


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.

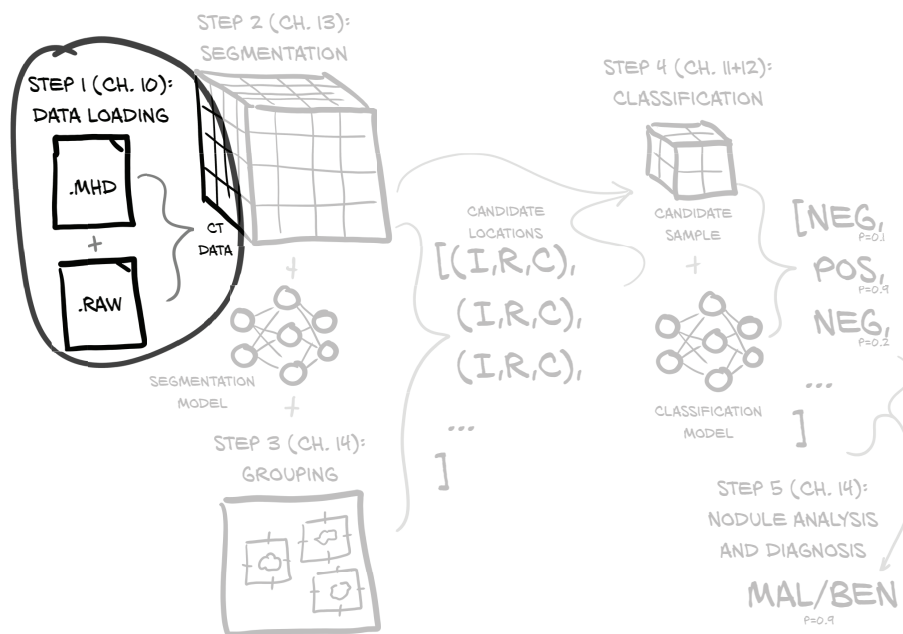


Figure 10.1 Our end-to-end lung cancer detection project, with a focus on this chapter's topic: step 1, data loading

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.

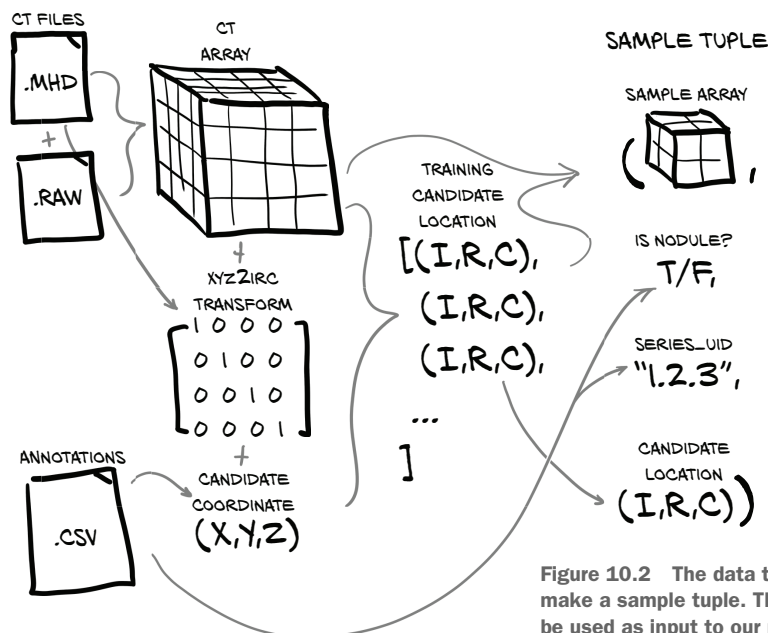


Figure 10.2 The data transforms required to make a sample tuple. These sample tuples will be used as input to our model training routine.

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 **node**, 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.

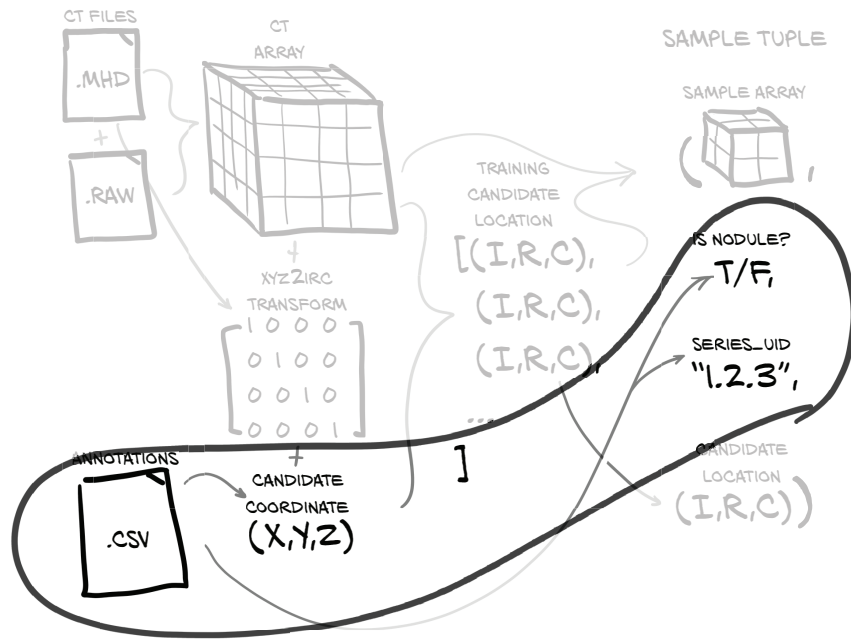


Figure 10.3 The LUNA annotations in candidates.csv contain the CT series, the nodule candidate's position, and a flag indicating if the candidate is actually a nodule or not.

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
```

Counts the number of lines in the file

```
$ 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
```

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:

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

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

```
$ 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
...
```

The last column is also different.


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 Nth 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.*,' 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 `dssets.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

```
→ @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}
```

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



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 `dssets.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.

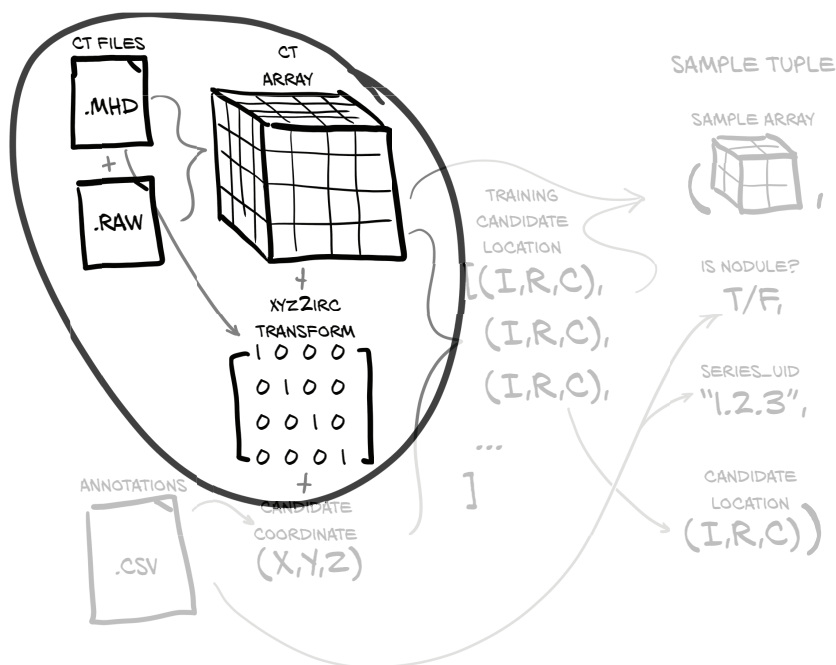


Figure 10.4 Loading a CT scan produces a **voxel** array and a transformation from patient coordinates to array indices.

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 dssets.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.float32

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.

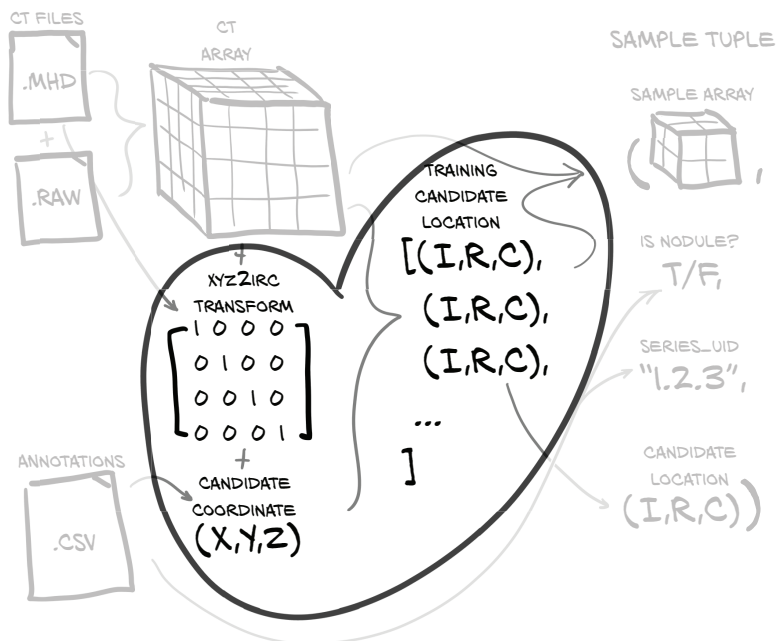


Figure 10.5 Using the transformation information to convert a nodule center coordinate in patient coordinates (X,Y,Z) to an array index (Index,Row,Column).

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*.

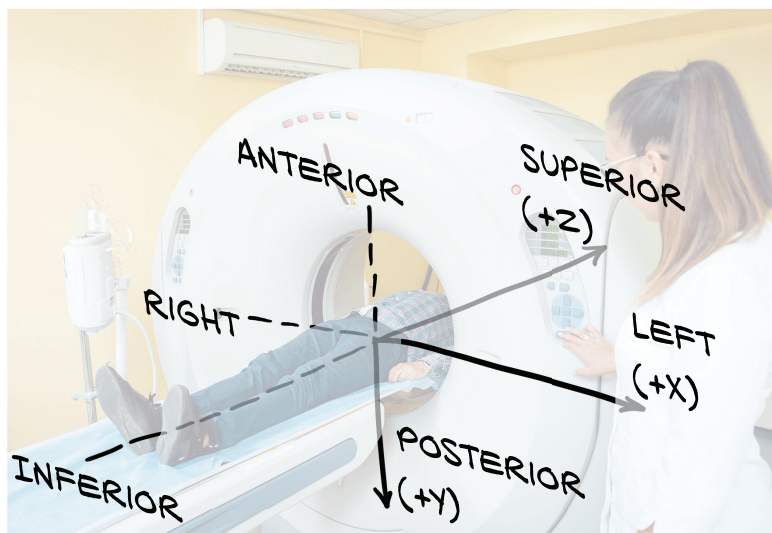


Figure 10.6 Our inappropriately clothed patient demonstrating the axes of the patient coordinate system

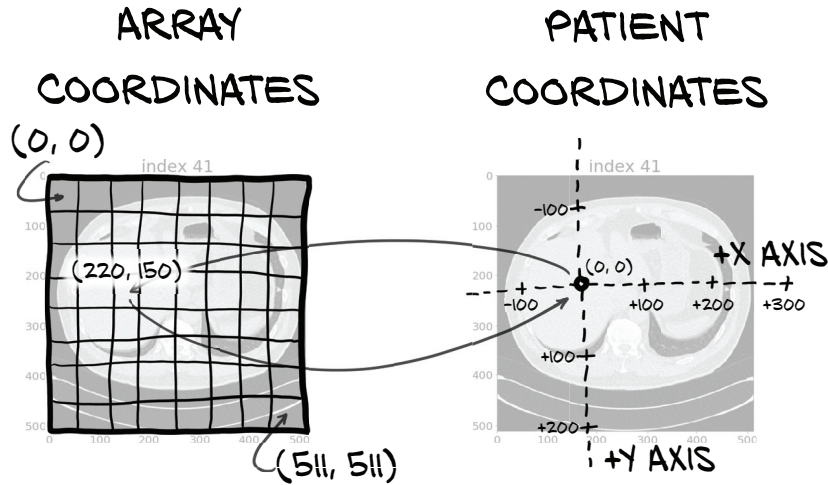




Figure 10.7 Array coordinates and patient coordinates have different origins and scaling.


 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

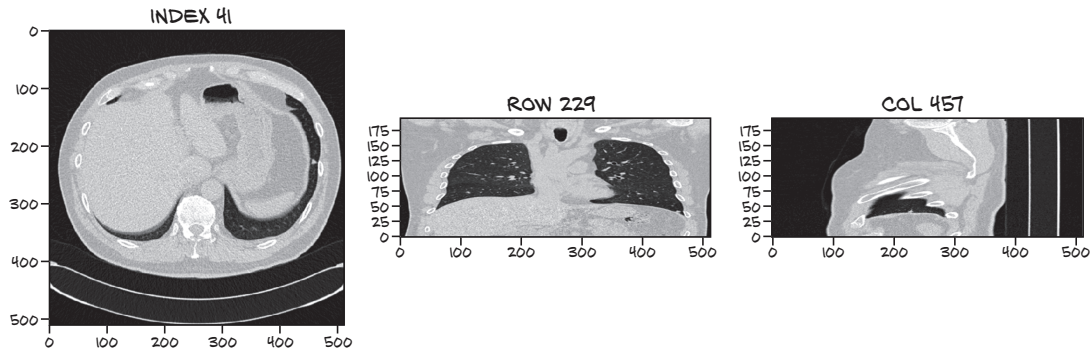


Figure 10.8 A CT scan with non-cubic voxels along the index-axis. Note how compressed the lungs are from top to bottom.

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

Swaps the order while we
convert to a NumPy array

```
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)
```

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

```
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]))
```

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.

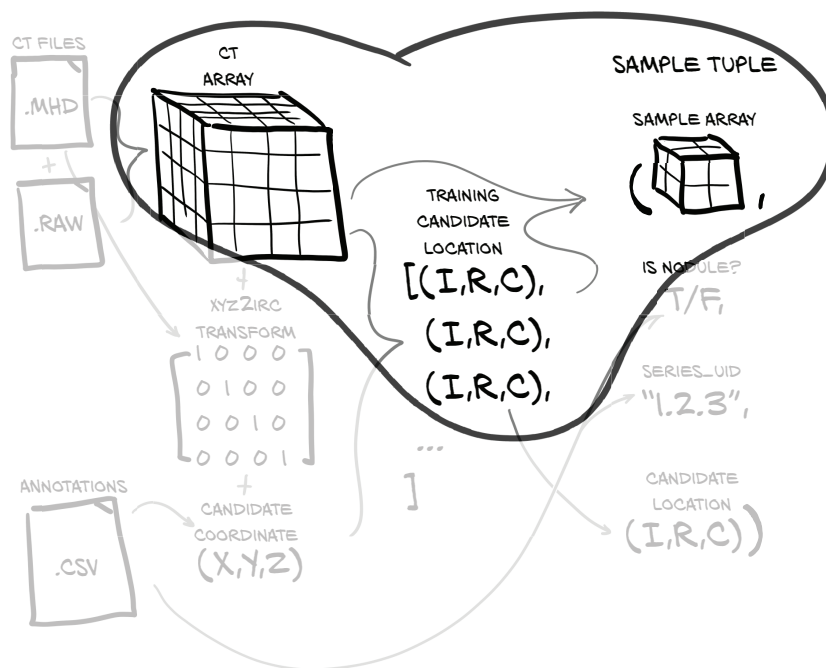


Figure 10.9 Cropping a candidate sample out of the larger CT voxel array using the candidate center's array coordinate information (Index,Row,Column)

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

The `getRowNodule` 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.getRowCandidate`

```
def getRowCandidate(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.]]]]),
cls_t → tensor([0, 1]),
        '1.3.6...287966244644280690737019247886',
        tensor([ 91, 360, 341]))
```

← `candidate_tup.series_uid` (elided)

← `center_irc`

← `candidate_t`

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.