo_list[ndx]
  width_irc = (32, 48, 48)

  candidate_a, center_irc = getCtRawCandidate(
    candidateInfo_tup.series_uid,
    candidateInfo_tup.center_xyz,
    width_irc,
  )
```

The return value candidate_a has shape (32,48,48); the axes are depth, height, and width.

[4] To something simpler, actually; but the point is, we have options.

We will get to those in a moment in sections 10.5.1 and 10.5.2.

The next thing we need to do in the `__getitem__` method is manipulate the data into the proper data types and required array dimensions that will be expected by downstream code.

Listing 10.14 dsets.py:189, `LunaDataset.__getitem__`

```
candidate_t = torch.from_numpy(candidate_a)
candidate_t = candidate_t.to(torch.float32)
candidate_t = candidate_t.unsqueeze(0)
```

.unsqueeze(0) adds the 'Channel' dimension.

Don't worry too much about exactly why we are manipulating dimensionality for now; the next chapter will contain the code that ends up consuming this output and imposing the constraints we're proactively meeting here. This *will* be something you should expect for every custom `Dataset` you implement. These conversions are a key part of transforming your Wild West data into nice, orderly tensors.

Finally, we need to build our classification tensor.

Listing 10.15 dsets.py:193, `LunaDataset.__getitem__`

```
pos_t = torch.tensor([
    not candidateInfo_tup.isNodule_bool,
    candidateInfo_tup.isNodule_bool
  ],
  dtype=torch.long,
)
```

This has two elements, one each for our possible candidate classes (nodule or non-nodule; or positive or negative, respectively). We could have a single output for the nodule status, but `nn.CrossEntropyLoss` expects one output value per class, so that's what we provide here. The exact details of the tensors you construct will change based on the type of project you're working on.

Let's take a look at our final sample tuple (the larger `nodule_t` output isn't particularly readable, so we elide most of it in the listing).

Listing 10.16 p2ch10_explore_data.ipynb

```
# In[10]:
LunaDataset()[0]

# Out[10]:
(tensor([[[[-899., -903., -825.,  ..., -901., -898., -893.],
           ...,
           [ -92.,  -63.,    4.,  ...,   63.,   70.,   52.]]]]),
 tensor([0, 1]),
 '1.3.6...287966244644280690737019247886',
 tensor([ 91, 360, 341]))
```

candidate_t

cls_t

candidate_tup.series_uid (elided)

center_irc

Here we see the four items from our `__getitem__` return statement.

10.5.1 Caching candidate arrays with the getCtRawCandidate function

In order to get decent performance out of `LunaDataset`, we'll need to invest in some on-disk caching. This will allow us to avoid having to read an entire CT scan from disk for every sample. Doing so would be prohibitively slow! Make sure you're paying attention to bottlenecks in your project and doing what you can to optimize them once they start slowing you down. We're kind of jumping the gun here since we haven't demonstrated that we need caching here. Without caching, the `LunaDataset` is easily 50 times slower! We'll revisit this in the chapter's exercises.

The function itself is easy. It's a file-cache-backed (https://pypi.python.org/pypi/diskcache) wrapper around the `Ct.getRawCandidate` method we saw earlier.

Listing 10.17 dsets.py:139

```
@functools.lru_cache(1, typed=True)
def getCt(series_uid):
  return Ct(series_uid)

@raw_cache.memoize(typed=True)
def getCtRawCandidate(series_uid, center_xyz, width_irc):
  ct = getCt(series_uid)
  ct_chunk, center_irc = ct.getRawCandidate(center_xyz, width_irc)
  return ct_chunk, center_irc
```

We use a few different caching methods here. First, we're caching the `getCt` return value in memory so that we can repeatedly ask for the same `Ct` instance without having to reload all of the data from disk. That's a huge speed increase in the case of repeated requests, but we're only keeping one CT in memory, so cache misses will be frequent if we're not careful about access order.

The `getCtRawCandidate` function that calls `getCt` *also* has its outputs cached, however; so after our cache is populated, `getCt` won't ever be called. These values are cached to disk using the Python library `diskcache`. We'll discuss why we have this specific caching setup in chapter 11. For now, it's enough to know that it's much, much faster to read in 2^{15} `float32` values from disk than it is to read in 2^{25} `int16` values, convert to `float32`, and then select a 2^{15} subset. From the second pass through the data forward, I/O times for input should drop to insignificance.

> **NOTE** If the definitions of these functions ever materially change, we will need to remove the cached values from disk. If we don't, the cache will continue to return them, even if now the function will not map the given inputs to the old output. The data is stored in the data-unversioned/cache directory.

10.5.2 Constructing our dataset in LunaDataset.__init__

Just about every project will need to separate samples into a training set and a validation set. We are going to do that here by designating every tenth sample, specified by the `val_stride` parameter, as a member of the validation set. We will also accept an `isValSet_bool` parameter and use it to determine whether we should keep only the training data, the validation data, or everything.

Listing 10.18 dsets.py:149, `class LunaDataset`

```
class LunaDataset(Dataset):
  def __init__(self,
          val_stride=0,
          isValSet_bool=None,
          series_uid=None,
      ):
    self.candidateInfo_list = copy.copy(getCandidateInfoList())

    if series_uid:
      self.candidateInfo_list = [
        x for x in self.candidateInfo_list if x.series_uid == series_uid
      ]
```

Copies the return value so the cached copy won't be impacted by altering self.candidateInfo_list

If we pass in a truthy `series_uid`, then the instance will only have nodules from that series. This can be useful for visualization or debugging, by making it easier to look at, for instance, a single problematic CT scan.

10.5.3 A training/validation split

We allow for the `Dataset` to partition out 1/*N*th of the data into a subset used for validating the model. How we will handle that subset is based on the value of the `isValSet_bool` argument.

Listing 10.19 dsets.py:162, `LunaDataset.__init__`

```
if isValSet_bool:
  assert val_stride > 0, val_stride
  self.candidateInfo_list = self.candidateInfo_list[::val_stride]
  assert self.candidateInfo_list
elif val_stride > 0:
  del self.candidateInfo_list[::val_stride]
  assert self.candidateInfo_list
```

Deletes the validation images (every val_stride-th item in the list) from self.candidateInfo_list. We made a copy earlier so that we don't alter the original list.

This means we can create two `Dataset` instances and be confident that there is strict segregation between our training data and our validation data. Of course, this depends on there being a consistent sorted order to `self.candidateInfo_list`, which we ensure by having there be a stable sorted order to the candidate info tuples, and by the `getCandidateInfoList` function sorting the list before returning it.

The other caveat regarding separation of training and validation data is that, depending on the task at hand, we might need to ensure that data from a single patient is only present either in training or in testing but not both. Here this is not a problem; otherwise, we would have needed to split the list of patients and CT scans before going to the level of nodules.

Let's take a look at the data using `p2ch10_explore_data.ipynb`:

```
# In[2]:
from p2ch10.dsets import getCandidateInfoList, getCt, LunaDataset
candidateInfo_list = getCandidateInfoList(requireOnDisk_bool=False)
positiveInfo_list = [x for x in candidateInfo_list if x[0]]
diameter_list = [x[1] for x in positiveInfo_list]

# In[4]:
for i in range(0, len(diameter_list), 100):
    print('{:4}  {:4.1f} mm'.format(i, diameter_list[i]))

# Out[4]:
   0  32.3 mm
 100  17.7 mm
 200  13.0 mm
 300  10.0 mm
 400   8.2 mm
 500   7.0 mm
 600   6.3 mm
 700   5.7 mm
 800   5.1 mm
 900   4.7 mm
1000   4.0 mm
1100   0.0 mm
1200   0.0 mm
1300   0.0 mm
```

We have a few very large candidates, starting at 32 mm, but they rapidly drop off to half that size. The bulk of the candidates are in the 4 to 10 mm range, and several hundred don't have size information at all. This looks as expected; you might recall that we had more actual nodules than we had diameter annotations. Quick sanity checks on your data can be very helpful; catching a problem or mistaken assumption early may save hours of effort!

The larger takeaway is that our training and validation splits should have a few properties in order to work well:

- Both sets should include examples of all variations of expected inputs.
- Neither set should have samples that aren't representative of expected inputs *unless* they have a specific purpose like training the model to be robust to outliers.
- The training set shouldn't offer unfair hints about the validation set that wouldn't be true for real-world data (for example, including the same sample in both sets; this is known as a *leak* in the training set).

10.5.4 Rendering the data

Again, either use p2ch10_explore_data.ipynb directly or start Jupyter Notebook and enter

```
# In[7]:
%matplotlib inline
from p2ch10.vis import findNoduleSamples, showNodule
noduleSample_list = findNoduleSamples()
```

This magic line sets up the ability for images to be displayed inline via the notebook.

> **TIP** For more information about Jupyter's matplotlib inline magic,[5] please see http://mng.bz/rrmD.

```
# In[8]:
series_uid = positiveSample_list[11][2]
showCandidate(series_uid)
```

This produces images akin to those showing CT and nodule slices earlier in this chapter.

If you're interested, we invite you to edit the implementation of the rendering code in p2ch10/vis.py to match your needs and tastes. The rendering code makes heavy use of Matplotlib (https://matplotlib.org), which is too complex a library for us to attempt to cover here.

Remember that rendering your data is not just about getting nifty-looking pictures. The point is to get an intuitive sense of what your inputs look like. Being able to tell at a glance "This problematic sample is very noisy compared to the rest of my data" or "That's odd, this looks pretty normal" can be useful when investigating issues. Effective rendering also helps foster insights like "Perhaps if I modify things like *so*, I can solve the issue I'm having." That level of familiarity will be necessary as you start tackling harder and harder projects.

> **NOTE** Due to the way each subset has been partitioned, combined with the sorting used when constructing `LunaDataset.candidateInfo_list`, the ordering of the entries in `noduleSample_list` is highly dependent on which subsets are present at the time the code is executed. Please remember this when trying to find a particular sample a second time, especially after decompressing more subsets.

10.6 Conclusion

In chapter 9, we got our heads wrapped around our data. In this chapter, we got *PyTorch's* head wrapped around our data! By transforming our DICOM-via-meta-image raw data into tensors, we've set the stage to start implementing a model and a training loop, which we'll see in the next chapter.

It's important not to underestimate the impact of the design decisions we've already made: the size of our inputs, the structure of our caching, and how we're partitioning our training and validation sets will all make a difference to the success or

[5] Their term, not ours!

failure of our overall project. Don't hesitate to revisit these decisions later, especially once you're working on your own projects.

10.7 Exercises

1. Implement a program that iterates through a `LunaDataset` instance, and time how long it takes to do so. In the interest of time, it might make sense to have an option to limit the iterations to the first `N=1000` samples.
   a. How long does it take to run the first time?
   b. How long does it take to run the second time?
   c. What does clearing the cache do to the runtime?
   d. What does using the *last* `N=1000` samples do to the first/second runtime?
2. Change the `LunaDataset` implementation to randomize the sample list during `__init__`. Clear the cache, and run the modified version. What does that do to the runtime of the first and second runs?
3. Revert the randomization, and comment out the `@functools.lru_cache(1, typed=True)` decorator to `getCt`. Clear the cache, and run the modified version. How does the runtime change now?

10.8 Summary

- Often, the code required to parse and load raw data is nontrivial. For this project, we implement a `Ct` class that loads data from disk and provides access to cropped regions around points of interest.
- Caching can be useful if the parsing and loading routines are expensive. Keep in mind that some caching can be done in memory, and some is best performed on disk. Each can have its place in a data-loading pipeline.
- PyTorch `Dataset` subclasses are used to convert data from its native form into tensors suitable to pass in to the model. We can use this functionality to integrate our real-world data with PyTorch APIs.
- Subclasses of `Dataset` need to provide implementations for two methods: `__len__` and `__getitem__`. Other helper methods are allowed but not required.
- Splitting our data into a sensible training set and a validation set requires that we make sure no sample is in both sets. We accomplish this here by using a consistent sort order and taking every tenth sample for our validation set.
- Data visualization is important; being able to investigate data visually can provide important clues about errors or problems. We are using Jupyter Notebooks and Matplotlib to render our data.

11 Training a classification model to detect suspected tumors

This chapter covers

- Using PyTorch `DataLoader`s to load data
- Implementing a model that performs classification on our CT data
- Setting up the basic skeleton for our application
- Logging and displaying metrics

In the previous chapters, we set the stage for our cancer-detection project. We covered medical details of lung cancer, took a look at the main data sources we will use for our project, and transformed our raw CT scans into a PyTorch `Dataset` instance. Now that we have a dataset, we can easily consume our training data. So let's do that!

11.1 A foundational model and training loop

We're going to do two main things in this chapter. We'll start by building the nodule classification model and training loop that will be the foundation that the rest of part 2 uses to explore the larger project. To do that, we'll use the `Ct` and `LunaDataset` classes we implemented in chapter 10 to feed `DataLoader` instances. Those instances, in turn, will feed our classification model with data via training and validation loops.

We'll finish the chapter by using the results from running that training loop to introduce one of the hardest challenges in this part of the book: how to get high-quality results from messy, limited data. In later chapters, we'll explore the specific ways in which our data is limited, as well as mitigate those limitations.

Let's recall our high-level roadmap from chapter 9, shown here in figure 11.1. Right now, we'll work on producing a model capable of performing step 4: classification. As a reminder, we will classify candidates as nodules or non-nodules (we'll build another classifier to attempt to tell malignant nodules from benign ones in chapter 14). That means we're going to assign a single, specific label to each sample that we present to the model. In this case, those labels are "nodule" and "non-nodule," since each sample represents a single candidate.

Getting an early end-to-end version of a meaningful part of your project is a great milestone to reach. Having something that works well enough for the results to be evaluated analytically let's you move forward with future changes, confident that you

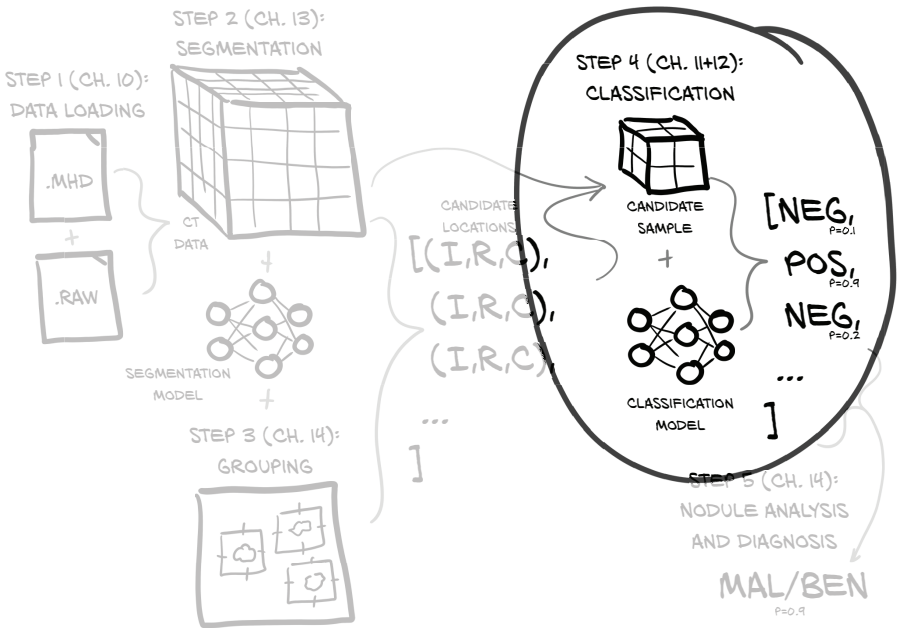

Figure 11.1 Our end-to-end project to detect lung cancer, with a focus on this chapter's topic: step 4, classification

are improving your results with each change—or at least that you're able to set aside any changes and experiments that don't work out! Expect to have to do a lot of experimentation when working on your own projects. Getting the best results will usually require considerable tinkering and tweaking.

But before we can get to the experimental phase, we must lay our foundation. Let's see what our part 2 training loop looks like in figure 11.2: it should seem generally familiar, given that we saw a similar set of core steps in chapter 5. Here we will also use a validation set to evaluate our training progress, as discussed in section 5.5.3.

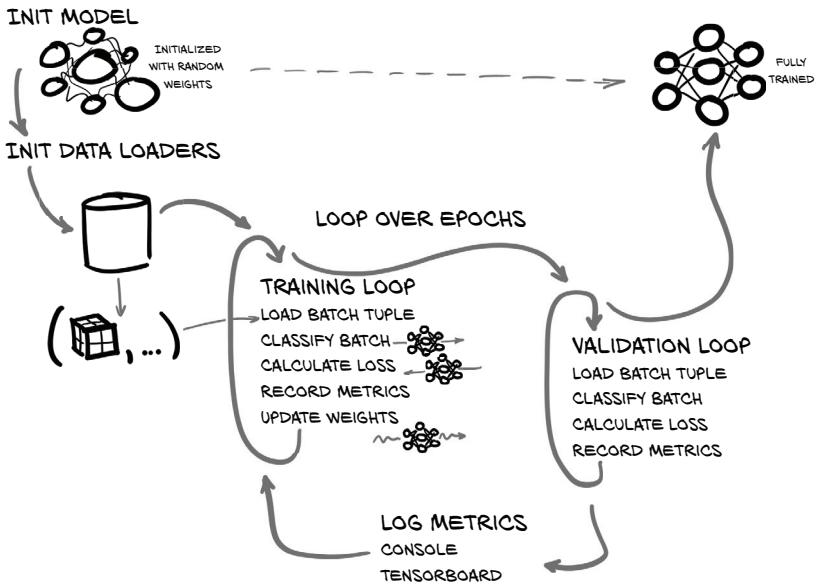

Figure 11.2 The training and validation script we will implement in this chapter

The basic structure of what we're going to implement is as follows:

- Initialize our model and data loading.
- Loop over a semi-arbitrarily chosen number of epochs.
  - Loop over each batch of training data returned by `LunaDataset`.
  - The data-loader worker process loads the relevant batch of data in the background.
  - Pass the batch into our classification model to get results.
  - Calculate our loss based on the difference between our predicted results and our ground-truth data.
  - Record metrics about our model's performance into a temporary data structure.
  - Update the model weights via backpropagation of the error.

- Loop over each batch of validation data (in a manner very similar to the training loop).
- Load the relevant batch of validation data (again, in the background worker process).
- Classify the batch, and compute the loss.
- Record information about how well the model performed on the validation data.
- Print out progress and performance information for this epoch.

As we go through the code for the chapter, keep an eye out for two main differences between the code we're producing here and what we used for a training loop in part 1. First, we'll put more structure around our program, since the project as a whole is quite a bit more complicated than what we did in earlier chapters. Without that extra structure, the code can get messy quickly. And for this project, we will have our main training application use a number of well-contained functions, and we will further separate code for things like our dataset into self-contained Python modules.

Make sure that for your own projects, you match the level of structure and design to the complexity level of your project. Too little structure, and it will become difficult to perform experiments cleanly, troubleshoot problems, or even describe what you're doing! Conversely, too *much* structure means you're wasting time writing infrastructure that you don't need and most likely slowing yourself down by having to conform to it after all that plumbing is in place. Plus it can be tempting to spend time on infrastructure as a procrastination tactic, rather than digging into the hard work of making actual progress on your project. Don't fall into that trap!

The other big difference between this chapter's code and part 1 will be a focus on collecting a variety of metrics about how training is progressing. Being able to accurately determine the impact of changes on training is impossible without having good metrics logging. Without spoiling the next chapter, we'll also see how important it is to collect not just metrics, but the *right metrics for the job*. We'll lay the infrastructure for tracking those metrics in this chapter, and we'll exercise that infrastructure by collecting and displaying the loss and percent of samples correctly classified, both overall and per class. That's enough to get us started, but we'll cover a more realistic set of metrics in chapter 12.

11.2 The main entry point for our application

One of the big structural differences from earlier training work we've done in this book is that part 2 wraps our work in a fully fledged command-line application. It will parse command-line arguments, have a full-featured `--help` command, and be easy to run in a wide variety of environments. All this will allow us to easily invoke the training routines from both Jupyter and a Bash shell.[1]

[1] Any shell, really, but if you're using a non-Bash shell, you already knew that.

Our application's functionality will be implemented via a class so that we can instantiate the application and pass it around if we feel the need. This can make testing, debugging, or invocation from other Python programs easier. We can invoke the application without needing to spin up a second OS-level process (we won't do explicit unit testing in this book, but the structure we create can be helpful for real projects where that kind of testing is appropriate).

One way to take advantage of being able to invoke our training by either function call or OS-level process is to wrap the function invocations into a Jupyter Notebook so the code can easily be called from either the native CLI or the browser.

Listing 11.1 code/p2_run_everything.ipynb

```
# In[2]:w
def run(app, *argv):
    argv = list(argv)
    argv.insert(0, '--num-workers=4')
    log.info("Running: {}({!r}).main()".format(app, argv))

    app_cls = importstr(*app.rsplit('.', 1))
    app_cls(argv).main()

    log.info("Finished: {}.{!r}).main()".format(app, argv))

# In[6]:
run('p2ch11.training.LunaTrainingApp', '--epochs=1')
```

We assume you have a four-core, eight-thread CPU. Change the 4 if needed.

This is a slightly cleaner call to __import__.

> **NOTE** The training here assumes that you're on a workstation that has a four-core, eight-thread CPU, 16 GB of RAM, and a GPU with 8 GB of RAM. Reduce `--batch-size` if your GPU has less RAM, and `--num-workers` if you have fewer CPU cores, or less CPU RAM.

Let's get some semistandard boilerplate code out of the way. We'll start at the end of the file with a pretty standard `if main` stanza that instantiates the application object and invokes the `main` method.

Listing 11.2 training.py:386

```
if __name__ == '__main__':
  LunaTrainingApp().main()
```

From there, we can jump back to the top of the file and have a look at the application class and the two functions we just called, `__init__` and `main`. We'll want to be able to accept command-line arguments, so we'll use the standard `argparse` library (https://docs.python.org/3/library/argparse.html) in the application's `__init__` function. Note that we can pass in custom arguments to the initializer, should we wish to do so. The `main` method will be the primary entry point for the core logic of the application.

Listing 11.3 training.py:31, `class LunaTrainingApp`

```
class LunaTrainingApp:
  def __init__(self, sys_argv=None):
    if sys_argv is None:
       sys_argv = sys.argv[1:]

    parser = argparse.ArgumentParser()
    parser.add_argument('--num-workers',
      help='Number of worker processes for background data loading',
      default=8,
      type=int,
    )
    # ... line 63
    self.cli_args = parser.parse_args(sys_argv)
    self.time_str = datetime.datetime.now().strftime('%Y-%m-%d_%H.%M.%S')

  # ... line 137
  def main(self):
    log.info("Starting {}, {}".format(type(self).__name__, self.cli_args))
```

If the caller doesn't provide arguments, we get them from the command line.

We'll use the timestamp to help identify training runs.

This structure is pretty general and could be reused for future projects. In particular, parsing arguments in `__init__` allows us to configure the application separately from invoking it.

If you check the code for this chapter on the book's website or GitHub, you might notice some extra lines mentioning `TensorBoard`. Ignore those for now; we'll discuss them in detail later in the chapter, in section 11.9.

11.3 Pretraining setup and initialization

Before we can begin iterating over each batch in our epoch, some initialization work needs to happen. After all, we can't train a model if we haven't even instantiated one yet! We need to do two main things, as we can see in figure 11.3. The first, as we just mentioned, is to initialize our model and optimizer; and the second is to initialize our `Dataset` and `DataLoader` instances. `LunaDataset` will define the randomized set of samples that will make up our training epoch, and our `DataLoader` instance will perform the work of loading the data out of our dataset and providing it to our application.

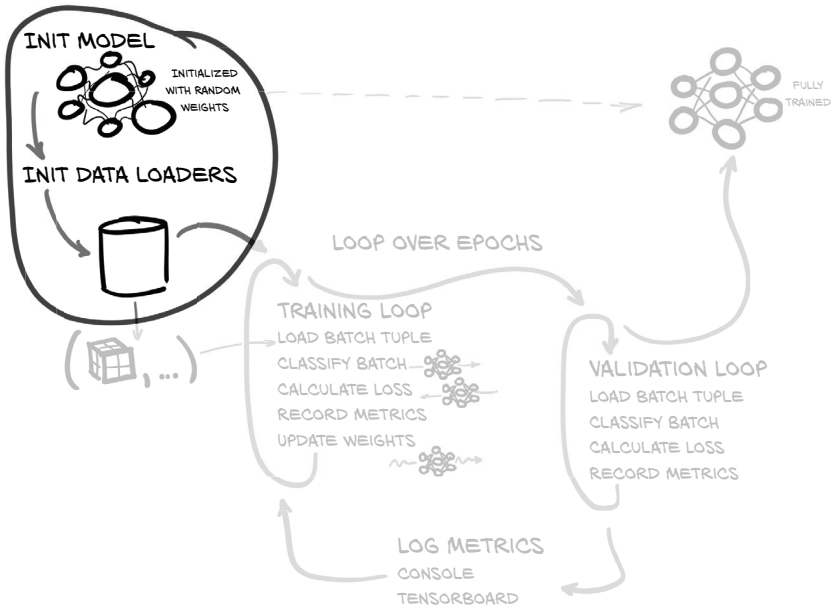

Figure 11.3 The training and validation script we will implement in this chapter, with a focus on the preloop variable initialization

11.3.1 Initializing the model and optimizer

For this section, we are treating the details of `LunaModel` as a black box. In section 11.4, we will detail the internal workings. You are welcome to explore changes to the implementation to better meet our goals for the model, although that's probably best done after finishing at least chapter 12.

Let's see what our starting point looks like.

Listing 11.4 training.py:31, `class LunaTrainingApp`

```
class LunaTrainingApp:
  def __init__(self, sys_argv=None):
    # ... line 70
    self.use_cuda = torch.cuda.is_available()
    self.device = torch.device("cuda" if self.use_cuda else "cpu")

    self.model = self.initModel()
    self.optimizer = self.initOptimizer()
```

```
def initModel(self):
  model = LunaModel()
  if self.use_cuda:
    log.info("Using CUDA; {} devices.".format(torch.cuda.device_count()))
    if torch.cuda.device_count() > 1:
       model = nn.DataParallel(model)
    model = model.to(self.device)
  return model

def initOptimizer(self):
  return SGD(self.model.parameters(), lr=0.001, momentum=0.99)
```

Detects multiple GPUs

Wraps the model

Sends model parameters to the GPU

If the system used for training has more than one GPU, we will use the `nn.DataParallel` class to distribute the work between all of the GPUs in the system and then collect and resync parameter updates and so on. This is almost entirely transparent in terms of both the model implementation and the code that uses that model.

DataParallel vs. DistributedDataParallel

In this book, we use `DataParallel` to handle utilizing multiple GPUs. We chose `DataParallel` because it's a simple drop-in wrapper around our existing models. It is not the best-performing solution for using multiple GPUs, however, and it is limited to working with the hardware available in a single machine.

PyTorch also provides `DistributedDataParallel`, which is the recommended wrapper class to use when you need to spread work between more than one GPU or machine. Since the proper setup and configuration are nontrivial, and we suspect that the vast majority of our readers won't see any benefit from the complexity, we won't cover `DistributedDataParallel` in this book. If you wish to learn more, we suggest reading the official documentation: https://pytorch.org/tutorials/intermediate/ddp_tutorial.html.

Assuming that `self.use_cuda` is true, the call `self.model.to(device)` moves the model parameters to the GPU, setting up the various convolutions and other calculations to use the GPU for the heavy numerical lifting. It's important to do so before constructing the optimizer, since, otherwise, the optimizer would be left looking at the CPU-based parameter objects rather than those copied to the GPU.

For our optimizer, we'll use basic stochastic gradient descent (SGD; https://pytorch.org/docs/stable/optim.html#torch.optim.SGD) with momentum. We first saw this optimizer in chapter 5. Recall from part 1 that many different optimizers are available in PyTorch; while we won't cover most of them in any detail, the official documentation (https://pytorch.org/docs/stable/optim.html#algorithms) does a good job of linking to the relevant papers.

Using SGD is generally considered a safe place to start when it comes to picking an optimizer; there are some problems that might not work well with SGD, but they're relatively rare. Similarly, a learning rate of 0.001 and a momentum of 0.9 are pretty safe choices. Empirically, SGD with those values has worked reasonably well for a wide range of projects, and it's easy to try a learning rate of 0.01 or 0.0001 if things aren't working well right out of the box.

That's not to say any of those values is the best for our use case, but trying to find better ones is getting ahead of ourselves. Systematically trying different values for learning rate, momentum, network size, and other similar configuration settings is called a *hyperparameter search*. There are other, more glaring issues we need to address first in the coming chapters. Once we address those, we can begin to fine-tune these values. As we mentioned in the section "Testing other optimizers" in chapter 5, there are also other, more exotic optimizers we might choose; but other than perhaps swapping `torch.optim.SGD` for `torch.optim.Adam`, understanding the trade-offs involved is a topic too advanced for this book.

11.3.2 Care and feeding of data loaders

The `LunaDataset` class that we built in the last chapter acts as the bridge between whatever Wild West data we have and the somewhat more structured world of tensors that the PyTorch building blocks expect. For example, `torch.nn.Conv3d` (https://pytorch.org/docs/stable/nn.html#conv3d) expects five-dimensional input: (*N*, *C*, *D*, *H*, *W*): number of samples, channels per sample, depth, height, and width. Quite different from the native 3D our CT provides!

You may recall the `ct_t.unsqueeze(0)` call in `LunaDataset.__getitem__` from the last chapter; it provides the fourth dimension, a "channel" for our data. Recall from chapter 4 that an RGB image has three channels, one each for red, green, and blue. Astronomical data could have dozens, one each for various slices of the electromagnetic spectrum—gamma rays, X-rays, ultraviolet light, visible light, infrared, microwaves, and/or radio waves. Since CT scans are single-intensity, our channel dimension is only size 1.

Also recall from part 1 that training on single samples at a time is typically an inefficient use of computing resources, because most processing platforms are capable of more parallel calculations than are required by a model to process a single training or validation sample. The solution is to group sample tuples together into a batch tuple, as in figure 11.4, allowing multiple samples to be processed at the same time. The fifth dimension (*N*) differentiates multiple samples in the same batch.

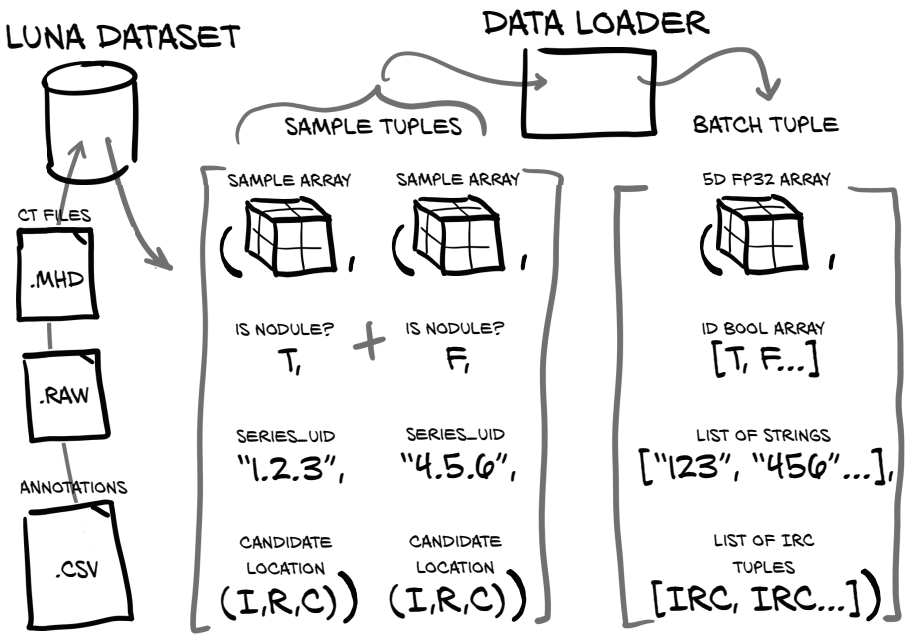

Figure 11.4 Sample tuples being collated into a single batch tuple inside a data loader

Conveniently, we don't have to implement any of this batching: the PyTorch `DataLoader` class will handle all of the collation work for us. We've already built the bridge from the CT scans to PyTorch tensors with our `LunaDataset` class, so all that remains is to plug our dataset into a data loader.

Listing 11.5 training.py:89, `LunaTrainingApp.initTrainDl`

```
def initTrainDl(self):
  train_ds = LunaDataset(          <-- Our custom dataset
    val_stride=10,
    isValSet_bool=False,
  )

  batch_size = self.cli_args.batch_size
  if self.use_cuda:
    batch_size *= torch.cuda.device_count()

  train_dl = DataLoader(           <-- An off-the-shelf class
    train_ds,
    batch_size=batch_size,                  <-- Batching is done automatically.
    num_workers=self.cli_args.num_workers,
    pin_memory=self.use_cuda,               <-- Pinned memory transfers to GPU quickly.
  )
```

```
  return train_dl

# ... line 137
def main(self):
  train_dl = self.initTrainDl()
  val_dl = self.initValDl()
```

The validation data loader is very similar to training.

In addition to batching individual samples, data loaders can also provide parallel loading of data by using separate processes and shared memory. All we need to do is specify `num_workers=…` when instantiating the data loader, and the rest is taken care of behind the scenes. Each worker process produces complete batches as in figure 11.4. This helps make sure hungry GPUs are well fed with data. Our `validation_ds` and `validation_dl` instances look similar, except for the obvious `isValSet_bool=True`.

When we iterate, like `for batch_tup in self.train_dl:`, we won't have to wait for each `Ct` to be loaded, samples to be taken and batched, and so on. Instead, we'll get the already loaded `batch_tup` immediately, and a worker process will be freed up in the background to begin loading another batch to use on a later iteration. Using the data-loading features of PyTorch can help speed up most projects, because we can overlap data loading and processing with GPU calculation.

11.4 Our first-pass neural network design

The possible design space for a convolutional neural network capable of detecting tumors is effectively infinite. Luckily, considerable effort has been spent over the past decade or so investigating effective models for image recognition. While these have largely focused on 2D images, the general architecture ideas transfer well to 3D, so there are many tested designs that we can use as a starting point. This helps because although our first network architecture is unlikely to be our best option, right now we are only aiming for "good enough to get us going."

We will base the network design on what we used in chapter 8. We will have to update the model somewhat because our input data is 3D, and we will add some complicating details, but the overall structure shown in figure 11.5 should feel familiar. Similarly, the work we do for this project will be a good base for your future projects, although the further you get from classification or segmentation projects, the more you'll have to adapt this base to fit. Let's dissect this architecture, starting with the four repeated blocks that make up the bulk of the network.

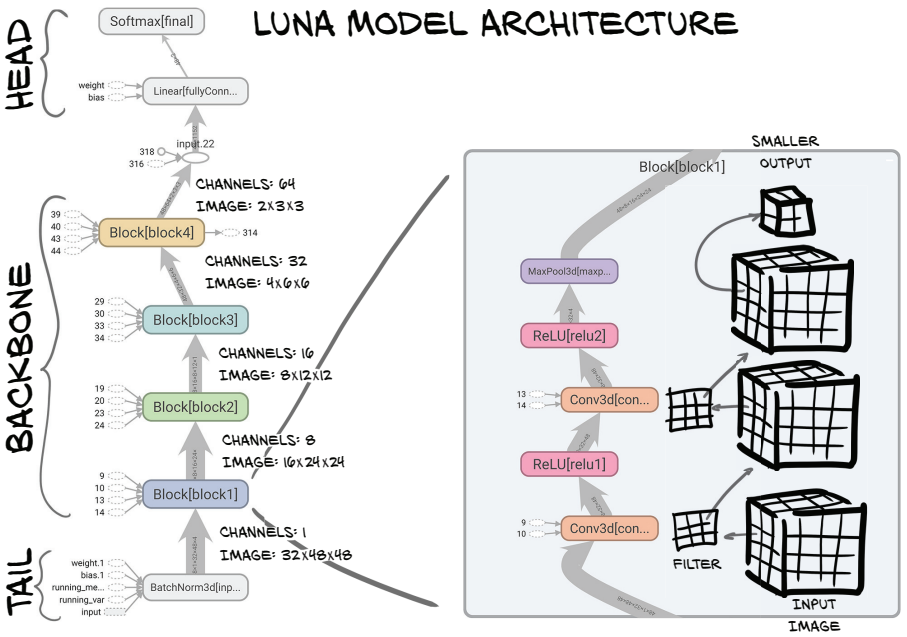

Figure 11.5 The architecture of the `LunaModel` class consisting of a batch-normalization tail, a four-block backbone, and a head comprised of a linear layer followed by softmax

11.4.1 The core convolutions

Classification models often have a structure that consists of a tail, a backbone (or body), and a head. The *tail* is the first few layers that process the input to the network. These early layers often have a different structure or organization than the rest of the network, as they must adapt the input to the form expected by the backbone. Here we use a simple batch normalization layer, though often the tail contains convolutional layers as well. Such convolutional layers are often used to aggressively downsample the size of the image; since our image size is already small, we don't need to do that here.

Next, the *backbone* of the network typically contains the bulk of the layers, which are usually arranged in series of *blocks*. Each block has the same (or at least a similar) set of layers, though often the size of the expected input and the number of filters changes from block to block. We will use a block that consists of two 3 × 3 convolutions, each followed by an activation, with a max-pooling operation at the end of the block. We can see this in the expanded view of figure 11.5 labeled `Block[block1]`. Here's what the implementation of the block looks like in code.

Listing 11.6 model.py:67, `class LunaBlock`

```
class LunaBlock(nn.Module):
  def __init__(self, in_channels, conv_channels):
    super().__init__()
```

```
    self.conv1 = nn.Conv3d(
      in_channels, conv_channels, kernel_size=3, padding=1, bias=True,
    )
    self.relu1 = nn.ReLU(inplace=True)  1((CO5-1))
     self.conv2 = nn.Conv3d(
      conv_channels, conv_channels, kernel_size=3, padding=1, bias=True,
    )
    self.relu2 = nn.ReLU(inplace=True)

    self.maxpool = nn.MaxPool3d(2, 2)

  def forward(self, input_batch):
    block_out = self.conv1(input_batch)
    block_out = self.relu1(block_out)
    block_out = self.conv2(block_out)
    block_out = self.relu2(block_out)

    return self.maxpool(block_out)
```

These could be implemented as calls to the functional API instead.

Finally, the *head* of the network takes the output from the backbone and converts it into the desired output form. For convolutional networks, this often involves flattening the intermediate output and passing it to a fully connected layer. For some networks, it makes sense to also include a second fully connected layer, although that is usually more appropriate for classification problems in which the imaged objects have more structure (think about cars versus trucks having wheels, lights, grill, doors, and so on) and for projects with a large number of classes. Since we are only doing binary classification, and we don't seem to need the additional complexity, we have only a single flattening layer.

Using a structure like this can be a good first building block for a convolutional network. There are more complicated designs out there, but for many projects they're overkill in terms of both implementation complexity and computational demands. It's a good idea to start simple and add complexity only when there's a demonstrable need for it.

We can see the convolutions of our block represented in 2D in figure 11.6. Since this is a small portion of a larger image, we ignore padding here. (Note that the ReLU activation function is not shown, as applying it does not change the image sizes.)

Let's walk through the information flow between our input voxels and a single voxel of output. We want to have a strong sense of how our output will respond when the inputs change. It might be a good idea to review chapter 8, particularly sections 8.1 through 8.3, just to make sure you're 100% solid on the basic mechanics of convolutions.

We're using $3 \times 3 \times 3$ convolutions in our block. A single $3 \times 3 \times 3$ convolution has a receptive field of $3 \times 3 \times 3$, which is almost tautological. Twenty-seven voxels are fed in, and one comes out.

It gets interesting when we use two $3 \times 3 \times 3$ convolutions stacked back to back. Stacking convolutional layers allows the final output voxel (or pixel) to be influenced by an input further away than the size of the convolutional kernel suggests. If that output

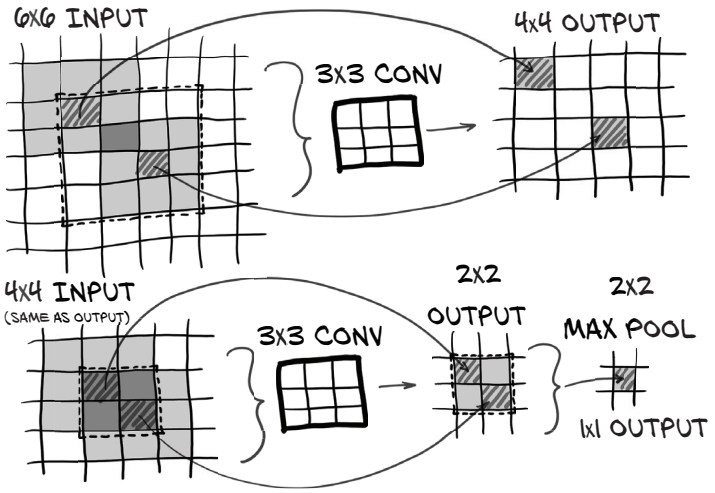

Figure 11.6 The convolutional architecture of a `LunaModel` block consisting of two 3 × 3 convolutions followed by a max pool. The final pixel has a receptive field of 6 × 6.

voxel is fed into another 3 × 3 × 3 kernel as one of the edge voxels, then some of the inputs to the first layer will be outside of the 3 × 3 × 3 area of input to the second. The final output of those two stacked layers has an *effective receptive field* of 5 × 5 × 5. That means that when taken together, the stacked layers act as similar to a single convolutional layer with a larger size.

Put another way, each 3 × 3 × 3 convolutional layer adds an additional one-voxel-per-edge border to the receptive field. We can see this if we trace the arrows in figure 11.6 backward; our 2 × 2 output has a receptive field of 4 × 4, which in turn has a receptive field of 6 × 6. Two stacked 3 × 3 × 3 layers uses fewer parameters than a full 5 × 5 × 5 convolution would (and so is also faster to compute).

The output of our two stacked convolutions is fed into a 2 × 2 × 2 max pool, which means we're taking a 6 × 6 × 6 effective field, throwing away seven-eighths of the data, and going with the one 5 × 5 × 5 field that produced the largest value.[2] Now, those "discarded" input voxels still have a chance to contribute, since the max pool that's one output voxel over has an overlapping input field, so it's possible they'll influence the final output that way.

Note that while we show the receptive field shrinking with each convolutional layer, we're using *padded* convolutions, which add a virtual one-pixel border around the image. Doing so keeps our input and output image sizes the same.

The `nn.ReLU` layers are the same as the ones we looked at in chapter 6. Outputs greater than 0.0 will be left unchanged, and outputs less than 0.0 will be clamped to zero.

This block will be repeated multiple times to form our model's backbone.

[2] Remember that we're actually working in 3D, despite the 2D figure.

11.4.2 The full model

Let's take a look at the full model implementation. We'll skip the block definition, since we just saw that in listing 11.6.

Listing 11.7 model.py:13, `class LunaModel`

```
class LunaModel(nn.Module):
  def __init__(self, in_channels=1, conv_channels=8):
    super().__init__()

    self.tail_batchnorm = nn.BatchNorm3d(1)    <-- Tail

    self.block1 = LunaBlock(in_channels, conv_channels)
    self.block2 = LunaBlock(conv_channels, conv_channels * 2)
    self.block3 = LunaBlock(conv_channels * 2, conv_channels * 4)
    self.block4 = LunaBlock(conv_channels * 4, conv_channels * 8)

    self.head_linear = nn.Linear(1152, 2)
    self.head_softmax = nn.Softmax(dim=1)
```

Backbone

Head

Here, our tail is relatively simple. We are going to normalize our input using `nn.BatchNorm3d`, which, as we saw in chapter 8, will shift and scale our input so that it has a mean of 0 and a standard deviation of 1. Thus, the somewhat odd Hounsfield unit (HU) scale that our input is in won't really be visible to the rest of the network. This is a somewhat arbitrary choice; we know what our input units are, and we know the expected values of the relevant tissues, so we could probably implement a fixed normalization scheme pretty easily. It's not clear which approach would be better.[3]

Our backbone is four repeated blocks, with the block implementation pulled out into the separate `nn.Module` subclass we saw earlier in listing 11.6. Since each block ends with a $2 \times 2 \times 2$ max-pool operation, after 4 layers we will have decreased the resolution of the image 16 times in each dimension. Recall from chapter 10 that our data is returned in chunks that are $32 \times 48 \times 48$, which will become $2 \times 3 \times 3$ by the end of the backbone.

Finally, our tail is just a fully connected layer followed by a call to `nn.Softmax`. Softmax is a useful function for single-label classification tasks and has a few nice properties: it bounds the output between 0 and 1, it's relatively insensitive to the absolute range of the inputs (only the *relative* values of the inputs matter), and it allows our model to express the degree of certainty it has in an answer.

The function itself is relatively simple. Every value from the input is used to exponentiate `e`, and the resulting series of values is then divided by the sum of all the results of exponentiation. Here's what it looks like implemented in a simple fashion as a nonoptimized softmax implementation in pure Python:

```
>>> logits = [1, -2, 3]
>>> exp = [e ** x for x in logits]
>>> exp
```

[3] Which is why there's an exercise to experiment with both in the next chapter!

```
[2.718, 0.135, 20.086]

>>> softmax = [x / sum(exp) for x in exp]
>>> softmax
[0.118, 0.006, 0.876]
```

Of course, we use the PyTorch version of `nn.Softmax` for our model, as it natively understands batches and tensors and will perform autograd quickly and as expected.

COMPLICATION: CONVERTING FROM CONVOLUTION TO LINEAR

Continuing on with our model definition, we come to a complication. We can't just feed the output of `self.block4` into a fully connected layer, since that output is a per-sample $2 \times 3 \times 3$ image with 64 channels, and fully connected layers expect a 1D vector as input (well, technically they expect a *batch* of 1D vectors, which is a 2D array, but the mismatch remains either way). Let's take a look at the `forward` method.

Listing 11.8 model.py:50, `LunaModel.forward`

```
def forward(self, input_batch):
  bn_output = self.tail_batchnorm(input_batch)

  block_out = self.block1(bn_output)
  block_out = self.block2(block_out)
  block_out = self.block3(block_out)
  block_out = self.block4(block_out)

  conv_flat = block_out.view(
    block_out.size(0),      <-- The batch size
    -1,
  )
  linear_output = self.head_linear(conv_flat)

  return linear_output, self.head_softmax(linear_output)
```

Note that before we pass data into a fully connected layer, we must flatten it using the `view` function. Since that operation is stateless (it has no parameters that govern its behavior), we can simply perform the operation in the `forward` function. This is somewhat similar to the functional interfaces we discussed in chapter 8. Almost every model that uses convolution and produces classifications, regressions, or other non-image outputs will have a similar component in the head of the network.

For the return value of the `forward` method, we return both the raw *logits* and the softmax-produced probabilities. We first hinted at logits in section 7.2.6: they are the numerical values produced by the network prior to being normalized into probabilities by the softmax layer. That might sound a bit complicated, but logits are really just the raw input to the softmax layer. They can have any real-valued input, and the softmax will squash them to the range 0–1.

We'll use the logits when we calculate the `nn.CrossEntropyLoss` during training,[4] and we'll use the probabilities for when we want to actually classify the samples. This kind of slight difference between what's used for training and what's used in production is fairly common, especially when the difference between the two outputs is a simple, stateless function like softmax.

Initialization

Finally, let's talk about initializing our network's parameters. In order to get well-behaved performance out of our model, the network's weights, biases, and other parameters need to exhibit certain properties. Let's imagine a degenerate case, where all of the network's weights are greater than 1 (and we do not have residual connections). In that case, repeated multiplication by those weights would result in layer outputs that became very large as data flowed through the layers of the network. Similarly, weights less than 1 would cause all layer outputs to become smaller and vanish. Similar considerations apply to the gradients in the backward pass.

Many normalization techniques can be used to keep layer outputs well behaved, but one of the simplest is to just make sure the network's weights are initialized such that intermediate values and gradients become neither unreasonably small nor unreasonably large. As we discussed in chapter 8, PyTorch does not help us as much as it should here, so we need to do some initialization ourselves. We can treat the following `_init_weights` function as boilerplate, as the exact details aren't particularly important.

Listing 11.9 model.py:30, `LunaModel._init_weights`

```
def _init_weights(self):
  for m in self.modules():
    if type(m) in {
      nn.Linear,
      nn.Conv3d,
    }:
      nn.init.kaiming_normal_(
        m.weight.data, a=0, mode='fan_out', nonlinearity='relu',
      )
      if m.bias is not None:
        fan_in, fan_out = \
          nn.init._calculate_fan_in_and_fan_out(m.weight.data)
        bound = 1 / math.sqrt(fan_out)
        nn.init.normal_(m.bias, -bound, bound)
```

11.5 Training and validating the model

Now it's time to take the various pieces we've been working with and assemble them into something we can actually execute. This training loop should be familiar—we saw loops like figure 11.7 in chapter 5.

[4] There are numerical stability benefits for doing so. Propagating gradients accurately through an exponential calculated using 32-bit floating-point numbers can be problematic.

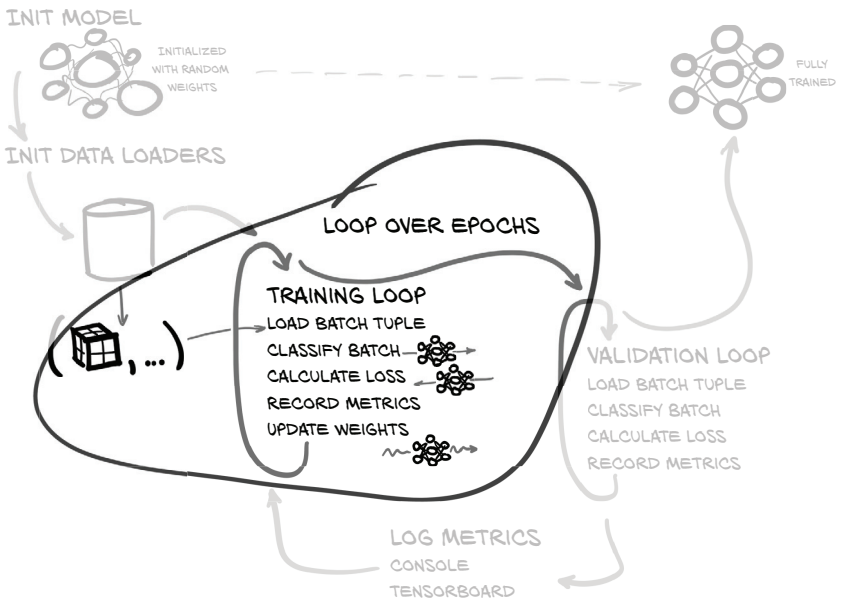

Figure 11.7 The training and validation script we will implement in this chapter, with a focus on the nested loops over each epoch and batches in the epoch

The code is relatively compact (the `doTraining` function is only 12 statements; it's longer here due to line-length limitations).

Listing 11.10 training.py:137, `LunaTrainingApp.main`

```
def main(self):
  # ... line 143
  for epoch_ndx in range(1, self.cli_args.epochs + 1):
    trnMetrics_t = self.doTraining(epoch_ndx, train_dl)
    self.logMetrics(epoch_ndx, 'trn', trnMetrics_t)

# ... line 165
def doTraining(self, epoch_ndx, train_dl):
  self.model.train()
  trnMetrics_g = torch.zeros(
    METRICS_SIZE,
    len(train_dl.dataset),
    device=self.device,
  )

  batch_iter = enumerateWithEstimate(
    train_dl,
    "E{} Training".format(epoch_ndx),
    start_ndx=train_dl.num_workers,
  )
  for batch_ndx, batch_tup in batch_iter:
    self.optimizer.zero_grad()
```

Initializes an empty metrics array

Sets up our batch looping with time estimate

Frees any leftover gradient tensors

```
        loss_var = self.computeBatchLoss(
          batch_ndx,
          batch_tup,
          train_dl.batch_size,
          trnMetrics_g
        )

        loss_var.backward()
        self.optimizer.step()

    self.totalTrainingSamples_count += len(train_dl.dataset)

    return trnMetrics_g.to('cpu')
```

We'll discuss this method in detail in the next section.

Actually updates the model weights

The main differences that we see from the training loops in earlier chapters are as follows:

- The `trnMetrics_g` tensor collects detailed per-class metrics during training. For larger projects like ours, this kind of insight can be very nice to have.
- We don't directly iterate over the `train_dl` data loader. We use `enumerateWithEstimate` to provide an estimated time of completion. This isn't crucial; it's just a stylistic choice.
- The actual loss computation is pushed into the `computeBatchLoss` method. Again, this isn't strictly necessary, but code reuse is typically a plus.

We'll discuss why we've wrapped `enumerate` with additional functionality in section 11.7.2; for now, assume it's the same as `enumerate(train_dl)`.

The purpose of the `trnMetrics_g` tensor is to transport information about how the model is behaving on a per-sample basis from the `computeBatchLoss` function to the `logMetrics` function. Let's take a look at `computeBatchLoss` next. We'll cover `logMetrics` after we're done with the rest of the main training loop.

11.5.1 The computeBatchLoss function

The `computeBatchLoss` function is called by both the training and validation loops. As the name suggests, it computes the loss over a batch of samples. In addition, the function also computes and records per-sample information about the output the model is producing. This lets us compute things like the percentage of correct answers per class, which allows us to hone in on areas where our model is having difficulty.

Of course, the function's core functionality is around feeding the batch into the model and computing the per-batch loss. We're using `CrossEntropyLoss` (https://pytorch.org/docs/stable/nn.html#torch.nn.CrossEntropyLoss), just like in chapter 7. Unpacking the batch tuple, moving the tensors to the GPU, and invoking the model should all feel familiar after that earlier training work.

Listing 11.11 training.py:225, `.computeBatchLoss`

```
def computeBatchLoss(self, batch_ndx, batch_tup, batch_size, metrics_g):
  input_t, label_t, _series_list, _center_list = batch_tup

  input_g = input_t.to(self.device, non_blocking=True)
  label_g = label_t.to(self.device, non_blocking=True)

  logits_g, probability_g = self.model(input_g)

  loss_func = nn.CrossEntropyLoss(reduction='none')
  loss_g = loss_func(
    logits_g,
    label_g[:,1],
  )
  # ... line 238
  return loss_g.mean()
```

reduction='none' gives the loss per sample.

Index of the one-hot-encoded class

Recombines the loss per sample into a single value

Here we are *not* using the default behavior to get a loss value averaged over the batch. Instead, we get a tensor of loss values, one per sample. This lets us track the individual losses, which means we can aggregate them as we wish (per class, for example). We'll see that in action in just a moment. For now, we'll return the mean of those per-sample losses, which is equivalent to the batch loss. In situations where you don't want to keep statistics per sample, using the loss averaged over the batch is perfectly fine. Whether that's the case is highly dependent on your project and goals.

Once that's done, we've fulfilled our obligations to the calling function in terms of what's required to do backpropagation and weight updates. Before we do that, however, we also want to record our per-sample stats for posterity (and later analysis). We'll use the `metrics_g` parameter passed in to accomplish this.

Listing 11.12 training.py:26

```
METRICS_LABEL_NDX=0
METRICS_PRED_NDX=1
METRICS_LOSS_NDX=2
METRICS_SIZE = 3

  # ... line 225
  def computeBatchLoss(self, batch_ndx, batch_tup, batch_size, metrics_g):
    # ... line 238
    start_ndx = batch_ndx * batch_size
    end_ndx = start_ndx + label_t.size(0)

    metrics_g[METRICS_LABEL_NDX, start_ndx:end_ndx] = \
      label_g[:,1].detach()
    metrics_g[METRICS_PRED_NDX, start_ndx:end_ndx] = \
      probability_g[:,1].detach()
    metrics_g[METRICS_LOSS_NDX, start_ndx:end_ndx] = \
      loss_g.detach()

    return loss_g.mean()
```

These named array indexes are declared at module-level scope.

We use detach since none of our metrics need to hold on to gradients.

Again, this is the loss over the entire batch.

By recording the label, prediction, and loss for each and every training (and later, validation) sample, we have a wealth of detailed information we can use to investigate the behavior of our model. For now, we're going to focus on compiling per-class statistics, but we could easily use this information to find the sample that is classified the most wrongly and start to investigate why. Again, for some projects, this kind of information will be less interesting, but it's good to remember that you have these kinds of options available.

11.5.2 The validation loop is similar

The validation loop in figure 11.8 looks very similar to training but is somewhat simplified. The key difference is that validation is read-only. Specifically, the loss value returned is not used, and the weights are not updated.

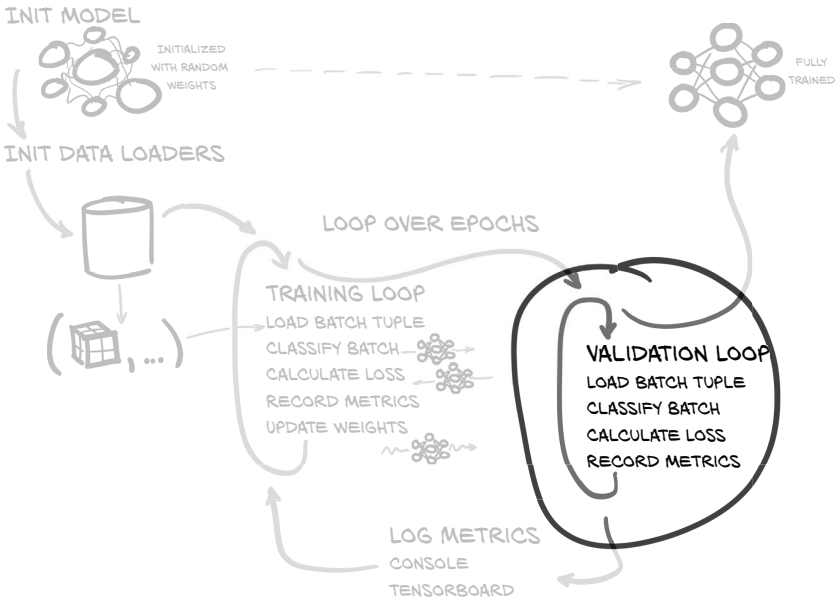

Figure 11.8 The training and validation script we will implement in this chapter, with a focus on the per-epoch validation loop

Nothing about the model should have changed between the start and end of the function call. In addition, it's quite a bit faster due to the `with torch.no_grad()` context manager explicitly informing PyTorch that no gradients need to be computed.

Listing 11.13 training.py:137, `LunaTrainingApp.main`

```
def main(self):
  for epoch_ndx in range(1, self.cli_args.epochs + 1):
    # ... line 157
    valMetrics_t = self.doValidation(epoch_ndx, val_dl)
```

```
    self.logMetrics(epoch_ndx, 'val', valMetrics_t)

# ... line 203
def doValidation(self, epoch_ndx, val_dl):
  with torch.no_grad():
    self.model.eval()                ◁— Turns off training-time behavior
    valMetrics_g = torch.zeros(
      METRICS_SIZE,
      len(val_dl.dataset),
      device=self.device,
    )

    batch_iter = enumerateWithEstimate(
      val_dl,
      "E{} Validation ".format(epoch_ndx),
      start_ndx=val_dl.num_workers,
    )
    for batch_ndx, batch_tup in batch_iter:
      self.computeBatchLoss(
        batch_ndx, batch_tup, val_dl.batch_size, valMetrics_g)

  return valMetrics_g.to('cpu')
```

Without needing to update network weights (recall that doing so would violate the entire premise of the validation set; something we never want to do!), we don't need to use the loss returned from `computeBatchLoss`, nor do we need to reference the optimizer. All that's left inside the loop is the call to `computeBatchLoss`. Note that we are still collecting metrics in `valMetrics_g` as a side effect of the call, even though we aren't using the overall per-batch loss returned by `computeBatchLoss` for anything.

11.6 Outputting performance metrics

The last thing we do per epoch is log our performance metrics for this epoch. As shown in figure 11.9, once we've logged metrics, we return to the training loop for the next epoch of training. Logging results and progress as we go is important, since if training goes off the rails ("does not converge" in the parlance of deep learning), we want to notice this is happening and stop spending time training a model that's not working out. In less catastrophic cases, it's good to be able to keep an eye on how your model behaves.

Earlier, we were collecting results in `trnMetrics_g` and `valMetrics_g` for logging progress per epoch. Each of these two tensors now contains everything we need to compute our percent correct and average loss per class for our training and validation runs. Doing this per epoch is a common choice, though somewhat arbitrary. In future chapters, we'll see how to manipulate the size of our epochs such that we get feedback about training progress at a reasonable rate.

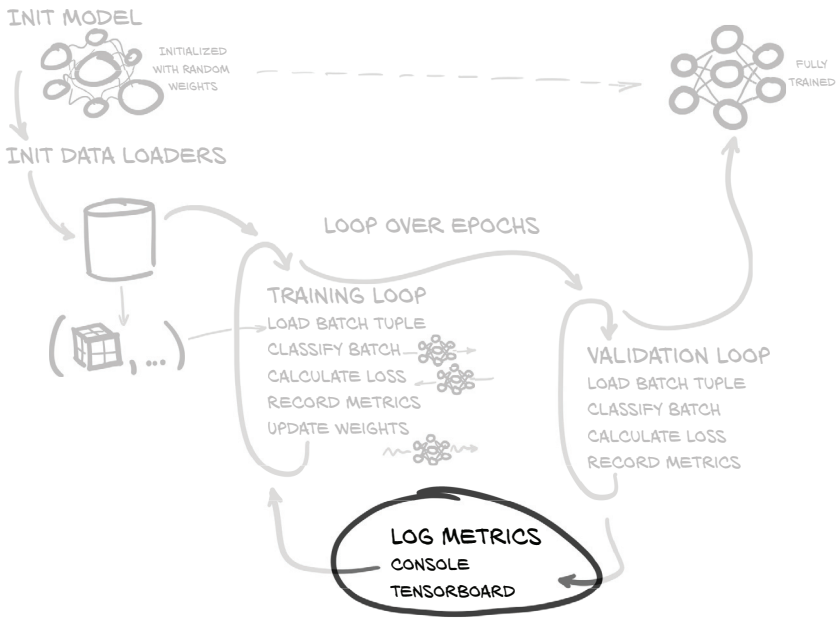

Figure 11.9 The training and validation script we will implement in this chapter, with a focus on the metrics logging at the end of each epoch

11.6.1 The logMetrics function

Let's talk about the high-level structure of the `logMetrics` function. The signature looks like this.

Listing 11.14 training.py:251, `LunaTrainingApp.logMetrics`

```
def logMetrics(
    self,
    epoch_ndx,
    mode_str,
    metrics_t,
    classificationThreshold=0.5,
):
```

We use `epoch_ndx` purely for display while logging our results. The `mode_str` argument tells us whether the metrics are for training or validation.

We consume either `trnMetrics_t` or `valMetrics_t`, which is passed in as the `metrics_t` parameter. Recall that both of those inputs are tensors of floating-point values that we filled with data during `computeBatchLoss` and then transferred back to the CPU right before we returned them from `doTraining` and `doValidation`. Both tensors have three rows and as many columns as we have samples (training samples or validation samples, depending). As a reminder, those three rows correspond to the following constants.

Listing 11.15 training.py:26

```
METRICS_LABEL_NDX=0
METRICS_PRED_NDX=1
METRICS_LOSS_NDX=2
METRICS_SIZE = 3
```

These are declared at module-level scope.

Tensor masking and Boolean indexing

Masked tensors are a common usage pattern that might be opaque if you have not encountered them before. You may be familiar with the NumPy concept called *masked arrays*; tensor and array masks behave the same way.

If you aren't familiar with masked arrays, an excellent page in the NumPy documentation (http://mng.bz/XPra) describes the behavior well. PyTorch purposely uses the same syntax and semantics as NumPy.

CONSTRUCTING MASKS

Next, we're going to construct masks that will let us limit our metrics to only the nodule or non-nodule (aka positive or negative) samples. We will also count the total samples per class, as well as the number of samples we classified correctly.

Listing 11.16 training.py:264, `LunaTrainingApp.logMetrics`

```
negLabel_mask = metrics_t[METRICS_LABEL_NDX] <= classificationThreshold
negPred_mask = metrics_t[METRICS_PRED_NDX] <= classificationThreshold

posLabel_mask = ~negLabel_mask
posPred_mask = ~negPred_mask
```

While we don't `assert` it here, we know that all of the values stored in `metrics_t[METRICS_LABEL_NDX]` belong to the set `{0.0, 1.0}` since we know that our nodule status labels are simply `True` or `False`. By comparing to `classificationThreshold`, which defaults to 0.5, we get an array of binary values where a `True` value corresponds to a non-nodule (aka negative) label for the sample in question.

We do a similar comparison to create the `negPred_mask`, but we must remember that the `METRICS_PRED_NDX` values are the positive predictions produced by our model and can be any floating-point value between 0.0 and 1.0, inclusive. That doesn't change our comparison, but it does mean the actual value can be close to 0.5. The positive masks are simply the inverse of the negative masks.

NOTE While other projects can utilize similar approaches, it's important to realize that we're taking some shortcuts that are allowed because this is a binary classification problem. If your next project has more than two classes or has samples that belong to multiple classes at the same time, you'll have to use more complicated logic to build similar masks.

Next, we use those masks to compute some per-label statistics and store them in a dictionary, `metrics_dict`.

Listing 11.17 training.py:270, `LunaTrainingApp.logMetrics`

```
neg_count = int(negLabel_mask.sum())
pos_count = int(posLabel_mask.sum())

neg_correct = int((negLabel_mask & negPred_mask).sum())
pos_correct = int((posLabel_mask & posPred_mask).sum())

metrics_dict = {}
metrics_dict['loss/all'] = \
  metrics_t[METRICS_LOSS_NDX].mean()
metrics_dict['loss/neg'] = \
  metrics_t[METRICS_LOSS_NDX, negLabel_mask].mean()
metrics_dict['loss/pos'] = \
  metrics_t[METRICS_LOSS_NDX, posLabel_mask].mean()

metrics_dict['correct/all'] = (pos_correct + neg_correct) \
  / np.float32(metrics_t.shape[1]) * 100
metrics_dict['correct/neg'] = neg_correct / np.float32(neg_count) * 100
metrics_dict['correct/pos'] = pos_correct / np.float32(pos_count) * 100
```

Converts to a normal Python integer

Avoids integer division by converting to np.float32

First we compute the average loss over the entire epoch. Since the loss is the single metric that is being minimized during training, we always want to be able to keep track of it. Then we limit the loss averaging to only those samples with a negative label using the `negLabel_mask` we just made. We do the same with the positive loss. Computing a per-class loss like this can be useful if one class is persistently harder to classify than another, since that knowledge can help drive investigation and improvements.

We'll close out the calculations with determining the fraction of samples we classified correctly, as well as the fraction correct from each label. Since we will display these numbers as percentages in a moment, we also multiply the values by 100. Similar to the loss, we can use these numbers to help guide our efforts when making improvements. After the calculations, we then log our results with three calls to `log.info`.

Listing 11.18 training.py:289, `LunaTrainingApp.logMetrics`

```
log.info(
  ("E{} {:8} {loss/all:.4f} loss, "
     + "{correct/all:-5.1f}% correct, "
  ).format(
    epoch_ndx,
    mode_str,
    **metrics_dict,
  )
)
log.info(
  ("E{} {:8} {loss/neg:.4f} loss, "
     + "{correct/neg:-5.1f}% correct ({neg_correct:} of {neg_count:})"
```

```
  ).format(
    epoch_ndx,
    mode_str + '_neg',
    neg_correct=neg_correct,
    neg_count=neg_count,
    **metrics_dict,
  )
)
log.info(
  # ... line 319
)
```

The 'pos' logging is similar to the 'neg' logging earlier.

The first log has values computed from all of our samples and is tagged `/all`, while the negative (non-nodule) and positive (nodule) values are tagged `/neg` and `/pos`, respectively. We don't show the third logging statement for positive values here; it's identical to the second except for swapping *neg* for *pos* in all cases.

11.7 Running the training script

Now that we've completed the core of the training.py script, we'll actually start running it. This will initialize and train our model and print statistics about how well the training is going. The idea is to get this kicked off to run in the background while we're covering the model implementation in detail. Hopefully we'll have results to look at once we're done.

We're running this script from the main code directory; it should have subdirectories called p2ch11, util, and so on. The `python` environment used should have all the libraries listed in requirements.txt installed. Once those libraries are ready, we can run:

This is the command line for Linux/Bash. Windows users will probably need to invoke Python differently, depending on the install method used.

```
$ python -m p2ch11.training
Starting LunaTrainingApp,
    Namespace(batch_size=256, channels=8, epochs=20, layers=3, num_workers=8)
<p2ch11.dsets.LunaDataset object at 0x7fa53a128710>: 495958 training samples
<p2ch11.dsets.LunaDataset object at 0x7fa537325198>: 55107 validation samples
Epoch 1 of 20, 1938/216 batches of size 256
E1 Training ----/1938, starting
E1 Training   16/1938, done at 2018-02-28 20:52:54, 0:02:57
...
```

As a reminder, we also provide a Jupyter Notebook that contains invocations of the training application.

Listing 11.19 code/p2_run_everything.ipynb

```
# In[5]:
run('p2ch11.prepcache.LunaPrepCacheApp')

# In[6]:
run('p2ch11.training.LunaTrainingApp', '--epochs=1')
```

If the first epoch seems to be taking a very long time (more than 10 or 20 minutes), it might be related to needing to prepare the cached data required by `LunaDataset`. See section 10.5.1 for details about the caching. The exercises for chapter 10 included writing a script to pre-stuff the cache in an efficient manner. We also provide the prepcache.py file to do the same thing; it can be invoked with `python -m p2ch11.prepcache`. Since we repeat our dsets.py files per chapter, the caching will need to be repeated for every chapter. This is somewhat space and time inefficient, but it means we can keep the code for each chapter much more well contained. For your future projects, we recommend reusing your cache more heavily.

Once training is underway, we want to make sure we're using the computing resources at hand the way we expect. An easy way to tell if the bottleneck is data loading or computation is to wait a few moments after the script starts to train (look for output like `E1 Training 16/7750, done at…`) and then check both `top` and `nvidia-smi`:

- If the eight Python worker processes are consuming >80% CPU, then the cache probably needs to be prepared (we know this here because the authors have made sure there aren't CPU bottlenecks in this project's implementation; this won't be generally true).
- If `nvidia-smi` reports that `GPU-Util` is >80%, then you're saturating your GPU. We'll discuss some strategies for efficient waiting in section 11.7.2.

The intent is that the GPU is saturated; we want to use as much of that computing power as we can to complete epochs quickly. A single NVIDIA GTX 1080 Ti should complete an epoch in under 15 minutes. Since our model is relatively simple, it doesn't take a lot of CPU preprocessing for the CPU to be the bottleneck. When working with models with greater depth (or more needed calculations in general), processing each batch will take longer, which will increase the amount of CPU processing we can do before the GPU runs out of work before the next batch of input is ready.

11.7.1 Needed data for training

If the number of samples is less than 495,958 for training or 55,107 for validation, it might make sense to do some sanity checking to be sure the full data is present and accounted for. For your future projects, make sure your dataset returns the number of samples that you expect.

First, let's take a look at the basic directory structure of our data-unversioned/part2/luna directory:

```
$ ls -1p data-unversioned/part2/luna/
subset0/
subset1/
...
subset9/
```

Next, let's make sure we have one .mhd file and one .raw file for each series UID

```
$ ls -1p data-unversioned/part2/luna/subset0/
1.3.6.1.4.1.14519.5.2.1.6279.6001.105756658031515062000744821260.mhd
1.3.6.1.4.1.14519.5.2.1.6279.6001.105756658031515062000744821260.raw
1.3.6.1.4.1.14519.5.2.1.6279.6001.108197895896446896160048741492.mhd
1.3.6.1.4.1.14519.5.2.1.6279.6001.108197895896446896160048741492.raw
...
```

and that we have the overall correct number of files:

```
$ ls -1 data-unversioned/part2/luna/subset?/* | wc -l
1776
$ ls -1 data-unversioned/part2/luna/subset0/* | wc -l
178
...
$ ls -1 data-unversioned/part2/luna/subset9/* | wc -l
176
```

If all of these seem right but things still aren't working, ask on Manning LiveBook (https://livebook.manning.com/book/deep-learning-with-pytorch/chapter-11) and hopefully someone can help get things sorted out.

11.7.2 Interlude: The enumerateWithEstimate function

Working with deep learning involves a lot of waiting. We're talking about real-world, sitting around, glancing at the clock on the wall, a watched pot never boils (but you could fry an egg on the GPU), straight up *boredom*.

The only thing worse than sitting and staring at a blinking cursor that hasn't moved for over an hour is flooding your screen with this:

```
2020-01-01 10:00:00,056 INFO training batch 1234
2020-01-01 10:00:00,067 INFO training batch 1235
2020-01-01 10:00:00,077 INFO training batch 1236
2020-01-01 10:00:00,087 INFO training batch 1237
...etc...
```

At least the quietly blinking cursor doesn't blow out your scrollback buffer!

Fundamentally, while doing all this waiting, we want to answer the question "Do I have time to go refill my water glass?" along with follow-up questions about having time to

- Brew a cup of coffee
- Grab dinner
- Grab dinner in Paris[5]

To answer these pressing questions, we're going to use our `enumerateWithEstimate` function. Usage looks like the following:

[5] If getting dinner in France doesn't involve an airport, feel free to substitute "Paris, Texas" to make the joke work; https://en.wikipedia.org/wiki/Paris_(disambiguation).

```
>>> for i, _ in enumerateWithEstimate(list(range(234)), "sleeping"):
...   time.sleep(random.random())
...
11:12:41,892 WARNING sleeping ----/234, starting
11:12:44,542 WARNING sleeping    4/234, done at 2020-01-01 11:15:16, 0:02:35
11:12:46,599 WARNING sleeping    8/234, done at 2020-01-01 11:14:59, 0:02:17
11:12:49,534 WARNING sleeping   16/234, done at 2020-01-01 11:14:33, 0:01:51
11:12:58,219 WARNING sleeping   32/234, done at 2020-01-01 11:14:41, 0:01:59
11:13:15,216 WARNING sleeping   64/234, done at 2020-01-01 11:14:43, 0:02:01
11:13:44,233 WARNING sleeping  128/234, done at 2020-01-01 11:14:35, 0:01:53
11:14:40,083 WARNING sleeping ----/234, done at 2020-01-01 11:14:40
>>>
```

That's 8 lines of output for over 200 iterations lasting about 2 minutes. Even given the wide variance of `random.random()`, the function had a pretty decent estimate after 16 iterations (in less than 10 seconds). For loop bodies with more constant timing, the estimates stabilize even more quickly.

In terms of behavior, `enumerateWithEstimate` is almost identical to the standard `enumerate` (the differences are things like the fact that our function returns a generator, whereas `enumerate` returns a specialized `<enumerate object at 0x…>`).

Listing 11.20 util.py:143, `def enumerateWithEstimate`

```
def enumerateWithEstimate(
    iter,
    desc_str,
    start_ndx=0,
    print_ndx=4,
    backoff=None,
    iter_len=None,
):
  for (current_ndx, item) in enumerate(iter):
    yield (current_ndx, item)
```

However, the side effects (logging, specifically) are what make the function interesting. Rather than get lost in the weeds trying to cover every detail of the implementation, if you're interested, you can consult the function docstring (https://github.com/deep-learning-with-pytorch/dlwpt-code/blob/master/util/util.py#L143) to get information about the function parameters and desk-check the implementation.

Deep learning projects can be very time intensive. Knowing when something is expected to finish means you can use your time until then wisely, and it can also clue you in that something isn't working properly (or an approach is unworkable) if the expected time to completion is much larger than expected.

11.8 Evaluating the model: Getting 99.7% correct means we're done, right?

Let's take a look at some (abridged) output from our training script. As a reminder, we've run this with the command line `python -m p2ch11.training`:

```
E1 Training ----/969, starting
...
E1 LunaTrainingApp
E1 trn      2.4576 loss,  99.7% correct
...
E1 val      0.0172 loss,  99.8% correct
...
```

After one epoch of training, both the training and validation set show at least 99.7% correct results. That's an A+! Time for a round of high-fives, or at least a satisfied nod and smile. We just solved cancer! … Right?

Well, no.

Let's take a closer (less-abridged) look at that epoch 1 output:

```
E1 LunaTrainingApp
E1 trn      2.4576 loss,  99.7% correct,
E1 trn_neg  0.1936 loss,  99.9% correct (494289 of 494743)
E1 trn_pos  924.34 loss,   0.2% correct (3 of 1215)
...
E1 val      0.0172 loss,  99.8% correct,
E1 val_neg  0.0025 loss, 100.0% correct (494743 of 494743)
E1 val_pos  5.9768 loss,   0.0% correct (0 of 1215)
```

On the validation set, we're getting non-nodules 100% correct, but the actual nodules are 100% wrong. The network is just classifying everything as not-a-nodule! The value 99.7% just means only approximately 0.3% of the samples are nodules.

After 10 epochs, the situation is only marginally better:

```
E10 LunaTrainingApp
E10 trn      0.0024 loss,  99.8% correct
E10 trn_neg  0.0000 loss, 100.0% correct
E10 trn_pos  0.9915 loss,   0.0% correct
E10 val      0.0025 loss,  99.7% correct
E10 val_neg  0.0000 loss, 100.0% correct
E10 val_pos  0.9929 loss,   0.0% correct
```

The classification output remains the same—none of the nodule (aka positive) samples are correctly identified. It's interesting that we're starting to see some decrease in the `val_pos` loss, however, while not seeing a corresponding increase in the `val_neg` loss. This implies that the network *is* learning something. Unfortunately, it's learning very, very slowly.

Even worse, this particular failure mode is the most dangerous in the real world! We want to avoid the situation where we classify a tumor as an innocuous structure,

because that would not facilitate a patient getting the evaluation and eventual treatment they might need. It's important to understand the consequences for misclassification for all your projects, as that can have a large impact on how you design, train, and evaluate your model. We'll discuss this more in the next chapter.

Before we get to that, however, we need to upgrade our tooling to make the results easier to understand. We're sure you love to squint at columns of numbers as much as anyone, but pictures are worth a thousand words. Let's graph some of these metrics.

11.9 Graphing training metrics with TensorBoard

We're going to use a tool called TensorBoard as a quick and easy way to get our training metrics out of our training loop and into some pretty graphs. This will allow us to follow the *trends* of those metrics, rather than only look at the instantaneous values per epoch. It gets much, much easier to know whether a value is an outlier or just the latest in a trend when you're looking at a visual representation.

"Hey, wait," you might be thinking, "isn't TensorBoard part of the TensorFlow project? What's it doing here in my PyTorch book?"

Well, yes, it is part of another deep learning framework, but our philosophy is "use what works." There's no reason to restrict ourselves by not using a tool just because it's bundled with another project we're not using. Both the PyTorch and TensorBoard devs agree, because they collaborated to add official support for TensorBoard into PyTorch. TensorBoard is great, and it's got some easy-to-use PyTorch APIs that let us hook data from just about anywhere into it for quick and easy display. If you stick with deep learning, you'll probably be seeing (and using) a *lot* of TensorBoard.

In fact, if you've been running the chapter examples, you should already have some data on disk ready and waiting to be displayed. Let's see how to run TensorBoard, and look at what it can show us.

11.9.1 Running TensorBoard

By default, our training script will write metrics data to the runs/ subdirectory. If you list the directory content, you might see something like this during your Bash shell session:

```
$ ls -lA runs/p2ch11/
total 24
drwxrwxr-x 2 elis elis 4096 Sep 15 13:22 2020-01-01_12.55.27-trn-dlwpt/
drwxrwxr-x 2 elis elis 4096 Sep 15 13:22 2020-01-01_12.55.27-val-dlwpt/
drwxrwxr-x 2 elis elis 4096 Sep 15 15:14 2020-01-01_13.31.23-trn-dwlpt/
drwxrwxr-x 2 elis elis 4096 Sep 15 15:14 2020-01-01_13.31.23-val-dwlpt/
```

The single-epoch run from earlier

The more recent 10-epoch training run

To get the `tensorboard` program, install the `tensorflow` (https://pypi.org/project/tensorflow) Python package. Since we're not actually going to use TensorFlow proper, it's fine if you install the default CPU-only package. If you have another version of

TensorBoard installed already, using that is fine too. Either make sure the appropriate directory is on your path, or invoke it with `../path/to/tensorboard --logdir runs/`. It doesn't really matter where you invoke it from, as long as you use the `--logdir` argument to point it at where your data is stored. It's a good idea to segregate your data into separate folders, as TensorBoard can get a bit unwieldy once you get over 10 or 20 experiments. You'll have to decide the best way to do that for each project as you go. Don't be afraid to move data around after the fact if you need to.

Let's start TensorBoard now:

```
$ tensorboard --logdir runs/
2020-01-01 12:13:16.163044: I tensorflow/core/platform/cpu_feature_guard.cc:140]
    Your CPU supports instructions that this TensorFlow binary was not
➥ compiled to use: AVX2 FMA  1((CO17-2))
TensorBoard 1.14.0 at http://localhost:6006/ (Press CTRL+C to quit)
```

These messages might be different or not present for you; that's fine.

Once that's done, you should be able to point your browser at http://localhost:6006 and see the main dashboard.[6] Figure 11.10 shows us what that looks like.

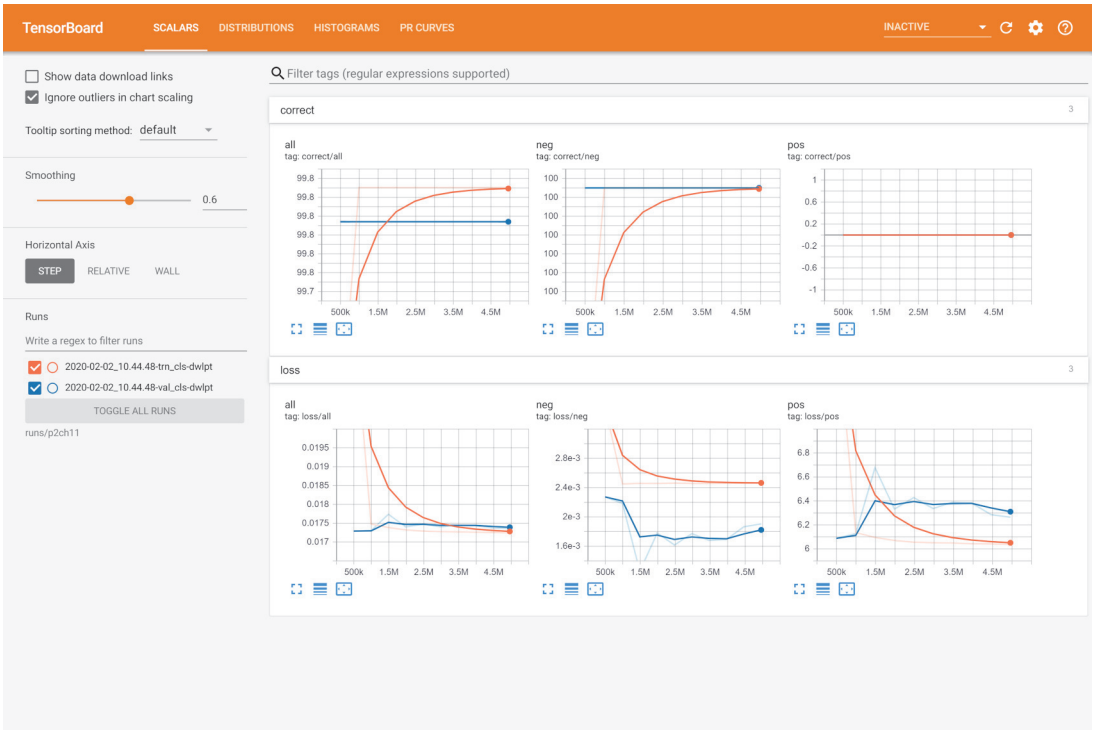

Figure 11.10 The main TensorBoard UI, showing a paired set of training and validation runs

[6] If you're running training on a different computer from your browser, you'll need to replace *localhost* with the appropriate hostname or IP address.

Along the top of the browser window, you should see the orange header. The right side of the header has the typical widgets for settings, a link to the GitHub repository, and the like. We can ignore those for now. The left side of the header has items for the data types we've provided. You should have at least the following:

- Scalars (the default tab)
- Histograms
- Precision-Recall Curves (shown as PR Curves)

You might see Distributions as well as the second UI tab (to the right of Scalars in figure 11.10). We won't use or discuss those here. Make sure you've selected Scalars by clicking it.

On the left is a set of controls for display options, as well as a list of runs that are present. The smoothing option can be useful if you have particularly noisy data; it will calm things down so that you can pick out the overall trend. The original non-smoothed data will still be visible in the background as a faded line in the same color. Figure 11.11 shows this, although it might be difficult to discern when printed in black and white.

Depending on how many times you've run the training script, you might have multiple runs to select from. With too many runs being rendered, the graphs can get overly noisy, so don't hesitate to deselect runs that aren't of interest at the moment.

If you want to permanently remove a run, the data can be deleted from disk while TensorBoard is running. You can do this to get rid of experiments that crashed, had

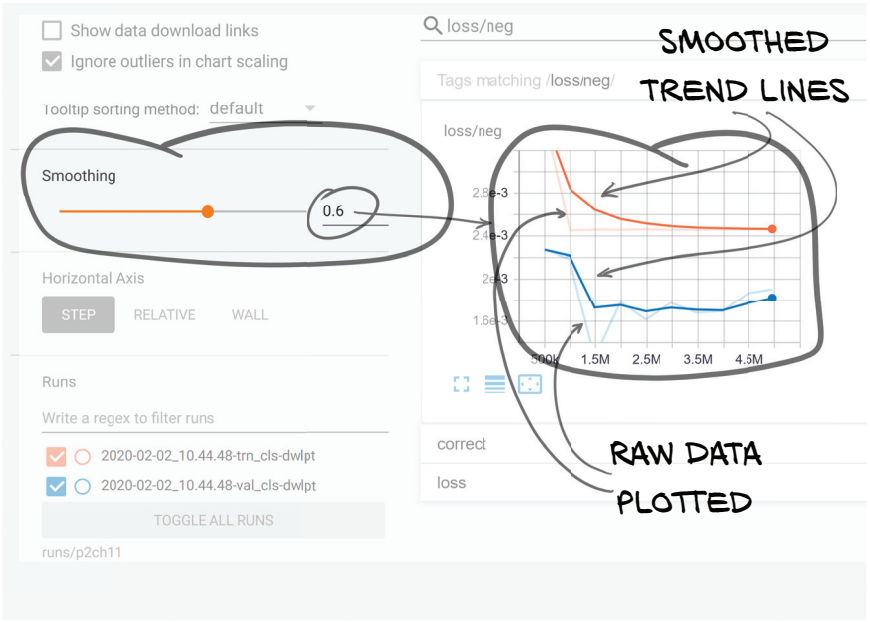

Figure 11.11 The TensorBoard sidebar with Smoothing set to 0.6 and two runs selected for display

bugs, didn't converge, or are so old they're no longer interesting. The number of runs can grow pretty quickly, so it can be helpful to prune it often and to rename runs or move runs that are particularly interesting to a more permanent directory so they don't get deleted by accident. To remove both the `train` and `validation` runs, execute the following (after changing the chapter, date, and time to match the run you want to remove):

```
$ rm -rf runs/p2ch11/2020-01-01_12.02.15_*
```

Keep in mind that removing runs will cause the runs that are later in the list to move up, which will result in them being assigned new colors.

OK, let's get to the point of TensorBoard: the pretty graphs! The main part of the screen should be filled with data from gathering training and validation metrics, as shown in figure 11.12.

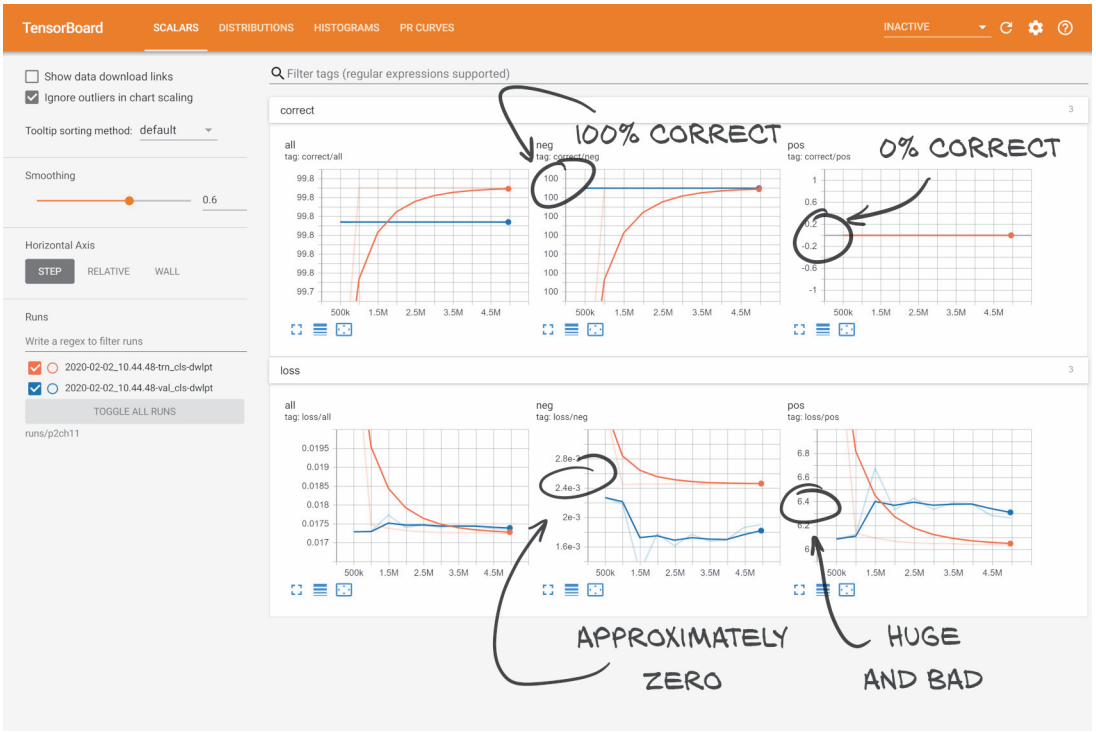

Figure 11.12 The main TensorBoard data display area showing us that our results on actual nodules are downright awful

That's much easier to parse and absorb than `E1 trn_pos 924.34 loss, 0.2% correct (3 of 1215)`! Although we're going to save discussion of what these graphs are telling us for section 11.10, now would be a good time to make sure it's clear what these numbers correspond to from our training program. Take a moment to cross-reference the

numbers you get by mousing over the lines with the numbers spit out by training.py during the same training run. You should see a direct correspondence between the Value column of the tooltip and the values printed during training. Once you're comfortable and confident that you understand exactly what TensorBoard is showing you, let's move on and discuss how to get these numbers to appear in the first place.

11.9.2 Adding TensorBoard support to the metrics logging function

We are going to use the `torch.utils.tensorboard` module to write data in a format that TensorBoard will consume. This will allow us to write metrics for this and any other project quickly and easily. TensorBoard supports a mix of NumPy arrays and PyTorch tensors, but since we don't have any reason to put our data into NumPy arrays, we'll use PyTorch tensors exclusively.

The first thing we need do is to create our `SummaryWriter` objects (which we imported from `torch.utils.tensorboard`). The only parameter we're going to pass in is `log_dir`, which we will initialize to something like `runs/p2ch11/2020-01-01_12.55.27-trn-dlwpt`. We can add a comment argument to our training script to change `dlwpt` to something more informative; use `python -m p2ch11.training --help` for more information.

We create two writers, one each for the training and validation runs. Those writers will be reused for every epoch. When the `SummaryWriter` class gets initialized, it also creates the `log_dir` directories as a side effect. These directories show up in TensorBoard and can clutter the UI with empty runs if the training script crashes before any data gets written, which can be common when you're experimenting with something. To avoid writing too many empty junk runs, we wait to instantiate the `SummaryWriter` objects until we're ready to write data for the first time. This function is called from `logMetrics()`.

Listing 11.21 training.py:127, .initTensorboardWriters

```
def initTensorboardWriters(self):
  if self.trn_writer is None:
    log_dir = os.path.join('runs', self.cli_args.tb_prefix, self.time_str)

    self.trn_writer = SummaryWriter(
      log_dir=log_dir + '-trn_cls-' + self.cli_args.comment)
    self.val_writer = SummaryWriter(
      log_dir=log_dir + '-val_cls-' + self.cli_args.comment)
```

If you recall, the first epoch is kind of a mess, with the early output in the training loop being essentially random. When we save the metrics from that first batch, those random results end up skewing things a bit. Recall from figure 11.11 that TensorBoard has smoothing to remove noise from the trend lines, which helps somewhat.

Another approach could be to skip metrics entirely for the first epoch's training data, although our model trains quickly enough that it's still useful to see the first

epoch's results. Feel free to change this behavior as you see fit; the rest of part 2 will continue with this pattern of including the first, noisy training epoch.

> **TIP** If you end up doing a lot of experiments that result in exceptions or killing the training script relatively quickly, you might be left with a number of junk runs cluttering up your runs/ directory. Don't be afraid to clean those out!

WRITING SCALARS TO TENSORBOARD

Writing scalars is straightforward. We can take the `metrics_dict` we've already constructed and pass in each key/value pair to the `writer.add_scalar` method. The `torch.utils.tensorboard.SummaryWriter` class has the `add_scalar` method (http://mng.bz/RAqj) with the following signature.

Listing 11.22 PyTorch torch/utils/tensorboard/writer.py:267

```
def add_scalar(self, tag, scalar_value, global_step=None, walltime=None):
    # ...
```

The `tag` parameter tells TensorBoard which graph we're adding values to, and the `scalar_value` parameter is our data point's Y-axis value. The `global_step` parameter acts as the X-axis value.

Recall that we updated the `totalTrainingSamples_count` variable inside the `doTraining` function. We'll use `totalTrainingSamples_count` as the X-axis of our TensorBoard plots by passing it in as the `global_step` parameter. Here's what that looks like in our code.

Listing 11.23 training.py:323, `LunaTrainingApp.logMetrics`

```
for key, value in metrics_dict.items():
  writer.add_scalar(key, value, self.totalTrainingSamples_count)
```

Note that the slashes in our key names (such as `'loss/all'`) result in TensorBoard grouping the charts by the substring before the `'/'`.

The documentation suggests that we should be passing in the epoch number as the `global_step` parameter, but that results in some complications. By using the number of training samples presented to the network, we can do things like change the number of samples per epoch and still be able to compare those future graphs to the ones we're creating now. Saying that a model trains in half the number of epochs is meaningless if each epoch takes four times as long! Keep in mind that this might not be standard practice, however; expect to see a variety of values used for the global step.

11.10 Why isn't the model learning to detect nodules?

Our model is clearly learning *something*—the loss trend lines are consistent as epochs increase, and the results are repeatable. There is a disconnect, however, between what the model is learning and what we *want* it to learn. What's going on? Let's use a quick metaphor to illustrate the problem.

Imagine that a professor gives students a final exam consisting of 100 True/False questions. The students have access to previous versions of this professor's tests going back 30 years, and every time there are only *one or two* questions with a True answer. The other 98 or 99 are False, every time.

Assuming that the grades aren't on a curve and instead have a typical scale of 90% correct or better being an A, and so on, it is trivial to get an A+: just mark every question as False! Let's imagine that this year, there is only one True answer. A student like the one on the left in figure 11.13 who mindlessly marked every answer as False would get a 99% on the final but wouldn't really demonstrate that they had learned anything (beyond how to cram from old tests, of course). That's basically what our model is doing right now.

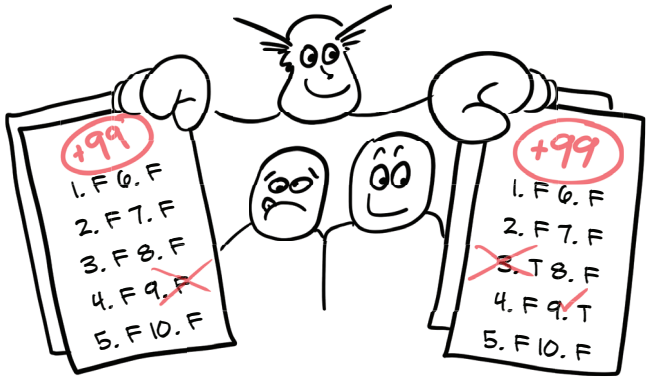

Figure 11.13 A professor giving two students the same grade, despite different levels of knowledge. Question 9 is the only question with an answer of True.

Contrast that with a student like the one on the right who also got 99% of the questions correct, but did so by answering two questions with True. Intuition tells us that the student on the right in figure 11.13 probably has a much better grasp of the material than the all-False student. Finding the one True question while only getting one answer wrong is pretty difficult! Unfortunately, neither our students' grades nor our model's grading scheme reflect this gut feeling.

We have a similar situation, where 99.7% of the answers to "Is this candidate a nodule?" are "Nope." Our model is taking the easy way out and answering False on every question.

Still, if we look back at our model's numbers more closely, the loss on the training and validation sets *is* decreasing! The fact that we're getting any traction at all on the cancer-detection problem should give us hope. It will be the work of the next chapter to realize this potential. We'll start chapter 12 by introducing some new, relevant

terminology, and then we'll come up with a better grading scheme that doesn't lend itself to being gamed quite as easily as what we've done so far.

11.11 Conclusion

We've come a long way this chapter—we now have a model and a training loop, and are able to consume the data we produced in the last chapter. Our metrics are being logged to the console as well as graphed visually.

While our results aren't usable yet, we're actually closer than it might seem. In chapter 12, we will improve the metrics we're using to track our progress, and use them to inform the changes we need to make to get our model producing reasonable results.

11.12 Exercises

1. Implement a program that iterates through a `LunaDataset` instance by wrapping it in a `DataLoader` instance, while timing how long it takes to do so. Compare these times to the times from the exercises in chapter 10. Be aware of the state of the cache when running the script.
   a. What impact does setting `num_workers=...` to 0, 1, and 2 have?
   b. What are the highest values your machine will support for a given combination of `batch_size=...` and `num_workers=...` without running out of memory?
2. Reverse the sort order of `noduleInfo_list`. How does that change the behavior of the model after one epoch of training?
3. Change `logMetrics` to alter the naming scheme of the runs and keys that are used in TensorBoard.
   a. Experiment with different forward-slash placement for keys passed in to `writer.add_scalar`.
   b. Have both training and validation runs use the same writer, and add the `trn` or `val` string to the name of the key.
   c. Customize the naming of the log directory and keys to suit your taste.

11.13 Summary

- Data loaders can be used to load data from arbitrary datasets in multiple processes. This allows otherwise-idle CPU resources to be devoted to preparing data to feed to the GPU.
- Data loaders load multiple samples from a dataset and collate them into a batch. PyTorch models expect to process batches of data, not individual samples.
- Data loaders can be used to manipulate arbitrary datasets by changing the relative frequency of individual samples. This allows for "after-market" tweaks to a dataset, though it might make more sense to change the dataset implementation directly.

- We will use PyTorch's `torch.optim.SGD` (stochastic gradient descent) optimizer with a learning rate of 0.001 and a momentum of 0.99 for the majority of part 2. These values are also reasonable defaults for many deep learning projects.
- Our initial model for classification will be very similar to the model we used in chapter 8. This lets us get started with a model that we have reason to believe will be effective. We can revisit the model design if we think it's the thing preventing our project from performing better.
- The choice of metrics that we monitor during training is important. It is easy to accidentally pick metrics that are misleading about how the model is performing. Using the overall percentage of samples classified correctly is not useful for our data. Chapter 12 will detail how to evaluate and choose better metrics.
- TensorBoard can be used to display a wide range of metrics visually. This makes it much easier to consume certain forms of information (particularly trend data) as they change per epoch of training.