12 Improving training with metrics and augmentation

This chapter covers

- Defining and computing precision, recall, and true/false positives/negatives
- Using the F1 score versus other quality metrics
- Balancing and augmenting data to reduce overfitting
- Using TensorBoard to graph quality metrics

The close of the last chapter left us in a predicament. While we were able to get the mechanics of our deep learning project in place, none of the results were actually useful; the network simply classified everything as non-nodule! To make matters worse, the results seemed great on the surface, since we were looking at the overall percent of the training and validation sets that were classified correctly. With our data heavily skewed toward negative samples, blindly calling everything negative is a

quick and easy way for our model to score well. Too bad doing so makes the model basically useless!

That means we're still focused on the same part of figure 12.1 as we were in chapter 11. But now we're working on getting our classification model working *well* instead of *at all*. This chapter is all about how to measure, quantify, express, and then improve on how well our model is doing its job.

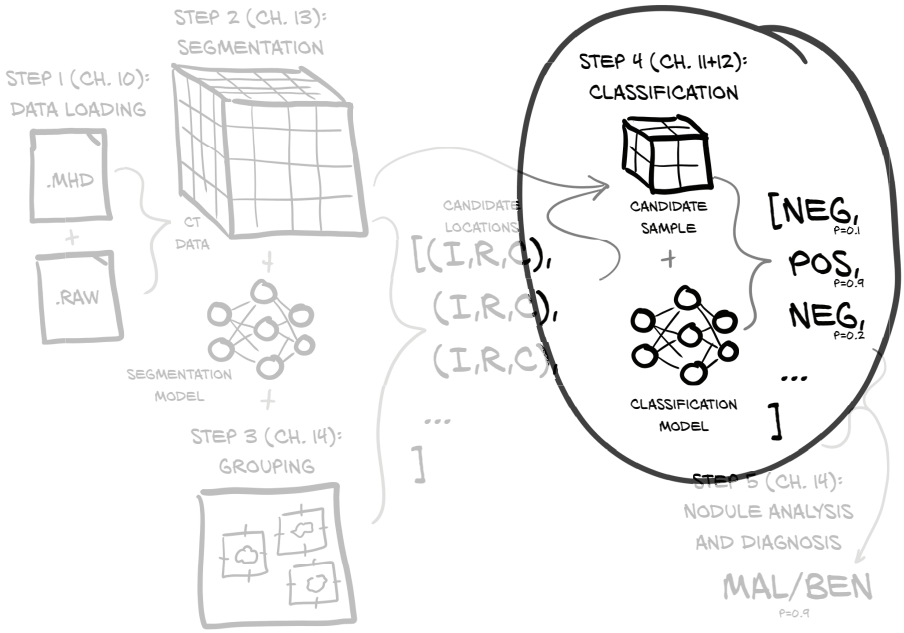


Figure 12.1 Our end-to-end lung cancer detection project, with a focus on this chapter's topic: step 4, classification

12.1 High-level plan for improvement

While a bit abstract, figure 12.2 shows us how we are going to approach that broad set of topics.

Let's walk through this somewhat abstract map of the chapter in detail. We will be dealing with the issues we're facing, like excessive focus on a single, narrow metric and the resulting behavior being useless in the general sense. In order to make some of this chapter's concepts a bit more concrete, we'll first employ a metaphor that puts our troubles in more tangible terms: in figure 12.2, (1) Guard Dogs and (2) Birds and Burglars.

After that, we will develop a graphical language to represent some of the core concepts needed to formally discuss the issues with the implementation from the last chapter: (3) Ratios: Recall and Precision. Once we have those concepts solidified, we'll touch on some math using those concepts that will encapsulate a more robust way of grading our model's performance and condensing it into a single number: (4) New Metric: F1 Score. We will implement the formula for those new metrics and look

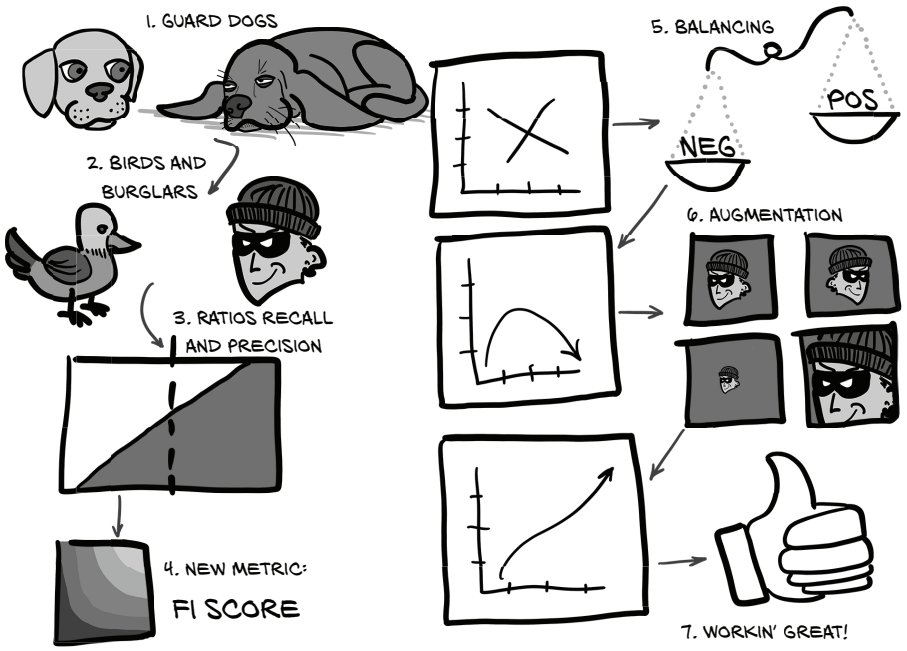


Figure 12.2 The metaphors we'll use to modify the metrics measuring our model to make it magnificent

at the how the resulting values change epoch by epoch during training. Finally, we'll make some much-needed changes to our `LunaDataset` implementation with an aim at improving our training results: (5) Balancing and (6) Augmentation. Then we will see if those experimental changes have the expected impact on our performance metrics.

By the time we're through with this chapter, our trained model will be performing much better: (7) Workin' Great! While it won't be ready to drop into clinical use just yet, it will be capable of producing results that are clearly better than random. This will mean we have a workable implementation of step 4, nodule candidate classification; and once we're finished, we can begin to think about how to incorporate steps 2 (segmentation) and 3 (grouping) into the project.

12.2 Good dogs vs. bad guys: False positives and false negatives

Instead of models and tumors, we're going to consider the two guard dogs in figure 12.3, both fresh out of obedience school. They both want to alert us to burglars—a rare but serious situation that requires prompt attention.

Unfortunately, while both dogs are good dogs, neither is a good *guard* dog. Our terrier (Roxie) barks at just about everything, while our old hound dog (Preston) barks almost exclusively at burglars—but only if he happens to be awake when they arrive.

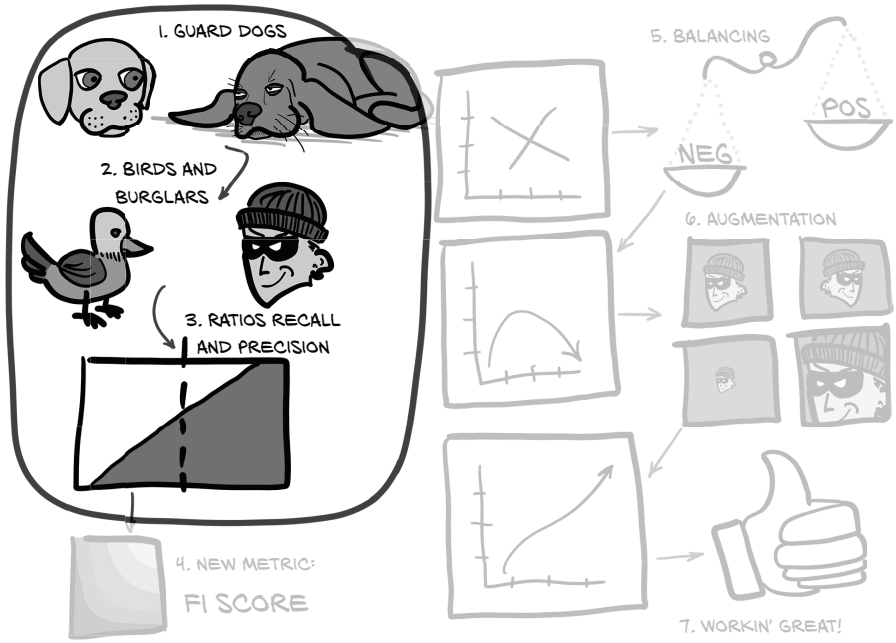


Figure 12.3 The set of topics for this chapter, with a focus on the framing metaphor

Roxie *will* alert us to a burglar just about every time. She will also alert us to fire engines, thunderstorms, helicopters, birds, the mail carrier, squirrels, passersby, and so on. If we follow up on every bark, we'll almost never get robbed (only the sneakiest of sneak-thieves can slip past). Perfect! … Except that being that diligent means we aren't really saving any work by having a guard dog. Instead, we'll be up every couple of hours, flashlight in hand, due to Roxie having smelled a cat, or heard an owl, or seen a late bus wander by. Roxie has a problematic number of false positives.

A *false positive* is an event that is classified as of interest or as a member of the desired class (positive as in "Yes, that's the type of thing I'm interested in knowing about") but that in truth is *not* really of interest. For the nodule-detection problem, it's when an actually uninteresting candidate is flagged as a nodule and, hence, in need of a radiologist's attention. For Roxie, these would be fire engines, thunderstorms, and so on. We will use an image of a cat as the canonical false positive in the next section and the figures that follow throughout the rest of the chapter.

Contrast false positives with *true positives*: items of interest that are classified correctly. These will be represented in the figures by a human burglar.

Meanwhile, if Preston barks, call the police, since that means someone has almost certainly broken in, the house is on fire, or Godzilla is attacking. Preston is a deep sleeper, however, and the sound of an in-progress home invasion isn't likely to rouse him, so we'll still get robbed just about every time someone tries. Again, while it's better than nothing, we're not really ending up with the peace of mind that motivated us to get a dog in the first place. Preston has a problematic number of false negatives.

A *false negative* is an event that is classified as not of interest or not a member of the desired class (negative as in "No, that's not the type of thing I'm interested in knowing about") but that in truth *is* actually of interest. For the nodule-detection problem, it's when a nodule (that is, a potential cancer) goes undetected. For Preston, these would be the robberies that he sleeps through. We'll get a bit creative here and use a picture of a *rodent* burglar for false negatives. They're sneaky!

Contrast false negatives with *true negatives*: uninteresting items that are correctly identified as such. We'll go with a picture of a bird for these.

Just to complete the metaphor, chapter 11's model is basically a cat that refuses to meow at anything that isn't a can of tuna (while stoically ignoring Roxie). Our focus at the end of the last chapter was on the percent correct for the overall training and validation sets. Clearly, that wasn't a great way to grade ourselves, and as we can see from each of our dogs' myopic focus on a single metric—like the number of true positives or true negatives—we need a metric with a broader focus to capture our overall performance.

12.3 Graphing the positives and negatives

Let's start developing the visual language we'll use to describe true/false positives/negatives. Please bear with us if our explanation gets repetitive; we want to make sure you develop a solid mental model for the ratios we're going to discuss. Consider figure 12.4, which shows events that might be of interest to one of our guard dogs.

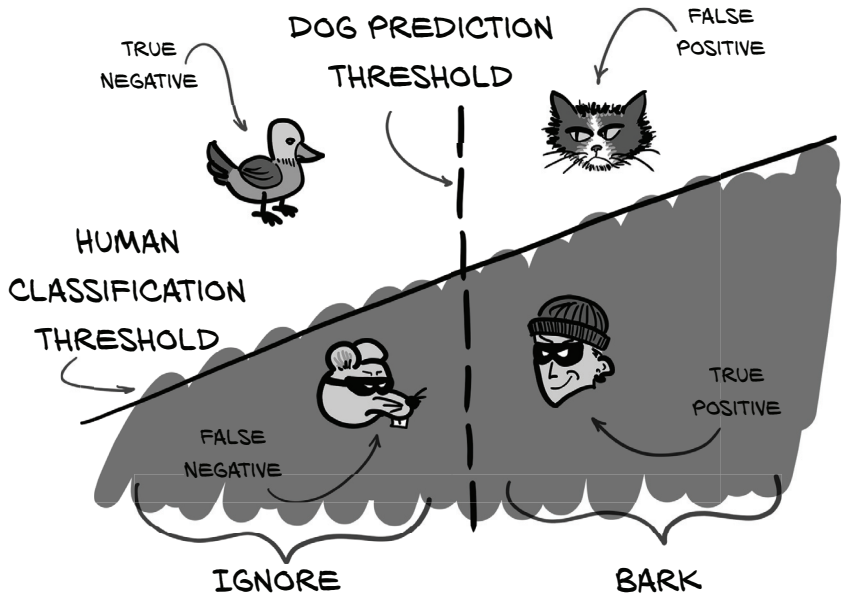


Figure 12.4 Cats, birds, rodents, and robbers make up our four classification quadrants. They are separated by a human label and the dog classification threshold.

We'll use two thresholds in figure 12.4. The first is the human-decided dividing line that separates burglars from harmless animals. In concrete terms, this is the label that is given for each training or validation sample. The second is the dog-determined *classification threshold* that determines whether the dog will bark at something. For a deep learning model, this is the predicted value that the model produces when considering a sample.

The combination of these two thresholds divides our events into quadrants: true/false positives/negatives. We will shade the events of concern with a darker background (what with those bad guys sneaking around in the dark all the time).

Of course, reality is far more complicated. There is no Platonic ideal of a burglar, and no single point relative to the classification threshold at which all burglars will be located. Instead, figure 12.5 shows us that some burglars will be particularly sneaky, and some birds will be particularly annoying. We will also go ahead and enclose our instances in a graph. Our X-axis will remain the bark-worthiness of each event, as determined by one of our guard dogs. We're going to have the Y-axis represent some vague set of qualities that we as humans are able to perceive, but our dogs cannot.

Since our model produces a binary classification, we can think of the prediction threshold as comparing a single-numerical-value output to our classification threshold value. This is why we will require that the classification threshold line to be perfectly vertical in figure 12.5.

Each possible burglar is different, so our guard dogs will need to evaluate many different situations, and that means more opportunities to make mistakes. We can see the clear diagonal line that separates the birds from the burglars, but Preston and Roxie can only perceive the X-axis here: they have a muddled, overlapped set of

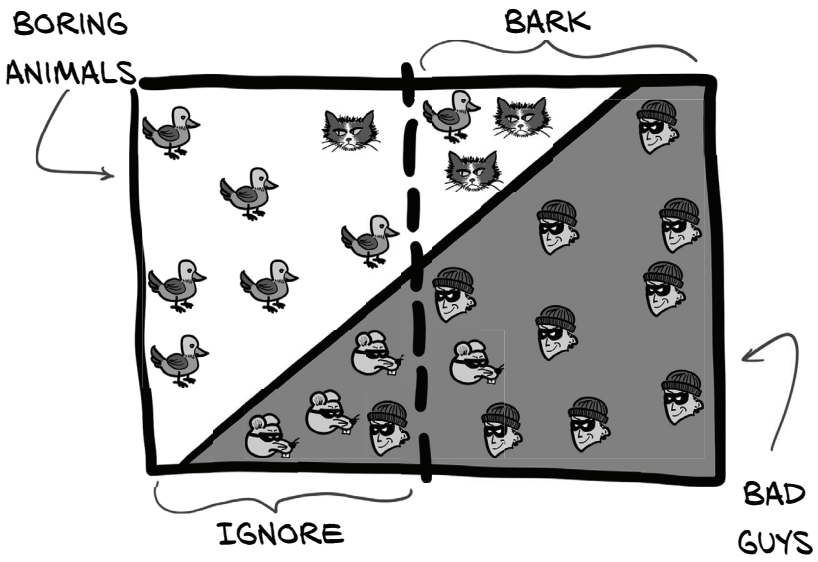


Figure 12.5 Each type of event will have many possible instances that our guard dogs will need to evaluate.

events in the middle of our graph. They must pick a vertical bark-worthiness threshold, which means it's impossible for either one of them to do so perfectly. Sometimes the person hauling your appliances to their van is the repair person you hired to fix your washing machine, and sometimes burglars show up in a van that says "Washing Machine Repair" on the side. Expecting a dog to pick up on those nuances is bound to fail.

The actual input data we're going to use has high dimensionality—we need to consider a ton of CT voxel values, along with more abstract things like candidate size, overall location in the lungs, and so on. The job of our model is to map each of these events and respective properties into this rectangle in such a way that we can separate those positive and negative events cleanly using a single vertical line (our classification threshold). This is done by the `nn.Linear` layers at the end of our model. The position of the vertical line corresponds exactly to the `classificationThreshold_float` we saw in section 11.6.1. There, we chose the hardcoded value 0.5 as our threshold.

Note that in reality, the data presented is not two-dimensional; it goes from very-high-dimensional after the second-to-last layer, to one-dimensional (here, our X-axis) at the output—just a single scalar per sample (which is then bisected by the classification threshold). Here, we use the second dimension (the Y-axis) to represent per-sample features that our model cannot see or use: things like age or gender of the patient, location of the nodule candidate in the lung, or even local aspects of the candidate that the model hasn't utilized. It also gives us a convenient way to represent confusion between non-nodule and nodule samples.

The quadrant areas in figure 12.5 and the count of samples contained in each will be the values we use to discuss model performance, since we can use the ratios between these values to construct increasingly complex metrics that we can use to objectively measure how well we are doing. As they say, "the proof is in the proportions."[1] Next, we'll use ratios between these event subsets to start defining better metrics.

12.3.1 Recall is Roxie's strength

Recall is basically "Make sure you never miss any interesting events!" Formally, *recall* is the ratio of the true positives to the union of true positives and false negatives. We can see this depicted in figure 12.6.

NOTE In some contexts, recall is referred to as *sensitivity*.

To improve recall, minimize false negatives. In guard dog terms, that means if you're unsure, bark at it, just in case. Don't let any rodent thieves sneak by on your watch!

Roxie accomplishes having an incredibly high recall by pushing her classification threshold all the way to the left, such that it encompasses nearly all of the positive events in figure 12.7. Note how doing so means her recall value is near 1.0, which means 99% of robbers are barked at. Since that's how Roxie defines success, in her mind, she's doing a great job. Never mind the huge expanse of false positives!

[1] No one actually says this.

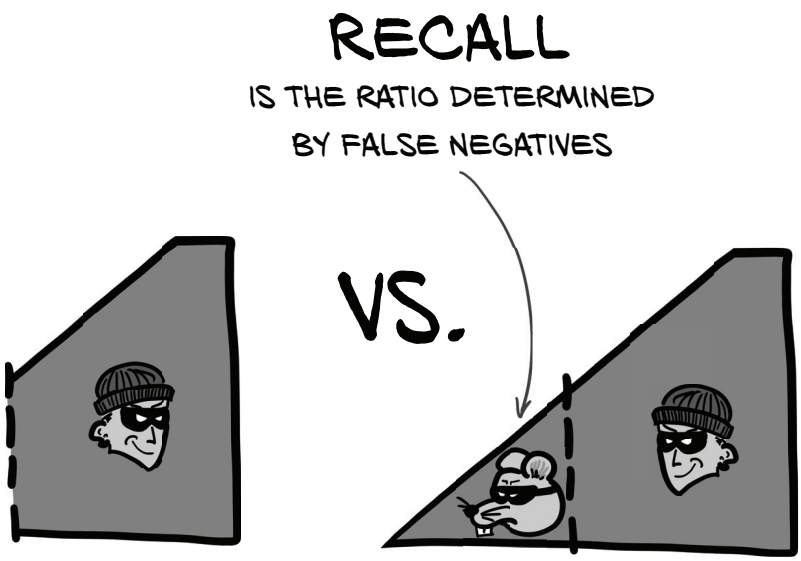


Figure 12.6 Recall is the ratio of the true positives to the union of true positives and false negatives. High recall minimizes false negatives.

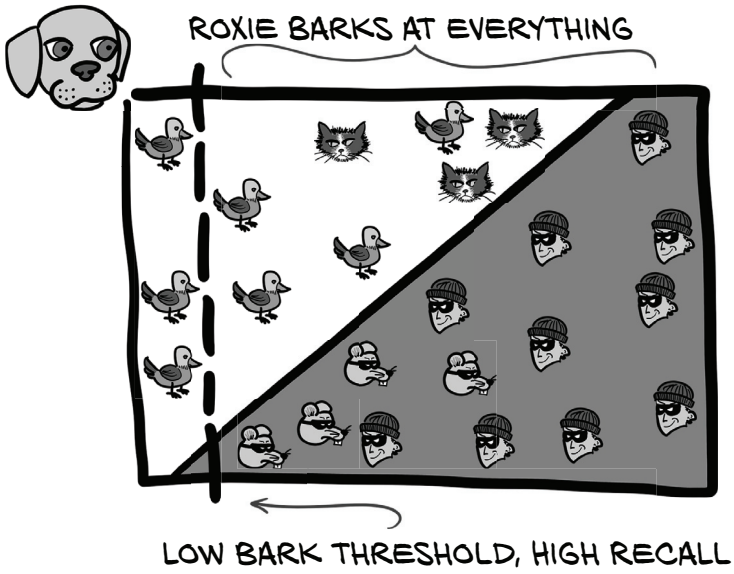


Figure 12.7 Roxie's choice of threshold prioritizes minimizing false negatives. Every last rat is barked at . . . and cats, and most birds.

12.3.2 Precision is Preston's forte

Precision is basically "Never bark unless you're sure." To improve precision, minimize false positives. Preston won't bark at something unless he's certain it's a burglar. More formally, *precision* is the ratio of the true positives to the union of true positives and false positives, as shown in figure 12.8.

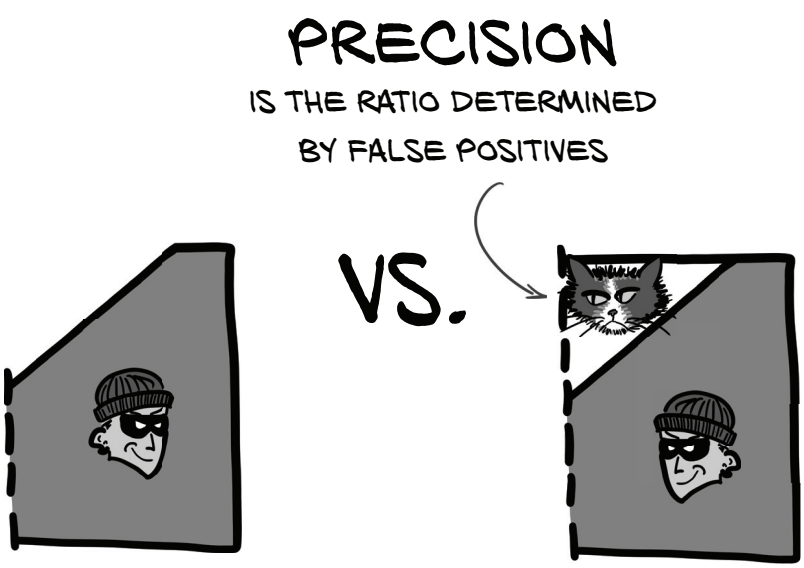


Figure 12.8 Precision is the ratio of the true positives to the union of true positives and false positives. High precision minimizes false positives.

Preston accomplishes having an incredibly high precision by pushing his classification threshold all the way to the right, such that it excludes as many uninteresting, negative events as he can manage (see figure 12.9). This is the opposite of Roxie's approach and means Preston has a precision of nearly 1.0: 99% of the things he barks at are robbers. This also matches his definition of being a good guard dog, even though a large number of events pass undetected.

While neither precision nor recall can be the single metric used to grade our model, they are both useful numbers to have on hand during training. Let's calculate and display these as part of our training program, and then we'll discuss other metrics we can employ.

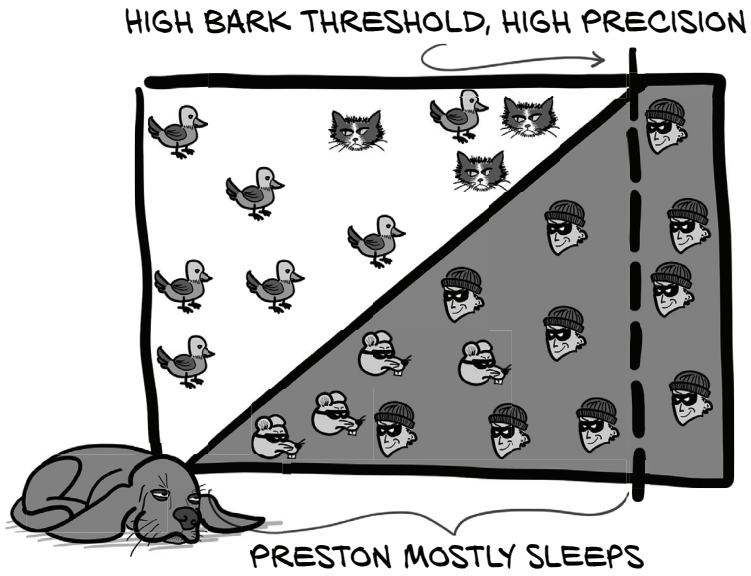


Figure 12.9 Preston's choice of threshold prioritizes minimizing false positives. Cats get left alone; only burglars are barked at!

12.3.3 Implementing precision and recall in logMetrics

Both precision and recall are valuable metrics to be able to track during training, since they provide important insight into how the model is behaving. If either of them drops to zero (as we saw in chapter 11!), it's likely that our model has started to behave in a degenerate manner. We can use the exact details of the behavior to guide where to investigate and experiment with getting training back on track. We'd like to update the `logMetrics` function to add precision and recall to the output we see for each epoch, to complement the loss and correctness metrics we already have.

We've been defining precision and recall in terms of "true positives" and the like thus far, so we will continue to do so in the code. It turns out that we are already computing some of the values we need, though we had named them differently.

Listing 12.1 training.py:315, `LunaTrainingApp.logMetrics`

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

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

falsePos_count = neg_count - neg_correct
falseNeg_count = pos_count - pos_correct
```

Here, we can see that `neg_correct` is the same thing as `trueNeg_count`! That actually makes sense, since non-nodule is our "negative" value (as in "a negative diagnosis"), and if the classifier gets the prediction correct, then that's a true negative. Similarly, correctly labeled nodule samples are true positives.

We do need to add the variables for our false positive and false negative values. That's straightforward, since we can take the total number of benign labels and subtract the count of the correct ones. What's left is the count of non-nodule samples misclassified as *positive*. Hence, they are false positives. Again, the false negative calculation is of the same form, but uses nodule counts.

With those values, we can compute `precision` and `recall` and store them in `metrics_dict`.

Listing 12.2 training.py:333, `LunaTrainingApp.logMetrics`

```
precision = metrics_dict['pr/precision'] = \
  truePos_count / np.float32(truePos_count + falsePos_count)
recall  = metrics_dict['pr/recall'] = \
  truePos_count / np.float32(truePos_count + falseNeg_count)
```

Note the double assignment: while having separate `precision` and `recall` variables isn't strictly necessary, they improve the readability of the next section. We also extend the logging statement in `logMetrics` to include the new values, but we skip the implementation for now (we'll revisit logging later in the chapter).

12.3.4 Our ultimate performance metric: The F1 score

While useful, neither precision nor recall entirely captures what we need in order to be able to evaluate a model. As we've seen with Roxie and Preston, it's possible to game either one individually by manipulating our classification threshold, resulting in a model that scores well on one or the other but does so at the expense of any real-world utility. We need something that combines both of those values in a way that prevents such gamesmanship. As we can see in figure 12.10, it's time to introduce our ultimate metric.

The generally accepted way of combining precision and recall is by using the F1 score (https://en.wikipedia.org/wiki/F1_score). As with other metrics, the F1 score ranges between 0 (a classifier with no real-world predictive power) and 1 (a classifier that has perfect predictions). We will update `logMetrics` to include this as well.

Listing 12.3 training.py:338, `LunaTrainingApp.logMetrics`

```
metrics_dict['pr/f1_score'] = \
  2 * (precision * recall) / (precision + recall)
```

At first glance, this might seem more complicated than we need, and it might not be immediately obvious how the F1 score behaves when trading off precision for recall or

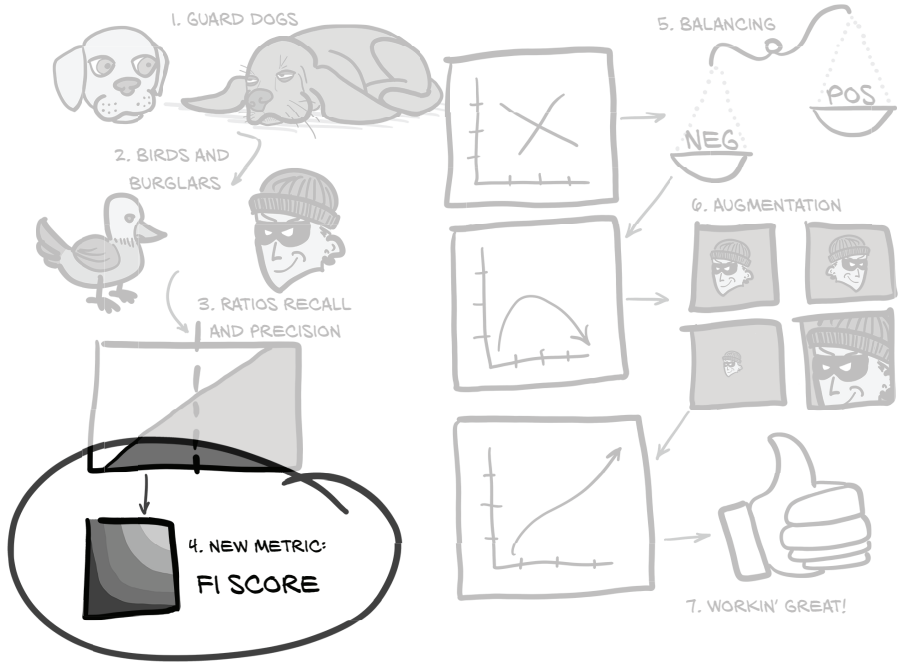


Figure 12.10 The set of topics for this chapter, with a focus on the final F1 score metric

vice versa. This formula has a lot of nice properties, however, and it compares favorably to several other, simpler alternatives that we might consider.

One immediate possibility for a scoring function is to average the values for precision and recall together. Unfortunately, this gives both `avg(p=1.0, r=0.0)` and `avg(p=0.5, r=0.5)` the same score of 0.5, and as we discussed earlier, a classifier with either precision or recall of zero is usually worthless. Giving something useless the same nonzero score as something useful disqualifies averaging as a meaningful metric immediately.

Still, let's visually compare averaging and F1 in figure 12.11. A few things stand out. First, we can see a lack of a curve or elbow in the contour lines for averaging. That's what lets our precision or recall skew to one side or the other! There will *never* be a situation where it doesn't make sense to maximize the score by having 100% recall (the Roxie approach) and then eliminate whichever false positives are easy to eliminate. That puts a floor on the addition score of 0.5 right out of the gate! Having a quality metric that is trivial to score at least 50% on doesn't feel right.

> **NOTE** What we are actually doing here is taking the *arithmetic mean* (https://en.wikipedia.org/wiki/Arithmetic_mean) of the precision and recall, both of which are *rates* rather than countable scalar values. Taking the arithmetic mean of rates doesn't typically give meaningful results. The F1 score is another name for the *harmonic mean* (https://en.wikipedia.org/wiki/Harmonic_mean) of the two rates, which is a more appropriate way of combining those kinds of values.

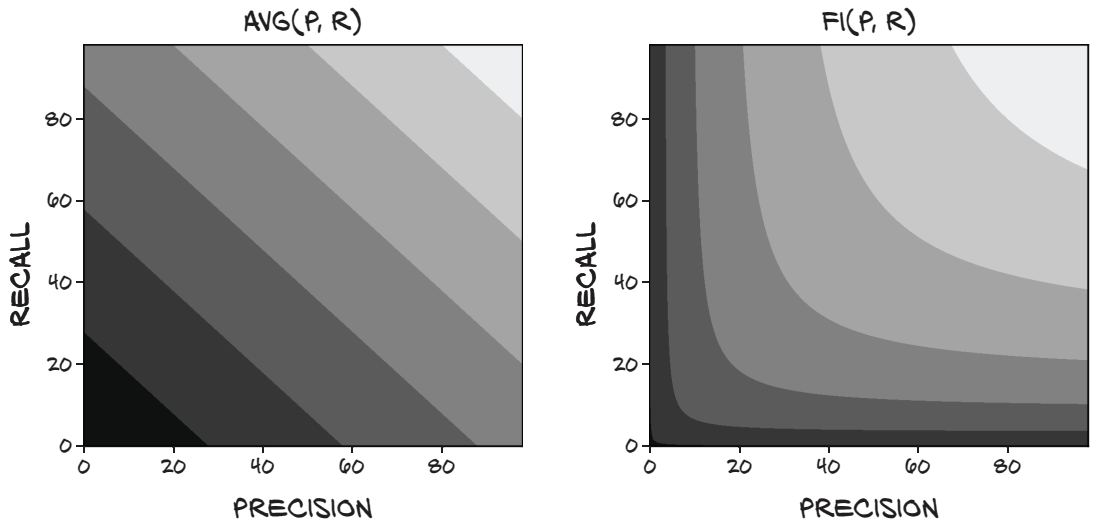


Figure 12.11 Computing the final score with `avg(p, r)`. Lighter values are closer to 1.0.

Contrast that with the F1 score: when recall is high but precision is low, trading off a lot of recall for even a little precision will move the score closer to that balanced sweet spot. There's a nice, deep elbow that is easy to slide into. That encouragement to have balanced precision and recall is what we want from our grading metric.

Let's say we still want a simpler metric, but one that doesn't reward skew at all. In order to correct for the weakness of addition, we might take the minimum of precision and recall (figure 12.12).

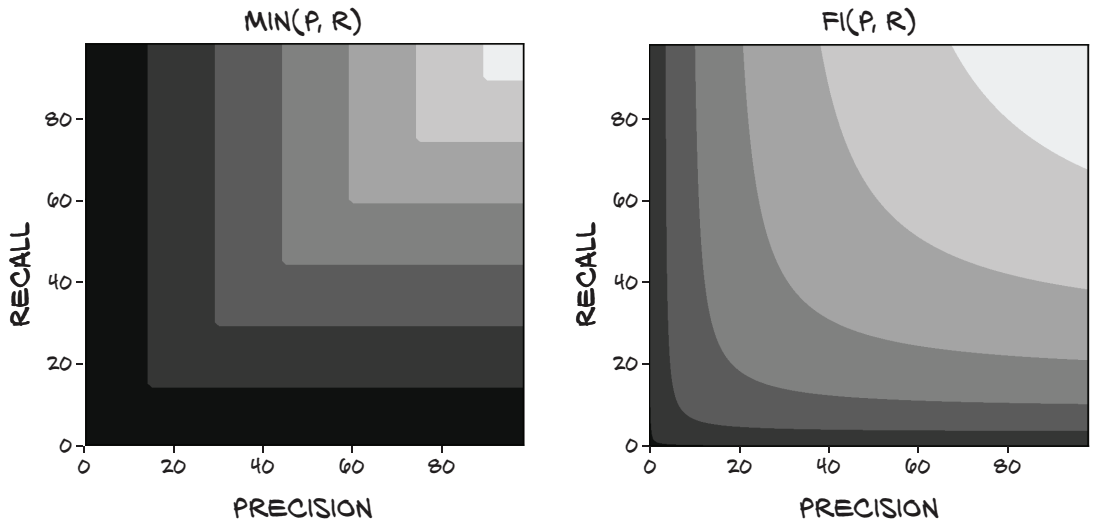


Figure 12.12 Computing the final score with `min(p, r)`

This is nice, because if either value is 0, the score is also 0, and the only way to get a score of 1.0 is to have both values be 1.0. However, it still leaves something to be desired, since making a model change that increased the recall from 0.7 to 0.9 while leaving precision constant at 0.5 wouldn't improve the score at all, nor would dropping recall down to 0.6! Although this metric is certainly penalizing having an imbalance between precision and recall, it isn't capturing a lot of nuance about the two values. As we have seen, it's easy to trade one off for the other simply by moving the classification threshold. We'd like our metric to reflect those trades.

We'll have to accept at least a bit more complexity to better meet our goals. We could multiply the two values together, as in figure 12.13. This approach keeps the nice property that if either value is 0, the score is 0, and a score of 1.0 means both inputs are perfect. It also favors a balanced trade-off between precision and recall at low values, though when it gets closer to perfect results, it becomes more linear. That's not great, since we really need to push both up to have a meaningful improvement at that point.

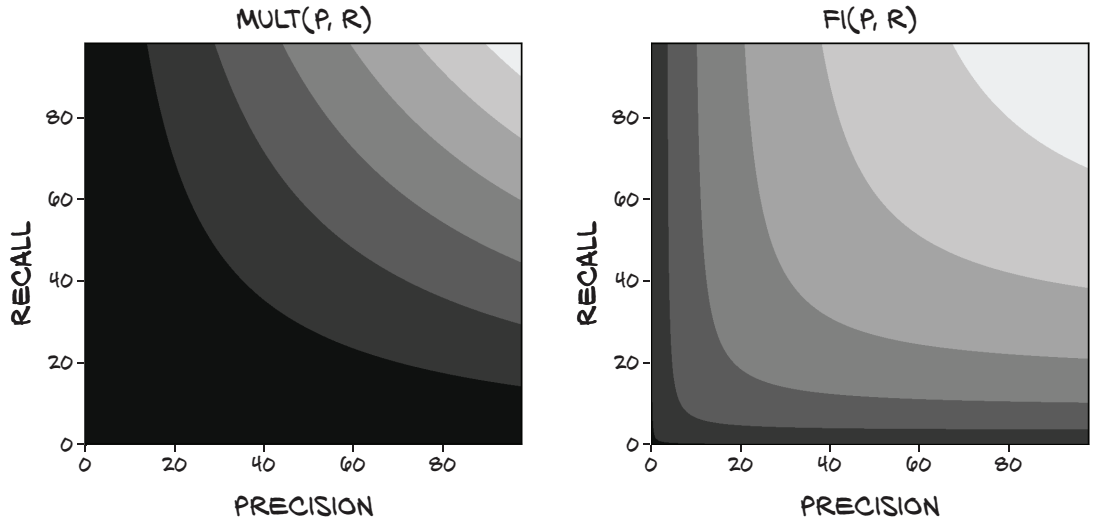


Figure 12.13 Computing the final score with `mult(p, r)`

NOTE Here we're taking the *geometric mean* (https://en.wikipedia.org/wiki/Geometric_mean) of two rates, which also doesn't produce meaningful results.

There's also the issue of having almost the entire quadrant from (0, 0) to (0.5, 0.5) be very close to zero. As we'll see, having a metric that's sensitive to changes in that region is important, especially in the early stages of our model design.

While using multiplication as our scoring function is feasible (it doesn't have any immediate disqualifications the way the previous scoring functions did), we will be using the F1 score to evaluate our classification model's performance going forward.

UPDATING THE LOGGING OUTPUT TO INCLUDE PRECISION, RECALL, AND F1 SCORE

Now that we have our new metrics, adding them to our logging output is pretty straightforward. We'll include precision, recall, and F1 in our main logging statement for each of our training and validation sets.

Listing 12.4 training.py:341, `LunaTrainingApp.logMetrics`

```
log.info(
  ("E{} {:8} {loss/all:.4f} loss, "
     + "{correct/all:-5.1f}% correct, "
     + "{pr/precision:.4f} precision, "
     + "{pr/recall:.4f} recall, "
     + "{pr/f1_score:.4f} f1 score"
  ).format(
    epoch_ndx,
    mode_str,
    **metrics_dict,
  )
)
```

Format string updated

In addition, we'll include exact values for the count of correctly identified and the total number of samples for each of the negative and positive samples.

Listing 12.5 training.py:353, `LunaTrainingApp.logMetrics`

```
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,
  )
)
```

The new version of the positive logging statement looks much the same.

12.3.5 How does our model perform with our new metrics?

Now that we've implemented our shiny new metrics, let's take them for a spin; we'll discuss the results after we show the results of the Bash shell session. You might want to read ahead while your system does its number crunching; this could take perhaps half an hour, depending on your system.[2] Exactly how long it takes will depend on your system's CPU, GPU, and disk speeds; our system with an SSD and GTX 1080 Ti took about 20 minutes per full epoch:

[2] If it's taking longer than that, make sure you've run the `prepcache` script.

```
$ ../.venv/bin/python -m p2ch12.training
Starting LunaTrainingApp...
...
E1 LunaTrainingApp

.../p2ch12/training.py:274: RuntimeWarning:
➥ invalid value encountered in double_scalars
  metrics_dict['pr/f1_score'] = 2 * (precision * recall) /
➥ (precision + recall)

E1 trn      0.0025 loss,  99.8% correct, 0.0000 prc, 0.0000 rcl, nan f1
E1 trn_ben  0.0000 loss, 100.0% correct (494735 of 494743)
E1 trn_mal  1.0000 loss,   0.0% correct (0 of 1215)

.../p2ch12/training.py:269: RuntimeWarning:
➥ invalid value encountered in long_scalars
  precision = metrics_dict['pr/precision'] = truePos_count /
➥ (truePos_count + falsePos_count)

E1 val      0.0025 loss,  99.8% correct, nan prc, 0.0000 rcl, nan f1
E1 val_ben  0.0000 loss, 100.0% correct (54971 of 54971)
E1 val_mal  1.0000 loss,   0.0% correct (0 of 136)
```

The exact count and line numbers of these RuntimeWarning lines might be different from run to run.

Bummer. We've got some warnings, and given that some of the values we computed were `nan`, there's probably a division by zero happening somewhere. Let's see what we can figure out.

First, since *none* of the positive samples in the training set are getting classified as positive, that means both precision and recall are zero, which results in our F1 score calculation dividing by zero. Second, for our validation set, `truePos_count` and `falsePos_count` are both zero due to *nothing* being flagged as positive. It follows that the denominator of our `precision` calculation is also zero; that makes sense, as that's where we're seeing another `RuntimeWarning`.

A handful of negative training samples are classified as positive (494735 of 494743 are classified as negative, so that leaves 8 samples misclassified). While that might seem odd at first, recall that we are collecting our training results *throughout the epoch*, rather than using the model's end-of-epoch state as we do for the validation results. That means the first batch is literally producing random results. A few of the samples from that first batch being flagged as positive isn't surprising.

> **NOTE** Due to both the random initialization of the network weights and the random ordering of the training samples, individual runs will likely exhibit slightly different behavior. Having exactly reproducible behavior can be desirable but is out of scope for what we're trying to do in part 2 of this book.

Well, that was somewhat painful. Switching to our new metrics resulted in going from A+ to "Zero, if you're lucky"—and if we're not lucky, the score is so bad that *it's not even a number*. Ouch.

That said, in the long run, this is good for us. We've known that our model's performance was garbage since chapter 11. If our metrics told us anything *but* that, it would point to a fundamental flaw in the metrics!

12.4 What does an ideal dataset look like?

Before we start crying into our cups over the current sorry state of affairs, let's instead think about what we actually want our model to do. Figure 12.14 says that first we need to balance our data so that our model can train properly. Let's build up the logical steps needed to get us there.

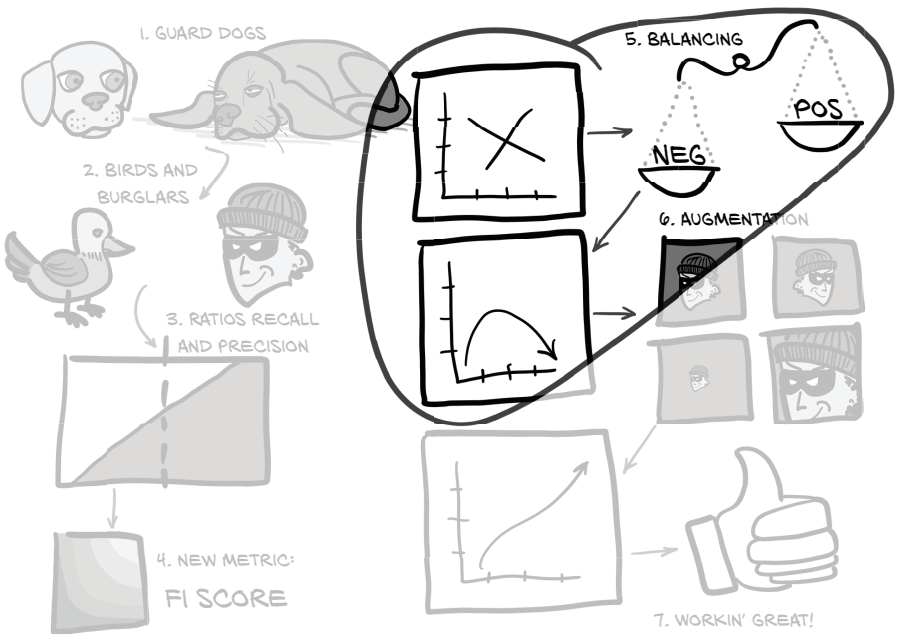


Figure 12.14 The set of topics for this chapter, with a focus on balancing our positive and negative samples

Recall figure 12.5 earlier, and the following discussion of classification thresholds. Getting better results by moving the threshold has limited effectiveness—there's just too much overlap between the positive and negative classes to work with.[3]

Instead, we want to see an image like figure 12.15. Here, our label threshold is nearly vertical. That's what we want, because it means the label threshold and our classification threshold can line up reasonably well. Similarly, most of the samples are concentrated at either end of the diagram. Both of these things require that our data be easily separable and that our model have the capacity to perform that separation. Our model currently has enough capacity, so that's not the issue. Instead, let's take a look at our data.

Recall that our data is wildly imbalanced. There's a 400:1 ratio of positive samples to negative ones. That's *crushingly* imbalanced! Figure 12.16 shows what that looks like. No wonder our "actually nodule" samples are getting lost in the crowd!

[3] Keep in mind that these images are just a representation of the classification space and do not represent ground truth.

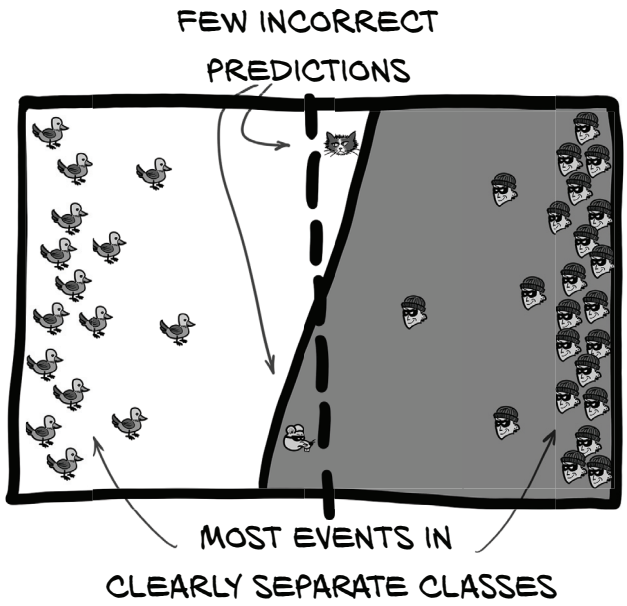


Figure 12.15 A well-trained model can cleanly separate data, making it easy to pick a classification threshold with few trade-offs.

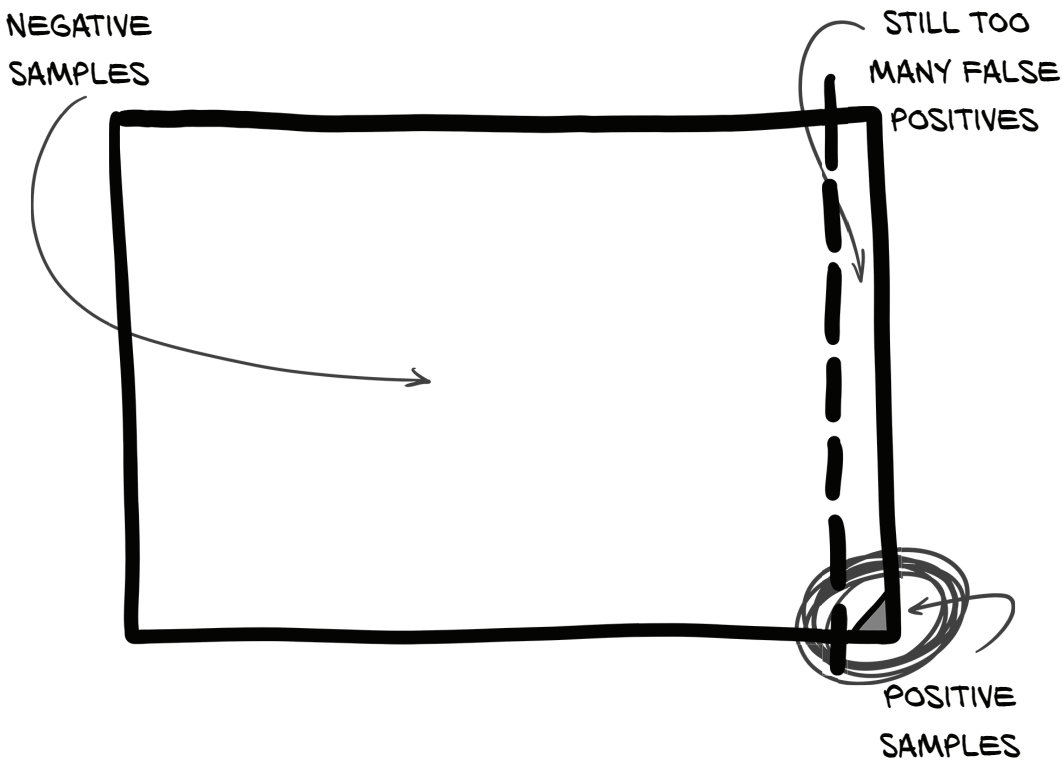


Figure 12.16 An imbalanced dataset that roughly approximates the imbalance in our LUNA classification data

Now, let's be perfectly clear: when we're done, our model will be able to handle this kind of data imbalance just fine. We could probably even train the model all the way there without changing the balancing, assuming we were willing to wait for a gajillion epochs first.[4] But we're busy people with things to do, so rather than cook our GPU until the heat death of the universe, let's try to make our training data look more ideal by changing the class balance we are training with.

12.4.1 Making the data look less like the actual and more like the "ideal"

The best thing to do would be to have relatively more positive samples. During the initial epoch of training, when we're going from randomized chaos to something more organized, having so few training samples be positive means they get drowned out.

The method by which this happens is somewhat subtle, however. Recall that since our network weights are initially randomized, the per-sample output of the network is also randomized (but clamped to the range [0-1]).

> **NOTE** Our loss function is `nn.CrossEntropyLoss`, which technically operates on the raw logits rather than the class probabilities. For our discussion, we'll ignore that distinction and assume the loss and the label-prediction deltas are the same thing.

The predictions numerically close to the correct label do not result in much change to the weights of the network, while predictions that are significantly different from the correct answer are responsible for a much greater change to the weights. Since the output is random when the model is initialized with random weights, we can assume that of our ~500k training samples (495,958, to be exact), we'll have the following approximate groups:

1. 250,000 negative samples will be predicted to be negative (0.0 to 0.5) and result in at most a small change to the network weights toward predicting negative.
2. 250,000 negative samples will be predicted to be positive (0.5 to 1.0) and result in a large swing toward the network weights predicting negative.
3. 500 positive samples will be predicted to be negative and result in a swing toward the network weights predicting positive.
4. 500 positive samples will be predicted to be positive and result in almost no change to the network weights.

> **NOTE** Keep in mind that the actual predictions are real numbers between 0.0 and 1.0 inclusive, so these groups won't have strict delineations.

Here's the kicker, though: groups 1 and 4 can be *any size*, and they will continue to have close to zero impact on training. The only thing that matters is that groups 2 and 3 can counteract each other's pull enough to prevent the network from collapsing to a degenerate "only output one thing" state. Since group 2 is 500 times larger than

[4] It's not clear if this is actually true, but it's plausible, and the loss *was* getting better . . .

group 3 and we're using a batch size of 32, roughly 500/32 = 15 batches will go by before seeing a single positive sample. That implies that 14 out of 15 training batches will be 100% negative and will only pull all model weights toward predicting negative. That lopsided pull is what produces the degenerate behavior we've been seeing.

Instead, we'd like to have just as many positive samples as negative ones. For the first part of training, then, half of both labels will be classified incorrectly, meaning that groups 2 and 3 should be roughly equal in size. We also want to make sure we present batches with a mix of negative and positive samples. Balance would result in the tug-of-war evening out, and the mixture of classes per batch will give the model a decent chance of learning to discriminate between the two classes. Since our LUNA data has only a small, fixed number of positive samples, we'll have to settle for taking the positive samples that we have and presenting them repeatedly during training.

Discrimination

Here, we define *discrimination* as "the ability to separate two classes from each other." Building and training a model that can tell "actually nodule" candidates from normal anatomical structures is the entire point of what we're doing in part 2.

Some other definitions of discrimination are more problematic. While out of scope for the discussion of our work here, there is a larger issue with models trained from real-world data. If that real-world dataset is collected from sources that have a real-world-discriminatory bias (for example, racial bias in arrest and conviction rates, or anything collected from social media), and that bias is not corrected for during dataset preparation or training, then the resulting model will continue to exhibit the same biases present in the training data. Just as in humans, racism is learned.

This means almost any model trained from internet-at-large data sources will be compromised in some fashion, unless extreme care is taken to scrub those biases from the model. Note that like our goal in part 2, this is considered an unsolved problem.

Recall our professor from chapter 11 who had a final exam with 99 false answers and 1 true answer. The next semester, after being told "You should have a more even balance of true and false answers," the professor decided to add a midterm with 99 true answers and 1 false one. "Problem solved!"

Clearly, the correct approach is to intermix true and false answers in a way that doesn't allow the students to exploit the larger structure of the tests to answer things correctly. Whereas a student would pick up on a pattern like "odd questions are true, even questions are false," the batching system used by PyTorch doesn't allow the model to "notice" or utilize that kind of pattern. Our training dataset will need to be updated to alternate between positive and negative samples, as in figure 12.17.

The unbalanced data is the proverbial needle in the haystack we mentioned at the start of chapter 9. If you had to perform this classification work by hand, you'd probably start to empathize with Preston.

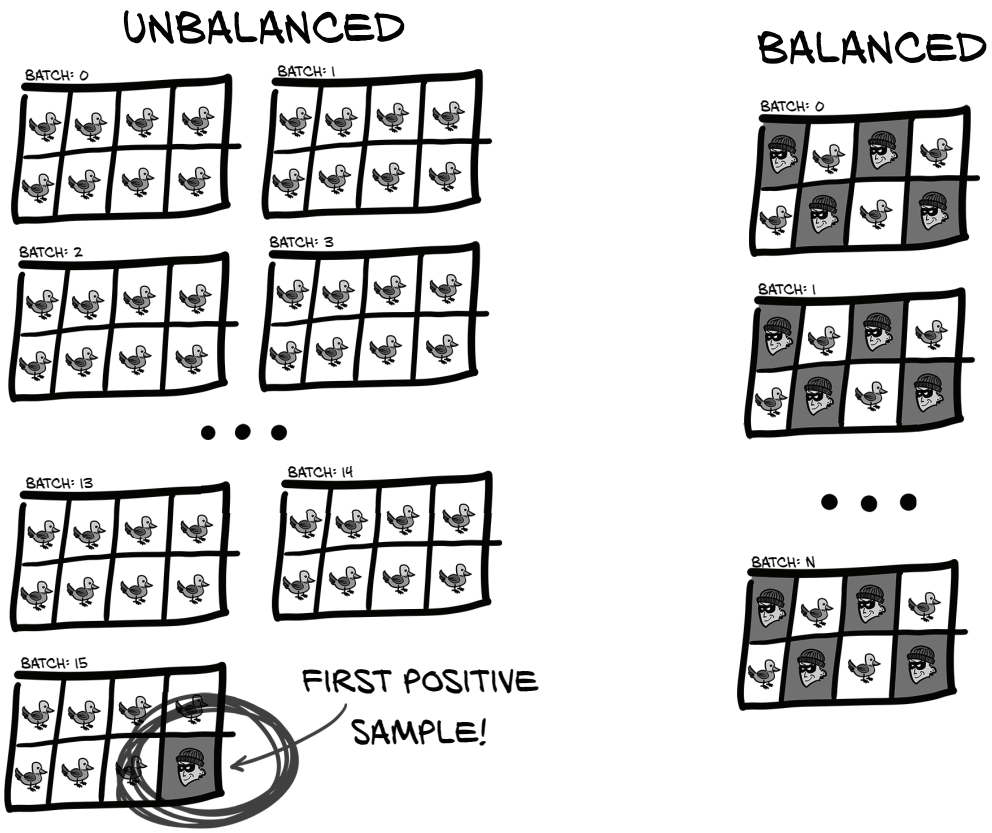


Figure 12.17 Batch after batch of imbalanced data will have nothing but negative events long before the first positive event, while balanced data can alternate every other sample.

We will not be doing any balancing for validation, however. Our model needs to function well in the real world, and the real world is imbalanced (after all, that's where we got the raw data!).

How should we accomplish this balancing? Let's discuss our choices.

SAMPLERS CAN RESHAPE DATASETS

One of the optional arguments to `DataLoader` is `sampler=…` . This allows the data loader to override the iteration order native to the dataset passed in and instead shape, limit, or reemphasize the underlying data as desired. This can be incredibly useful when working with a dataset that isn't under your control. Taking a public dataset and reshaping it to meet your needs is far less work than reimplementing that dataset from scratch.

The downside is that many of the mutations we could accomplish with samplers require that we break encapsulation of the underlying dataset. For example, let's assume we have a dataset like CIFAR-10 (www.cs.toronto.edu/~kriz/cifar.html) that

consists of 10 equally weighted classes, and we want to instead have 1 class (say, "airplane") now make up 50% of all of the training images. We could decide to use `WeightedRandomSampler` (http://mng.bz/8plK) and weight each of the "airplane" sample indexes higher, but constructing the `weights` argument requires that we know in advance which indexes are airplanes.

As we discussed, the `Dataset` API only specifies that subclasses provide `__len__` and `__getitem__`, but there is nothing direct we can use to ask "Which samples are airplanes?" We'd either have to load up every sample beforehand to inquire about the class of that sample, or we'd have to break encapsulation and hope the information we need is easily obtained from looking at the internal implementation of the `Dataset` subclass.

Since neither of those options is particularly ideal in cases where we have control over the dataset directly, the code for part 2 implements any needed data shaping inside the `Dataset` subclasses instead of relying on an external sampler.

Implementing class balancing in the dataset

We are going to directly change our `LunaDataset` to present a balanced, one-to-one ratio of positive and negative samples for training. We will keep separate lists of negative training samples and positive training samples, and alternate returning samples from each of those two lists. This will prevent the degenerate behavior of the model scoring well by simply answering "false" to every sample presented. In addition, the positive and negative classes will be intermixed so that the weight updates are forced to discriminate between the classes.

Let's add a `ratio_int` to `LunaDataset` that will control the label for the *N*th sample as well as keep track of our samples separated by label.

Listing 12.6 dsets.py:217, `class LunaDataset`

```
class LunaDataset(Dataset):
  def __init__(self,
         val_stride=0,
         isValSet_bool=None,
         ratio_int=0,
      ):
    self.ratio_int = ratio_int
    # ... line 228
    self.negative_list = [
      nt for nt in self.candidateInfo_list if not nt.isNodule_bool
    ]
    self.pos_list = [
      nt for nt in self.candidateInfo_list if nt.isNodule_bool
    ]
    # ... line 265

  def shuffleSamples(self):
    if self.ratio_int:
      random.shuffle(self.negative_list)
      random.shuffle(self.pos_list)
```

We will call this at the top of each epoch to randomize the order of samples being presented.

With this, we now have dedicated lists for each label. Using these lists, it becomes much easier to return the label we want for a given index into the dataset. In order to make sure we're getting the indexing right, we should sketch out the ordering we want. Let's assume a `ratio_int` of 2, meaning a 2:1 ratio of negative to positive samples. That would mean every third index should be positive:

```
DS Index   0 1 2 3 4 5 6 7 8 9 ...
Label      + - - + - - + - - +
Pos Index  0     1     2     3
Neg Index    0 1   2 3   4 5
```

The relationship between the dataset index and the positive index is simple: divide the dataset index by 3 and then round down. The negative index is slightly more complicated, in that we have to subtract 1 from the dataset index and then subtract the most recent positive index as well.

Implemented in our `LunaDataset` class, that looks like the following.

Listing 12.7 dsets.py:286, `LunaDataset.__getitem__`

```
    def __getitem__(self, ndx):
      if self.ratio_int:
        pos_ndx = ndx // (self.ratio_int + 1)

        if ndx % (self.ratio_int + 1):
          neg_ndx = ndx - 1 - pos_ndx
          neg_ndx %= len(self.negative_list)
          candidateInfo_tup = self.negative_list[neg_ndx]
        else:
          pos_ndx %= len(self.pos_list)
          candidateInfo_tup = self.pos_list[pos_ndx]
      else:
        candidateInfo_tup = self.candidateInfo_list[ndx]
```

A ratio_int of zero means use the native balance.

A nonzero remainder means this should be a negative sample.

Overflow results in wraparound.

Returns the Nth sample if not balancing classes

That can get a little hairy, but if you desk-check it out, it will make sense. Keep in mind that with a low ratio, we'll run out of positive samples before exhausting the dataset. We take care of that by taking the modulus of `pos_ndx` before indexing into `self.pos_list`. While the same kind of index overflow should never happen with `neg_ndx` due to the large number of negative samples, we do the modulus anyway, just in case we later decide to make a change that might cause it to overflow.

We'll also make a change to our dataset's length. Although this isn't strictly necessary, it's nice to speed up individual epochs. We're going to hardcode our `__len__` to be 200,000.

Listing 12.8 dsets.py:280, `LunaDataset.__len__`

```
def __len__(self):
  if self.ratio_int:
    return 200000
  else:
    return len(self.candidateInfo_list)
```

We're no longer tied to a specific number of samples, and presenting "a full epoch" doesn't really make sense when we would have to repeat positive samples many, many times to present a balanced training set. By picking 200,000 samples, we reduce the time between starting a training run and seeing results (faster feedback is always nice!), and we give ourselves a nice, clean number of samples per epoch. Feel free to adjust the length of an epoch to meet your needs.

For completeness, we also add a command-line parameter.

Listing 12.9 training.py:31, `class LunaTrainingApp`

```
class LunaTrainingApp:
  def __init__(self, sys_argv=None):
    # ... line 52
    parser.add_argument('--balanced',
      help="Balance the training data to half positive, half negative.",
      action='store_true',
      default=False,
    )
```

Then we pass that parameter into the `LunaDataset` constructor.

Listing 12.10 training.py:137, `LunaTrainingApp.initTrainDl`

```
def initTrainDl(self):
  train_ds = LunaDataset(
    val_stride=10,
    isValSet_bool=False,
    ratio_int=int(self.cli_args.balanced),
  )
```

Here we rely on python's `True` being convertible to a `1`.

We're all set. Let's run it!

12.4.2 Contrasting training with a balanced LunaDataset to previous runs

As a reminder, our unbalanced training run had results like these:

```
$ python -m p2ch12.training
...
E1 LunaTrainingApp
E1 trn      0.0185 loss,  99.7% correct, 0.0000 precision, 0.0000 recall,
➥ nan f1 score
```

```
E1 trn_neg  0.0026 loss, 100.0% correct (494717 of 494743)
E1 trn_pos  6.5267 loss,   0.0% correct (0 of 1215)
...
E1 val      0.0173 loss,  99.8% correct, nan precision, 0.0000 recall,
➥ nan f1 score
E1 val_neg  0.0026 loss, 100.0% correct (54971 of 54971)
E1 val_pos  5.9577 loss,   0.0% correct (0 of 136)
```

But when we run with `--balanced`, we see the following:

```
$ python -m p2ch12.training --balanced
...
E1 LunaTrainingApp
E1 trn      0.1734 loss,  92.8% correct, 0.9363 precision, 0.9194 recall,
➥ 0.9277 f1 score
E1 trn_neg  0.1770 loss,  93.7% correct (93741 of 100000)
E1 trn_pos  0.1698 loss,  91.9% correct (91939 of 100000)
...
E1 val      0.0564 loss,  98.4% correct, 0.1102 precision, 0.7941 recall,
➥ 0.1935 f1 score
E1 val_neg  0.0542 loss,  98.4% correct (54099 of 54971)
E1 val_pos  0.9549 loss,  79.4% correct (108 of 136)
```

This seems much better! We've given up about 5% correct answers on the negative samples to gain 86% correct positive answers. We're back into a solid B range again![5]

As in chapter 11, however, this result is deceptive. Since there are 400 times as many negative samples as positive ones, even getting just 1% wrong means we'd be incorrectly classifying negative samples as positive four times more often than there are actually positive samples in total!

Still, this is clearly better than the outright wrong behavior from chapter 11 and much better than a random coin flip. In fact, we've even crossed over into being (almost) legitimately useful in real-world scenarios. Recall our overworked radiologist poring over each and every speck of a CT: well, now we've got something that can do a reasonable job of screening out 95% of the false positives. That's a huge help, since it translates into about a tenfold increase in productivity for the machine-assisted human.

Of course, there's still that pesky issue of the 14% of positive samples that were missed, which we should probably deal with. Perhaps some additional epochs of training would help. Let's see (and again, expect to spend at least 10 minutes per epoch):

```
$ python -m p2ch12.training --balanced --epochs 20
...
E2 LunaTrainingApp
E2 trn      0.0432 loss,  98.7% correct, 0.9866 precision, 0.9879 recall,
➥ 0.9873 f1 score
E2 trn_ben  0.0545 loss,  98.7% correct (98663 of 100000)
E2 trn_mal  0.0318 loss,  98.8% correct (98790 of 100000)
```

[5] And remember that this is after only the 200,000 training samples presented, not the 500,000+ of the unbalanced dataset, so we got there in less than half the time.

```
E2 val      0.0603 loss,  98.5% correct, 0.1271 precision, 0.8456 recall,
➥ 0.2209 f1 score
E2 val_ben  0.0584 loss,  98.6% correct (54181 of 54971)
E2 val_mal  0.8471 loss,  84.6% correct (115 of 136)
...
E5 trn      0.0578 loss,  98.3% correct, 0.9839 precision, 0.9823 recall,
➥ 0.9831 f1 score
E5 trn_ben  0.0665 loss,  98.4% correct (98388 of 100000)
E5 trn_mal  0.0490 loss,  98.2% correct (98227 of 100000)
E5 val      0.0361 loss,  99.2% correct, 0.2129 precision, 0.8235 recall,
➥ 0.3384 f1 score
E5 val_ben  0.0336 loss,  99.2% correct (54557 of 54971)
E5 val_mal  1.0515 loss,  82.4% correct (112 of 136)...
...
E10 trn      0.0212 loss,  99.5% correct, 0.9942 precision, 0.9953 recall,
➥ 0.9948 f1 score
E10 trn_ben  0.0281 loss,  99.4% correct (99421 of 100000)
E10 trn_mal  0.0142 loss,  99.5% correct (99530 of 100000)
E10 val      0.0457 loss,  99.3% correct, 0.2171 precision, 0.7647 recall,
➥ 0.3382 f1 score
E10 val_ben  0.0407 loss,  99.3% correct (54596 of 54971)
E10 val_mal  2.0594 loss,  76.5% correct (104 of 136)
...
E20 trn      0.0132 loss,  99.7% correct, 0.9964 precision, 0.9974 recall,
➥ 0.9969 f1 score
E20 trn_ben  0.0186 loss,  99.6% correct (99642 of 100000)
E20 trn_mal  0.0079 loss,  99.7% correct (99736 of 100000)
E20 val      0.0200 loss,  99.7% correct, 0.4780 precision, 0.7206 recall,
➥ 0.5748 f1 score
E20 val_ben  0.0133 loss,  99.8% correct (54864 of 54971)
E20 val_mal  2.7101 loss,  72.1% correct (98 of 136)
```

Ugh. That's a lot of text to scroll past to get to the numbers we're interested in. Let's power through and focus on the `val_mal XX.X% correct` numbers (or skip ahead to the TensorBoard graph in the next section.) After epoch 2, we were at 87.5%; on epoch 5, we peaked with 92.6%; and then by epoch 20 we dropped down to 86.8%—*below* our second epoch!

> **NOTE** As mentioned earlier, expect each run to have unique behavior due to random initialization of network weights and random selection and ordering of training samples per epoch.

The training set numbers don't seem to be having the same problem. Negative training samples are classified correctly 98.8% of the time, and positive samples are 99.1% correct. What's going on?

12.4.3 Recognizing the symptoms of overfitting

What we are seeing are clear signs of overfitting. Let's take a look at the graph of our loss on positive samples, in figure 12.18.

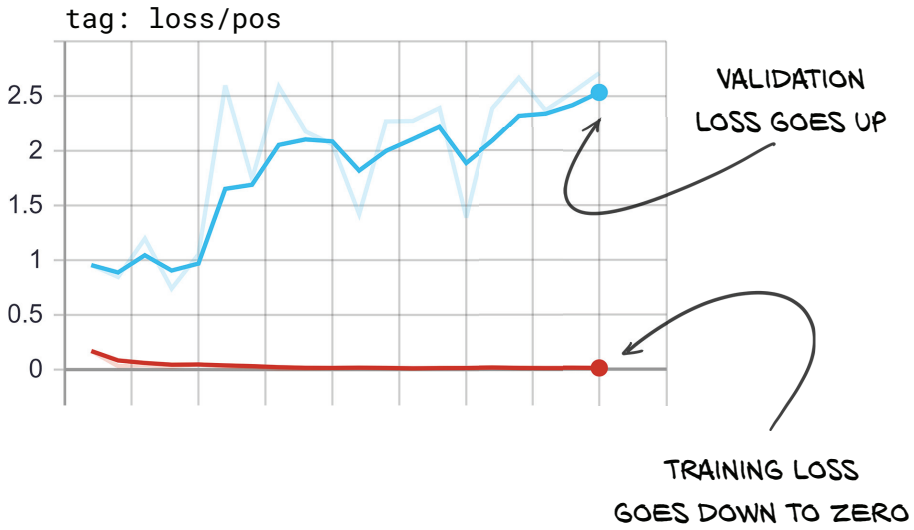


Figure 12.18 Our positive loss showing clear signs of overfitting, as the training loss and validation loss are trending in different directions

Here, we can see that the training loss for our positive samples is nearly zero—each positive training sample gets a nearly perfect prediction. Our validation loss for positive samples is *increasing*, though, and that means our real-world performance is likely getting worse. At this point, it's often best to stop the training script, since the model is no longer improving.

> **TIP** Generally, if your model's performance is improving on your training set while getting worse on your validation set, the model has started overfitting.

We must take care to examine the right metrics, however, since this trend is only happening on our *positive* loss. If we take a look at our overall loss, everything seems fine! That's because our validation set is not balanced, so the overall loss is dominated by our negative samples. As shown in figure 12.19, we are not seeing the same divergent behavior for our negative samples. Instead, our negative loss looks great! That's because we have 400 times more negative samples, so it's much, much harder for the model to remember individual details. Our positive training set has only 1,215 samples, though. While we repeat those samples multiple times, that doesn't make them harder to memorize. The model is shifting from generalized principles to essentially memorizing quirks of those 1,215 samples and claiming that anything that's not one of those few samples is negative. This includes both negative training samples and everything in our validation set (both positive and negative).

Clearly, some generalization is still going on, since we are classifying about 70% of the positive validation set correctly. We just need to change how we're training the model so that our training set and validation set both trend in the right direction.

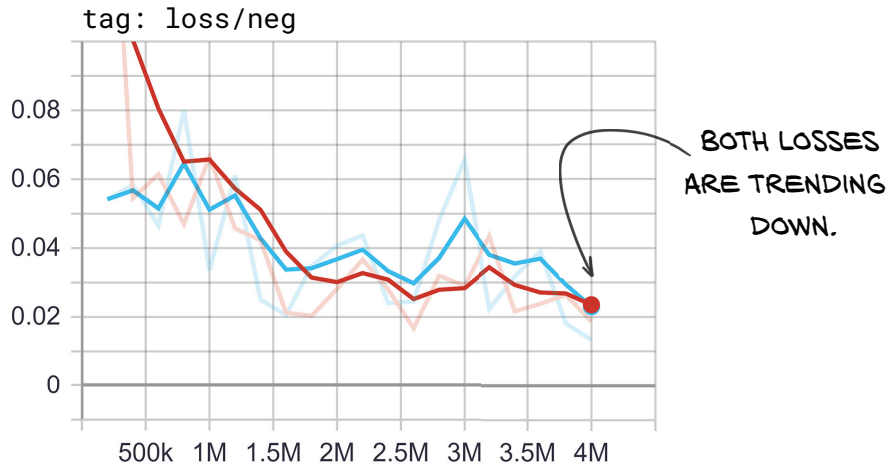


Figure 12.19 Our negative loss showing no signs of overfitting

12.5 Revisiting the problem of overfitting

We touched on the concept of overfitting in chapter 5, and now it's time to take a closer look at how to address this common situation. Our goal with training a model is to teach it to recognize the *general properties* of the classes we are interested in, as expressed in our dataset. Those general properties are present in some or all samples of the class and can be *generalized* and used to predict samples that haven't been trained on. When the model starts to learn *specific properties* of the training set, overfitting occurs, and the model starts to lose the ability to generalize. In case that's a bit too abstract, let's use another analogy.

12.5.1 An overfit face-to-age prediction model

Let's pretend we have a model that takes an image of a human face as input and outputs a predicted age in years. A good model would pick up on age signifiers like wrinkles, gray hair, hairstyle, clothing choices, and similar, and use those to build a general model of what different ages look like. When presented with a new picture, it would consider things like "conservative haircut" and "reading glasses" and "wrinkles" to conclude "around 65 years old."

An overfit model, by contrast, instead remembers specific people by remembering identifying details. "That haircut and those glasses mean it's Frank. He's 62.8 years old"; "Oh, that scar means it's Harry. He's 39.3"; and so on. When shown a new person, the model won't recognize the person and will have absolutely no idea what age to predict.

Even worse, if shown a picture of Frank Jr. (the spittin' image of his dad, at least when he's wearing his glasses!), the model will say, "I think that's Frank. He's 62.8 years old." Never mind that Junior is 25 years younger!

Overfitting is usually due to having too few training samples when compared to the ability of the model to just memorize the answers. The median human can memorize the birthdays of their immediate family but would have to resort to generalizations when predicting the ages of any group larger than a small village.

Our face-to-age model has the capacity to simply memorize the photos of anyone who doesn't look exactly their age. As we discussed in part 1, model capacity is a somewhat abstract concept, but is roughly a function of the number of parameters of the model times how efficiently those parameters are used. When a model has a high capacity relative to the amount of data needed to memorize the hard samples from the training set, it's likely that the model will begin to overfit on those more difficult training samples.

12.6 Preventing overfitting with data augmentation

It's time to take our model training from good to great. We need to cover one last step in figure 12.20.

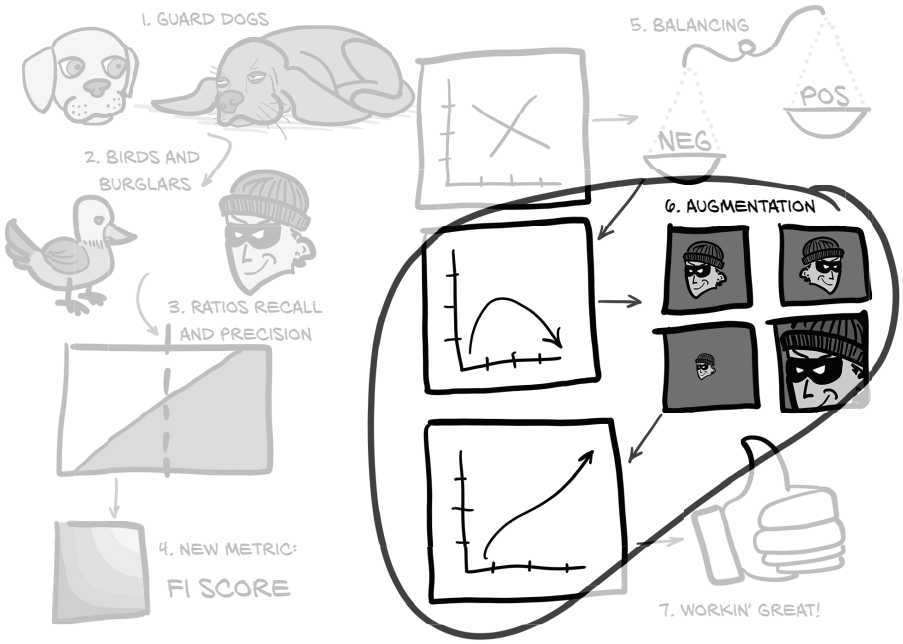


Figure 12.20 The set of topics for this chapter, with a focus on data augmentation

We *augment* a dataset by applying synthetic alterations to individual samples, resulting in a new dataset with an effective size that is larger than the original. The typical goal is for the alterations to result in a synthetic sample that remains representative of the same general class as the source sample, but that cannot be trivially memorized alongside the original. When done properly, this augmentation can increase the training set

size beyond what the model is capable of memorizing, resulting in the model being forced to increasingly rely on generalization, which is exactly what we want. Doing so is especially useful when dealing with limited data, as we saw in section 12.4.1.

Of course, not all augmentations are equally useful. Going back to our example of a face-to-age prediction model, we could trivially change the red channel of the four corner pixels of each image to a random value 0–255, which would result in a dataset 4 billion times larger the original. Of course, this wouldn't be particularly useful, since the model can pretty trivially learn to ignore the red dots in the image corners, and the rest of the image remains as easy to memorize as the single, unaugmented original image. Contrast that approach with flipping the image left to right. Doing so would only result in a dataset twice as large as the original, but each image would be quite a bit more useful for training purposes. The general properties of aging are not correlated left to right, so a mirrored image remains representative. Similarly, it's rare for facial pictures to be perfectly symmetrical, so a mirrored version is unlikely to be trivially memorized alongside the original.

12.6.1 Specific data augmentation techniques

We are going to implement five specific types of data augmentation. Our implementation will allow us to experiment with any or all of them, individually or in aggregate. The five techniques are as follows:

- Mirroring the image up-down, left-right, and/or front-back
- Shifting the image around by a few voxels
- Scaling the image up or down
- Rotating the image around the head-foot axis
- Adding noise to the image

For each technique, we want to make sure our approach maintains the training sample's representative nature, while being different enough that the sample is useful to train with.

We'll define a function `getCtAugmentedCandidate` that is responsible for taking our standard chunk-of-CT-with-candidate-inside and modifying it. Our main approach will define an affine transformation matrix (http://mng.bz/Edxq) and use it with the PyTorch `affine_grid` (https://pytorch.org/docs/stable/nn.html#affine-grid) and `grid_sample` (https://pytorch.org/docs/stable/nn.html#torch.nn.functional.grid_sample) functions to resample our candidate.

Listing 12.11 dsets.py:149, `def getCtAugmentedCandidate`

```
def getCtAugmentedCandidate(
    augmentation_dict,
    series_uid, center_xyz, width_irc,
    use_cache=True):
  if use_cache:
    ct_chunk, center_irc = \
```

```
      getCtRawCandidate(series_uid, center_xyz, width_irc)
  else:
    ct = getCt(series_uid)
    ct_chunk, center_irc = ct.getRawCandidate(center_xyz, width_irc)

  ct_t = torch.tensor(ct_chunk).unsqueeze(0).unsqueeze(0).to(torch.float32)
```

We first obtain `ct_chunk`, either from the cache or directly by loading the CT (something that will come in handy once we are creating our own candidate centers), and then convert it to a tensor. Next is the affine grid and sampling code.

Listing 12.12 dsets.py:162, `def getCtAugmentedCandidate`

```
transform_t = torch.eye(4)
# ...                                  <-- Modifications to transform_tensor will go here.
# ... line 195
affine_t = F.affine_grid(
    transform_t[:3].unsqueeze(0).to(torch.float32),
    ct_t.size(),
    align_corners=False,
  )

augmented_chunk = F.grid_sample(
    ct_t,
    affine_t,
    padding_mode='border',
    align_corners=False,
  ).to('cpu')
# ... line 214
return augmented_chunk[0], center_irc
```

Without anything additional, this function won't do much. Let's see what it takes to add in some actual transforms.

> **NOTE** It's important to structure your data pipeline such that your caching steps happen *before* augmentation! Doing otherwise will result in your data being augmented once and then persisted in that state, which defeats the purpose.

MIRRORING

When mirroring a sample, we keep the pixel values exactly the same and only change the orientation of the image. Since there's no strong correlation between tumor growth and left-right or front-back, we should be able to flip those without changing the representative nature of the sample. The index-axis (referred to as Z in patient coordinates) corresponds to the direction of gravity in an upright human, however, so there's a possibility of a difference in the top and bottom of a tumor. We are going to assume it's fine, since quick visual investigation doesn't show any gross bias. Were we working toward a clinically relevant project, we'd need to confirm that assumption with an expert.

Listing 12.13 dsets.py:165, def getCtAugmentedCandidate

```
for i in range(3):
  if 'flip' in augmentation_dict:
    if random.random() > 0.5:
      transform_t[i,i] *= -1
```

The `grid_sample` function maps the range [–1, 1] to the extents of both the old and new tensors (the rescaling happens implicitly if the sizes are different). This range mapping means that to mirror the data, all we need to do is multiply the relevant element of the transformation matrix by –1.

SHIFTING BY A RANDOM OFFSET

Shifting the nodule candidate around shouldn't make a huge difference, since convolutions are translation independent, though this will make our model more robust to imperfectly centered nodules. What will make a more significant difference is that the offset might not be an integer number of voxels; instead, the data will be resampled using trilinear interpolation, which can introduce some slight blurring. Voxels at the edge of the sample will be repeated, which can be seen as a smeared, streaky section along the border.

Listing 12.14 dsets.py:165, def getCtAugmentedCandidate

```
for i in range(3):
  # ... line 170
  if 'offset' in augmentation_dict:
    offset_float = augmentation_dict['offset']
    random_float = (random.random() * 2 - 1)
    transform_t[i,3] = offset_float * random_float
```

Note that our `'offset'` parameter is the maximum offset expressed in the same scale as the [–1, 1] range the grid sample function expects.

SCALING

Scaling the image slightly is very similar to mirroring and shifting. Doing so can also result in the same repeated edge voxels we just mentioned when discussing shifting the sample.

Listing 12.15 dsets.py:165, def getCtAugmentedCandidate

```
for i in range(3):
  # ... line 175
  if 'scale' in augmentation_dict:
    scale_float = augmentation_dict['scale']
    random_float = (random.random() * 2 - 1)
    transform_t[i,i] *= 1.0 + scale_float * random_float
```

Since `random_float` is converted to be in the range [–1, 1], it doesn't actually matter if we add `scale_float * random_float` to or subtract it from 1.0.

ROTATING

Rotation is the first augmentation technique we're going to use where we have to carefully consider our data to ensure that we don't break our sample with a conversion that causes it to no longer be representative. Recall that our CT slices have uniform spacing along the rows and columns (X- and Y-axes), but in the index (or Z) direction, the voxels are non-cubic. That means we can't treat those axes as interchangeable.

One option is to resample our data so that our resolution along the index-axis is the same as along the other two, but that's not a true solution because the data along that axis would be very blurry and smeared. Even if we interpolate more voxels, the fidelity of the data would remain poor. Instead, we'll treat that axis as special and confine our rotations to the X-Y plane.

Listing 12.16 dsets.py:181, def getCtAugmentedCandidate

```
if 'rotate' in augmentation_dict:
  angle_rad = random.random() * math.pi * 2
  s = math.sin(angle_rad)
  c = math.cos(angle_rad)

  rotation_t = torch.tensor([
    [c, -s, 0, 0],
    [s, c, 0, 0],
    [0, 0, 1, 0],
    [0, 0, 0, 1],
  ])

  transform_t @= rotation_t
```

NOISE

Our final augmentation technique is different from the others in that it is actively destructive to our sample in a way that flipping or rotating the sample is not. If we add too much noise to the sample, it will swamp the real data and make it effectively impossible to classify. While shifting and scaling the sample would do something similar if we used extreme input values, we've chosen values that will only impact the edge of the sample. Noise will have an impact on the entire image.

Listing 12.17 dsets.py:208, def getCtAugmentedCandidate

```
if 'noise' in augmentation_dict:
  noise_t = torch.randn_like(augmented_chunk)
  noise_t *= augmentation_dict['noise']

  augmented_chunk += noise_t
```

The other augmentation types have increased the effective size of our dataset. Noise makes our model's job *harder*. We'll revisit this once we see some training results.

Examining augmented candidates

We can see the result of our efforts in figure 12.21. The upper-left image shows an unaugmented positive candidate, and the next five show the effect of each augmentation type in isolation. Finally, the bottom row shows the combined result three times.

Since each `__getitem__` call to the augmenting dataset reapplies the augmentations randomly, each image on the bottom row looks different. This also means it's nearly impossible to generate an image exactly like this again! It's also important to

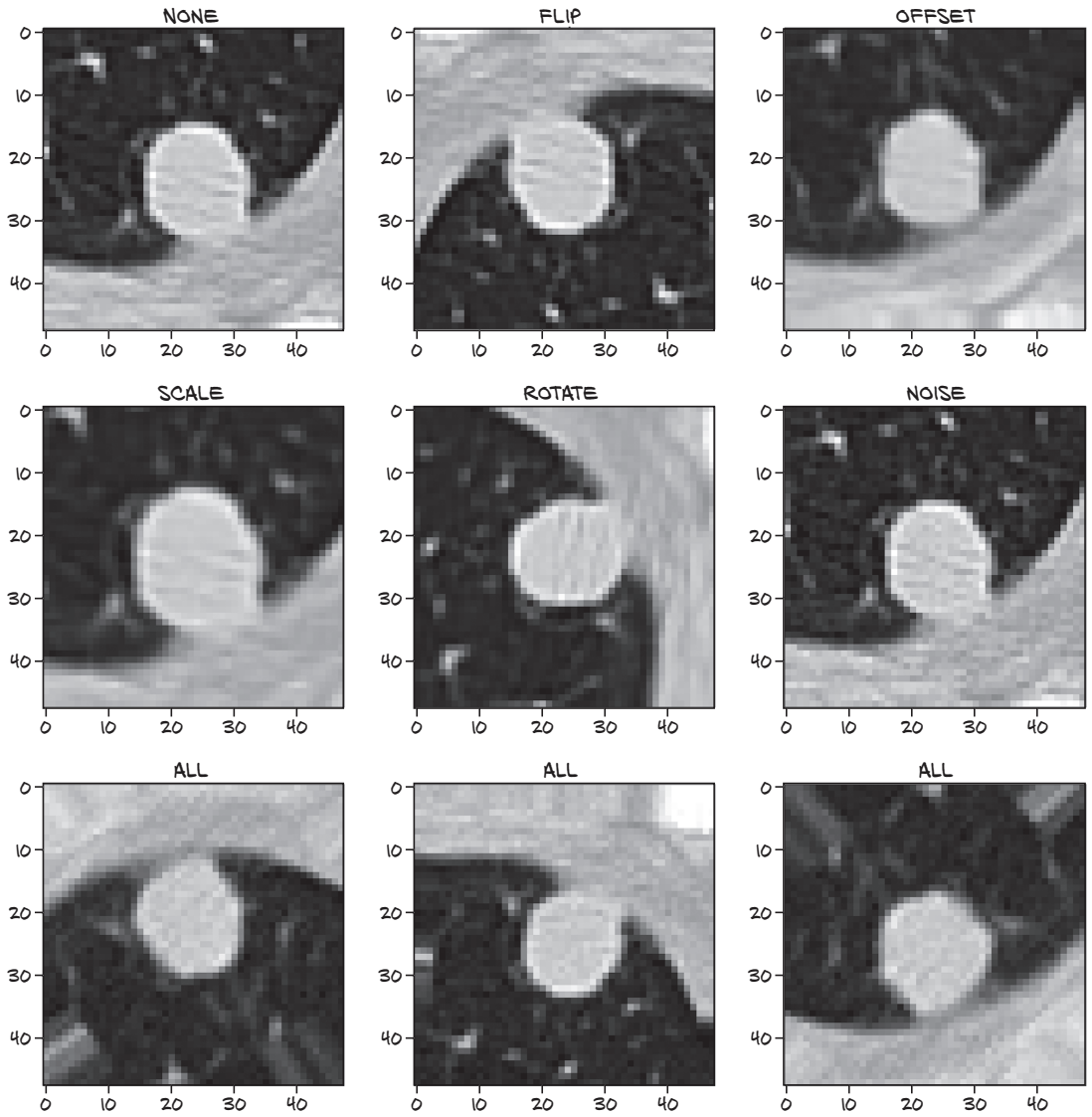


Figure 12.21 Various augmentation types performed on a positive nodule sample

remember that sometimes the `'flip'` augmentation will result in *no* flip. Returning always-flipped images is just as limiting as not flipping in the first place. Now let's see if any of this makes a difference.

12.6.2 Seeing the improvement from data augmentation

We are going to train additional models, one per augmentation type discussed in the last section, with an additional model training run that combines all of the augmentation types. Once they're finished, we'll take a look at our numbers in TensorBoard.

In order to be able to turn our new augmentation types on and off, we need to expose the construction of `augmentation_dict` to our command-line interface. Arguments to our program will be added by `parser.add_argument` calls (not shown, but similar to the ones our program already has), which will then be fed into code that actually constructs `augmentation_dict`.

Listing 12.18 training.py:105, `LunaTrainingApp.__init__`

```
self.augmentation_dict = {}
if self.cli_args.augmented or self.cli_args.augment_flip:
  self.augmentation_dict['flip'] = True
if self.cli_args.augmented or self.cli_args.augment_offset:
  self.augmentation_dict['offset'] = 0.1
if self.cli_args.augmented or self.cli_args.augment_scale:
  self.augmentation_dict['scale'] = 0.2
if self.cli_args.augmented or self.cli_args.augment_rotate:
  self.augmentation_dict['rotate'] = True
if self.cli_args.augmented or self.cli_args.augment_noise:
  self.augmentation_dict['noise'] = 25.0
```

These values were empirically chosen to have a reasonable impact, but better values probably exist.

Now that we have those command-line arguments ready, you can either run the following commands or revisit p2_run_everything.ipynb and run cells 8 through 16. Either way you run it, expect these to take a significant time to finish:

```
$ .venv/bin/python -m p2ch12.prepcache

$ .venv/bin/python -m p2ch12.training --epochs 20 \
        --balanced sanity-bal

$ .venv/bin/python -m p2ch12.training --epochs 10 \
        --balanced --augment-flip   sanity-bal-flip

$ .venv/bin/python -m p2ch12.training --epochs 10 \
        --balanced --augment-shift  sanity-bal-shift

$ .venv/bin/python -m p2ch12.training --epochs 10 \
        --balanced --augment-scale  sanity-bal-scale

$ .venv/bin/python -m p2ch12.training --epochs 10 \
        --balanced --augment-rotate sanity-bal-rotate

$ .venv/bin/python -m p2ch12.training --epochs 10 \
```

You only need to prep the cache once per chapter.

You might have this run from earlier in the chapter; in that case there's no need to rerun it!

```
        --balanced --augment-noise  sanity-bal-noise

$ .venv/bin/python -m p2ch12.training --epochs 20 \
        --balanced --augmented sanity-bal-aug
```

While that's running, we can start TensorBoard. Let's direct it to only show these runs by changing the `logdir` parameter like so: `../path/to/tensorboard --logdir runs/p2ch12`.

Depending on the hardware you have at your disposal, the training might take a long time. Feel free to skip the `flip`, `shift`, and `scale` training jobs and reduce the first and last runs to 11 epochs if you need to move things along more quickly. We chose 20 runs because that helps them stand out from the other runs, but 11 should work as well.

If you let everything run to completion, your TensorBoard should have data like that shown in figure 12.22. We're going to deselect everything except the validation data, to reduce clutter. When you're looking at your data live, you can also change the smoothing value, which can help clarify the trend lines. Take a quick look at the figure, and then we'll go over it in some detail.

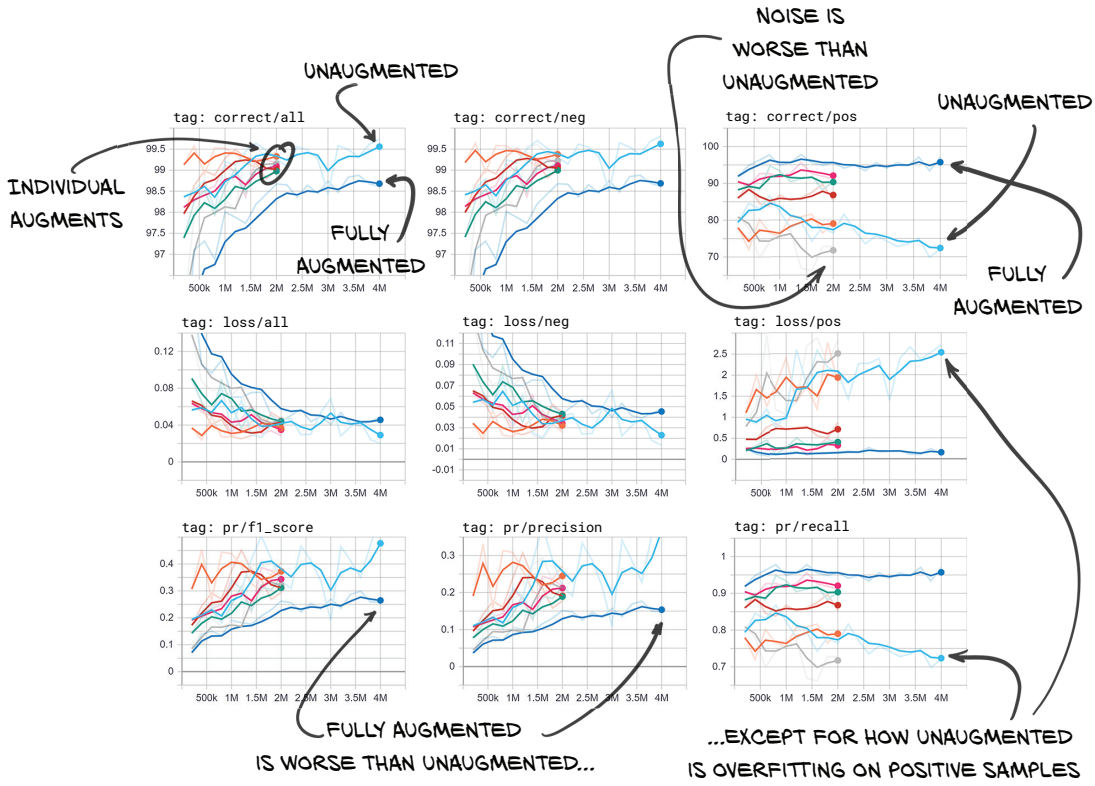


Figure 12.22 Percent correctly classified, loss, F1 score, precision, and recall for the validation set from networks trained with a variety of augmentation schemes

The first thing to notice in the upper-left graph ("tag: correct/all") is that the individual augmentation types are something of a jumble. Our unaugmented and fully augmented runs are on opposite sides of that jumble. That means when combined, our augmentation is more than the sum of its parts. Also of interest is that our fully augmented run gets many more wrong answers. While that's bad generally, if we look at the right column of images (which focus on the positive candidate samples we actually care about—the ones that are really nodules), we see that our fully augmented model is *much* better at finding the positive candidate samples. The recall for the fully augmented model is great! It's also much better at not overfitting. As we saw earlier, our unaugmented model gets worse over time.

One interesting thing to note is that the noise-augmented model is *worse* at identifying nodules than the unaugmented model. This makes sense if we remember that we said noise makes the model's job harder.

Another interesting thing to see in the live data (it's somewhat lost in the jumble here) is that the rotation-augmented model is nearly as good as the fully augmented model when it comes to recall, and it has much better precision. Since our F1 score is precision limited (due to the higher number of negative samples), the rotation-augmented model also has a better F1 score.

We'll stick with the fully augmented model going forward, since our use case requires high recall. The F1 score will still be used to determine which epoch to save as the best. In a real-world project, we might want to devote extra time to investigating whether a different combination of augmentation types and parameter values could yield better results.

12.7 Conclusion

We spent a lot of time and energy in this chapter reformulating how we think about our model's performance. It's easy to be misled by poor methods of evaluation, and it's crucial to have a strong intuitive understanding of the factors that feed into evaluating a model well. Once those fundamentals are internalized, it's much easier to spot when we're being led astray.

We've also learned about how to deal with data sources that aren't sufficiently populated. Being able to synthesize representative training samples is incredibly useful. Situations where we have too much training data are rare indeed!

Now that we have a classifier that is performing reasonably, we'll turn our attention to automatically finding candidate nodules to classify. Chapter 13 will start there; then, in chapter 14, we will feed those candidates back into the classifier we developed here and venture into building one more classifier to tell malignant nodules from benign ones.

12.8 Exercises

1. The F1 score can be generalized to support values other than 1.
   a. Read https://en.wikipedia.org/wiki/F1_score, and implement F2 and F0.5 scores.
   b. Determine which of F1, F2, and F0.5 makes the most sense for this project. Track that value, and compare and contrast it with the F1 score. [6]
2. Implement a `WeightedRandomSampler` approach to balancing the positive and negative training samples for `LunaDataset` with `ratio_int` set to `0`.
   a. How did you get the required information about the class of each sample?
   b. Which approach was easier? Which resulted in more readable code?
3. Experiment with different class-balancing schemes.
   a. What ratio results in the best score after two epochs? After 20?
   b. What if the ratio is a function of `epoch_ndx`?
4. Experiment with different data augmentation approaches.
   a. Can any of the existing approaches be made more aggressive (noise, offset, and so on)?
   b. Does the inclusion of noise augmentation help or hinder your training results?
      – Are there other values that change this result?
   c. Research data augmentation that other projects have used. Are any applicable here?
      – Implement "mixup" augmentation for positive nodule candidates. Does it help?
5. Change the initial normalization from `nn.BatchNorm` to something custom, and retrain the model.
   a. Can you get better results using fixed normalization?
   b. What normalization offset and scale make sense?
   c. Do nonlinear normalizations like square roots help?
6. What other kinds of data can TensorBoard display besides those we've covered here?
   a. Can you have it display information about the weights of your network?
   b. What about intermediate results from running your model on a particular sample?
      – Does having the backbone of the model wrapped in an instance of `nn.Sequential` help or hinder this effort?

[6] Yep, that's a hint it's not the F1 score!

12.9 Summary

- A binary label and a binary classification threshold combine to partition the dataset into four quadrants: true positives, true negatives, false negatives, and false positives. These four quantities provide the basis for our improved performance metrics.
- Recall is the ability of a model to maximize true positives. Selecting every single item guarantees perfect recall—because all the correct answers are included—but also exhibits poor precision.
- Precision is the ability of a model to minimize false positives. Selecting nothing guarantees perfect precision—because no incorrect answers are included—but also exhibits poor recall.
- The F1 score combines precision and recall into a single metric that describes model performance. We use the F1 score to determine what impact changes to training or the model have on our performance.
- Balancing the training set to have an equal number of positive and negative samples during training can result in the model performing better (defined as having a positive, increasing F1 score).
- Data augmentation takes existing organic data samples and modifies them such that the resulting augmented sample is non-trivially different from the original, but remains representative of samples of the same class. This allows additional training without overfitting in situations where data is limited.
- Common data augmentation strategies include changes in orientation, mirroring, rescaling, shifting by an offset, and adding noise. Depending on the project, other more specific strategies may also be relevant.

13 Using segmentation to find suspected nodules

This chapter covers

- Segmenting data with a pixel-to-pixel model
- Performing segmentation with U-Net
- Understanding mask prediction using Dice loss
- Evaluating a segmentation model's performance

In the last four chapters, we have accomplished a lot. We've learned about CT scans and lung tumors, datasets and data loaders, and metrics and monitoring. We have also *applied* many of the things we learned in part 1, and we have a working classifier. We are still operating in a somewhat artificial environment, however, since we require hand-annotated nodule candidate information to load into our classifier. We don't have a good way to create that input automatically. Just feeding the entire CT into our model—that is, plugging in overlapping $32 \times 32 \times 32$ patches of data—would result in $31 \times 31 \times 7 = 6{,}727$ patches per CT, or about 10 times the number of annotated samples we have. We'd need to overlap the edges; our classifier expects the nodule candidate to be centered, and even then the inconsistent positioning would probably present issues.

As we explained in chapter 9, our project uses multiple steps to solve the problem of locating possible nodules, identifying them, with an indication of their possible malignancy. This is a common approach among practitioners, while in deep learning research there is a tendency to demonstrate the ability of individual models to solve complex problems in an end-to-end fashion. The multistage project design we use in this book gives us a good excuse to introduce new concepts step by step.

13.1 Adding a second model to our project

In the previous two chapters, we worked on step 4 of our plan shown in figure 13.1: classification. In this chapter, we'll go back not just one but two steps. We need to find a way to tell our classifier where to look. To do this, we are going to take raw CT scans and find everything that might be a nodule.[1] This is the highlighted step 2 in the figure. To find these possible nodules, we have to flag voxels that look like they might be part of a nodule, a process known as *segmentation*. Then, in chapter 14, we will deal with step 3 and provide the bridge by transforming the segmentation masks from this image into location annotations.

By the time we're finished with this chapter, we'll have created a new model with an architecture that can perform per-pixel labeling, or segmentation. The code that

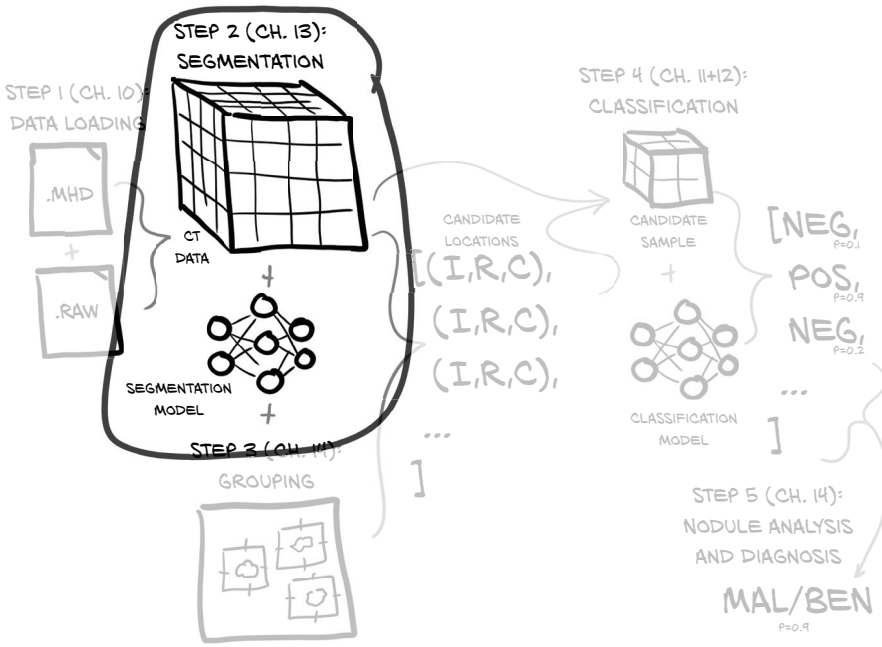


Figure 13.1 Our end-to-end lung cancer detection project, with a focus on this chapter's topic: step 2, segmentation

[1] We expect to mark quite a few things that are not nodules; thus, we use the classification step to reduce the number of these.

will accomplish this will be very similar to the code from the last chapter, especially if we focus on the larger structure. All of the changes we're going to make will be smaller and targeted. As we see in figure 13.2, we need to make updates to our model (step 2A in the figure), dataset (2B), and training loop (2C) to account for the new model's inputs, outputs, and other requirements. (Don't worry if you don't recognize each component in each of these steps in step 2 on the right side of the diagram. We'll go through the details when we get to each step.) Finally, we'll examine the results we get when running our new model (step 3 in the figure).

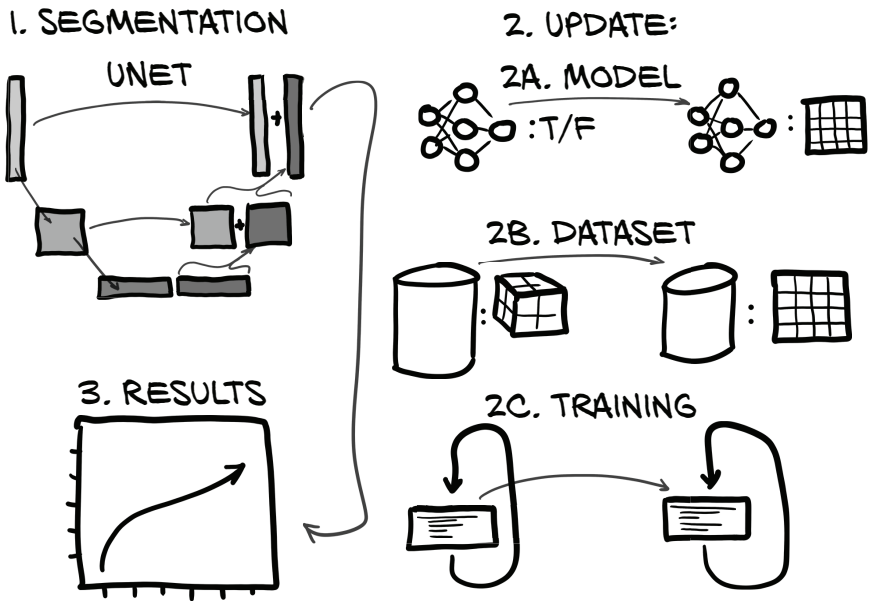


Figure 13.2 The new model architecture for segmentation, along with the model, dataset, and training loop updates we will implement

Breaking down figure 13.2 into steps, our plan for this chapter is as follows:

1. *Segmentation.* First we will learn how segmentation works with a U-Net model, including what the new model components are and what happens to them as we go through the segmentation process. This is step 1 in figure 13.2.
2. *Update.* To implement segmentation, we need to change our existing code base in three main places, shown in the substeps on the right side of figure 13.2.The code will be structurally very similar to what we developed for classification, but will differ in detail:
    a. *Update the model (step 2A).* We will integrate a preexisting U-Net into our segmentation model. Our model in chapter 12 output a simple true/false classification; our model in this chapter will instead output an entire image.

b. *Change the dataset (step 2B).* We need to change our dataset to not only deliver bits of the CT but also provide masks for the nodules. The classification dataset consisted of 3D crops around nodule candidates, but we'll need to collect both full CT slices and 2D crops for segmentation training and validation.
c. *Adapt the training loop (step 2C).* We need to adapt the training loop so we bring in a new loss to optimize. Because we want to display images of our segmentation results in TensorBoard, we'll also do things like saving our model weights to disk.

3. *Results.* Finally, we'll see the fruits of our efforts when we look at the quantitative segmentation results.

13.2 Various types of segmentation

To get started, we need to talk about different flavors of segmentation. For this project, we will be using *semantic* segmentation, which is the act of classifying individual pixels in an image using labels just like those we've seen for our classification tasks, for example, "bear," "cat," "dog," and so on. If done properly, this will result in distinct chunks or regions that signify things like "all of these pixels are part of a cat." This takes the form of a label mask or heatmap that identifies areas of interest. We will have a simple binary label: true values will correspond to nodule candidates, and false values mean uninteresting healthy tissue. This partially meets our need to find nodule candidates that we will later feed into our classification network.

Before we get into the details, we should briefly discuss other approaches we could take to finding our nodule candidates. For example, *instance segmentation* labels individual objects of interest with distinct labels. So whereas semantic segmentation would label a picture of two people shaking hands with two labels ("person" and "background"), instance segmentation would have three labels ("person1," "person2," and "background") with a boundary somewhere around the clasped hands. While this could be useful for us to distinguish "nodule1" from "nodule2," we will instead use grouping to identify individual nodules. That approach will work well for us since nodules are unlikely to touch or overlap.

Another approach to these kinds of tasks is *object detection*, which locates an item of interest in an image and puts a bounding box around the item. While both instance segmentation and object detection could be great for our uses, their implementations are somewhat complex, and we don't feel they are the best things for you to learn next. Also, training object-detection models typically requires much more computational resources than our approach requires. If you're feeling up to the challenge, the YOLOv3 paper is a more entertaining read than most deep learning research papers.[2] For us, though, semantic segmentation it is.

[2] Joseph Redmon and Ali Farhadi, "YOLOv3: An Incremental Improvement," https://pjreddie.com/media/files/papers/YOLOv3.pdf. Perhaps check it out once you've finished the book.

NOTE As we go through the code examples in this chapter, we're going to rely on you checking the code from GitHub for much of the larger context. We'll be omitting code that's uninteresting or similar to what's come before in earlier chapters, so that we can focus on the crux of the issue at hand.

13.3 Semantic segmentation: Per-pixel classification

Often, segmentation is used to answer questions of the form "Where is a cat in this picture?" Obviously, most pictures of a cat, like figure 13.3, have a lot of non-cat in them; there's the table or wall in the background, the keyboard the cat is sitting on, that kind of thing. Being able to say "This pixel is part of the cat, and this other pixel is part of the wall" requires fundamentally different model output and a different internal structure from the classification models we've worked with thus far. Classification can tell us whether a cat is present, while segmentation will tell us where we can find it.

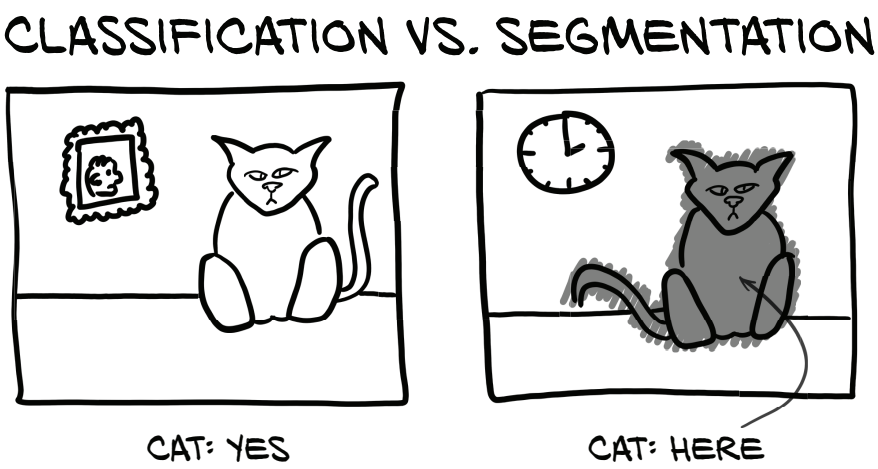


Figure 13.3 Classification results in one or more binary flags, while segmentation produces a mask or heatmap.

If your project requires differentiating between a near cat and a far cat, or a cat on the left versus a cat on the right, then segmentation is probably the right approach. The image-consuming classification models that we've implemented so far can be thought of as funnels or magnifying glasses that take a large bunch of pixels and focus them down into a single "point" (or, more accurately, a single set of class predictions), as shown in figure 13.4. Classification models provide answers of the form "Yes, this huge pile of pixels has a cat in it, somewhere," or "No, no cats here." This is great when you don't care where the cat is, just that there is (or isn't) one in the image.

Repeated layers of convolution and downsampling mean the model starts by consuming raw pixels to produce specific, detailed detectors for things like texture and color, and then builds up higher-level conceptual feature detectors for parts like eyes

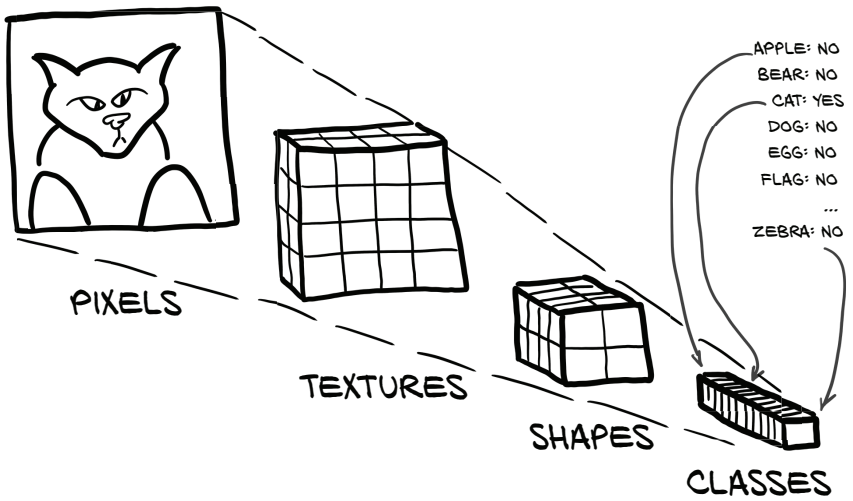


Figure 13.4 The magnifying glass model structure for classification

and ears and mouth and nose[3] that finally result in "cat" versus "dog." Due to the increasing receptive field of the convolutions after each downsampling layer, those higher-level detectors can use information from an increasingly large area of the input image.

Unfortunately, since segmentation needs to produce an image-like output, ending up at a single classification-like list of binary-ish flags won't work. As we recall from section 11.4, downsampling is key to increasing the receptive fields of the convolutional layers, and is what helps reduce the array of pixels that make up an image to a single list of classes. Notice figure 13.5, which repeats figure 11.6.

In the figure, our inputs flow from the left to right in the top row and are continued in the bottom row. In order to work out the receptive field—the area influencing the single pixel at bottom right—we can go backward. The max-pool operation has 2 × 2 inputs producing each final output pixel. The 3 × 3 conv in the middle of the bottom row looks at one adjacent pixel (including diagonally) in each direction, so the total receptive field of the convolutions that result in the 2 x 2 output is 4 x 4 (with the right "x" characters). The 3 × 3 convolution in the top row then adds an additional pixel of context in each direction, so the receptive field of the single output pixel at bottom right is a 6 × 6 field in the input at top left. With the downsampling from the max pool, the receptive field of the next block of convolutions will have double the width, and each additional downsampling will double it again, while shrinking the size of the output.

We'll need a different model architecture if we want our output to be the same size as our input. One simple model to use for segmentation would have repeated convolutional layers without any downsampling. Given appropriate padding, that would result in output the same size as the input (good), but a very limited receptive field

[3] … "head, shoulders, knees, and toes, knees and toes," as my (Eli's) toddlers would sing.

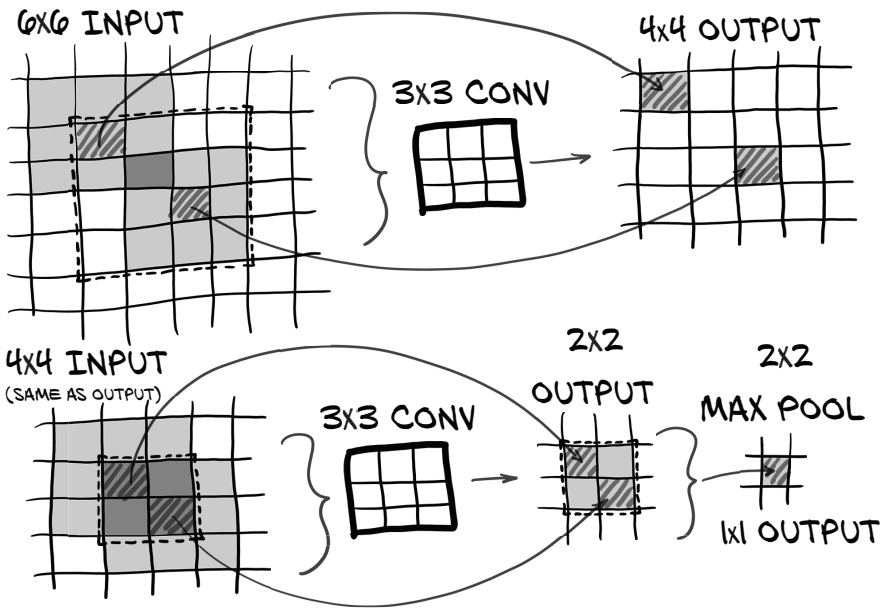


Figure 13.5 The convolutional architecture of a `LunaModel` block, consisting of two 3 × 3 convolutions followed by a max pool. The final pixel has a 6 × 6 receptive field.

(bad) due to the limited reach based on how much overlap multiple layers of small convolutions will have. The classification model uses each downsampling layer to double the effective reach of the following convolutions; and without that increase in effective field size, each segmented pixel will only be able to consider a very local neighborhood.

> **NOTE** Assuming 3 × 3 convolutions, the receptive field size for a simple model of stacked convolutions is $2 * L + 1$, with L being the number of convolutional layers.

Four layers of 3 × 3 convolutions will have a receptive field of 9 × 9 per output pixel. By inserting a 2 × 2 max pool between the second and third convolutions, and another at the end, we increase the receptive field to …

> **NOTE** See if you can figure out the math yourself; when you're done, check back here.

... 16 × 16. The final series of conv-conv-pool has a receptive field of 6 × 6, but that happens *after* the first max pool, which makes the final effective receptive field 12 × 12 in the original input resolution. The first two conv layers add a total border of 2 pixels around the 12 × 12, for a total of 16 × 16.

So the question remains: how can we improve the receptive field of an output pixel while maintaining a 1:1 ratio of input pixels to output pixels? One common answer is

to use a technique called *upsampling*, which takes an image of a given resolution and produces an image of a higher resolution. Upsampling at its simplest just means replacing each pixel with an $N \times N$ block of pixels, each with the same value as the original input pixel. The possibilities only get more complex from there, with options like linear interpolation and learned deconvolution.

13.3.1 The U-Net architecture

Before we end up diving down a rabbit hole of possible upsampling algorithms, let's get back to our goal for the chapter. Per figure 13.6, step 1 is to get familiar with a foundational segmentation algorithm called U-Net.

The U-Net architecture is a design for a neural network that can produce pixel-wise output and that was invented for segmentation. As you can see from the highlight in figure 13.6, a diagram of the U-Net architecture looks a bit like the letter *U*, which explains the origins of the name. We also immediately see that it is quite a bit more complicated than the mostly sequential structure of the classifiers we are familiar with. We'll see a more detailed version of the U-Net architecture shortly, in figure 13.7, and learn exactly what each of those components is doing. Once we understand the model architecture, we can work on training one to solve our segmentation task.

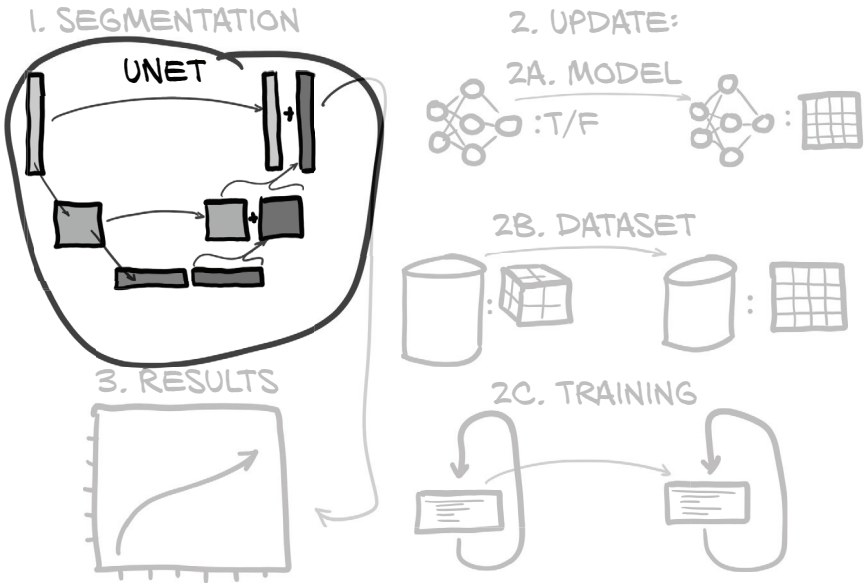


Figure 13.6 The new model architecture for segmentation, that we will be working with

The U-Net architecture shown in figure 13.7 was an early breakthrough for image segmentation. Let's take a look and then walk through the architecture.

In this diagram, the boxes represent intermediate results and the arrows represent operations between them. The U-shape of the architecture comes from the multiple

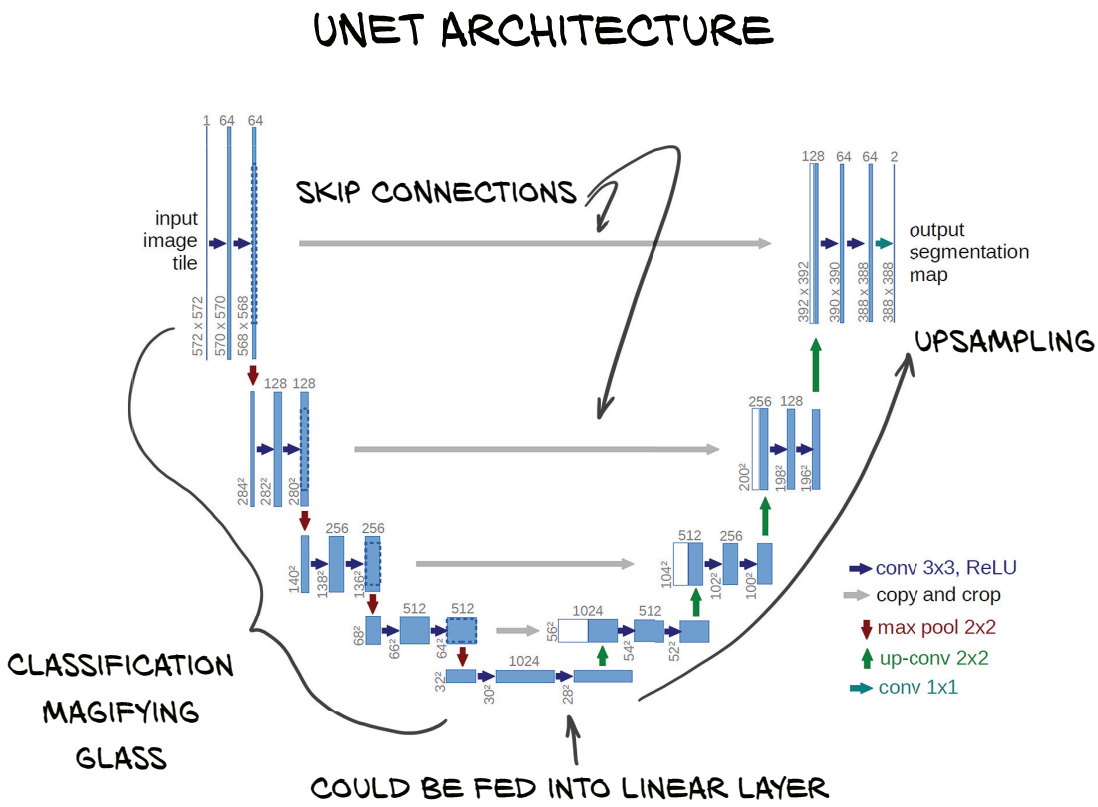


Figure 13.7 From the U-Net paper, with annotations. Source: The base of this figure is courtesy Olaf Ronneberger et al., from the paper "U-Net: Convolutional Networks for Biomedical Image Segmentation," which can be found at https://arxiv.org/abs/1505.04597 and https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net.

resolutions at which the network operates. In the top row is the full resolution (512 × 512 for us), the row below has half that, and so on. The data flows from top left to bottom center through a series of convolutions and downscaling, as we saw in the classifiers and looked at in detail in chapter 8. Then we go up again, using upscaling convolutions to get back to the full resolution. Unlike the original U-Net, we will be padding things so we don't lose pixels off the edges, so our resolution is the same on the left and on the right.

Earlier network designs already had this U-shape, which people attempted to use to address the limited receptive field size of fully convolutional networks. To address this limited field size, they used a design that copied, inverted, and appended the focusing portions of an image-classification network to create a symmetrical model that goes from fine detail to wide receptive field and back to fine detail.

Those earlier network designs had problems converging, however, most likely due to the loss of spatial information during downsampling. Once information reaches a large number of very downscaled images, the exact location of object boundaries gets

harder to encode and therefore reconstruct. To address this, the U-Net authors added the skip connections we see at the center of the figure. We first touched on skip connections in chapter 8, although they are employed differently here than in the ResNet architecture. In U-Net, skip connections short-circuit inputs along the downsampling path into the corresponding layers in the upsampling path. These layers receive as input both the upsampled results of the wide receptive field layers from lower in the U as well as the output of the earlier fine detail layers via the "copy and crop" bridge connections. This is the key innovation behind U-Net (which, interestingly, predated ResNet).

All of this means those final detail layers are operating with the best of both worlds. They've got both information about the larger context surrounding the immediate area and fine detail data from the first set of full-resolution layers.

The "conv 1x1" layer at far right, in the head of the network, changes the number of channels from 64 to 2 (the original paper had 2 output channels; we have 1 in our case). This is somewhat akin to the fully connected layer we used in our classification network, but per-pixel, channel-wise: it's a way to convert from the number of filters used in the last upsampling step to the number of output classes needed.

13.4 Updating the model for segmentation

It's time to move through step 2A in figure 13.8. We've had enough theory about segmentation and history about U-Net; now we want to update our code, starting with the model. Instead of just outputting a binary classification that gives us a single output of true or false, we integrate a U-Net to get to a model that's capable of outputting a

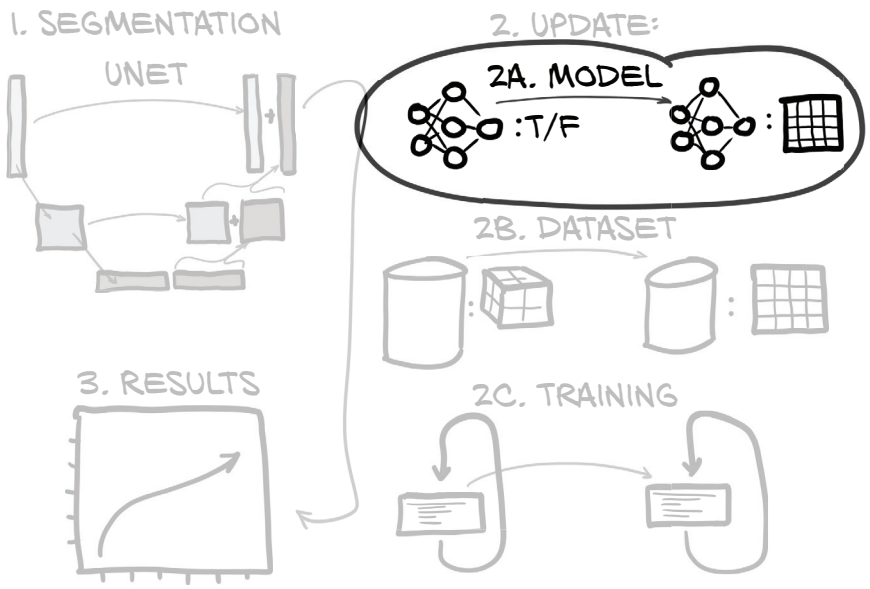


Figure 13.8 The outline of this chapter, with a focus on the changes needed for our segmentation model

probability for every pixel: that is, performing segmentation. Rather than implementing a custom U-Net segmentation model from scratch, we're going to appropriate an existing implementation from an open source repository on GitHub.

The U-Net implementation at https://github.com/jvanvugt/pytorch-unet seems to meet our needs well.[4] It's MIT licensed (copyright 2018 Joris), it's contained in a single file, and it has a number of parameter options for us to tweak. The file is included in our code repository at util/unet.py, along with a link to the original repository and the full text of the license used.

> **NOTE** While it's less of an issue for personal projects, it's important to be aware of the license terms attached to open source software you use for a project. The MIT license is one of the most permissive open source licenses, and it still places requirements on users of MIT licensed code! Also be aware that authors retain copyright even if they publish their work in a public forum (yes, even on GitHub), and if they do not include a license, that does *not* mean the work is in the public domain. Quite the opposite! It means you don't have *any* license to use the code, any more than you'd have the right to wholesale copy a book you borrowed from the library.

We suggest taking some time to inspect the code and, based on the knowledge you have built up until this point, identify the building blocks of the architecture as they are reflected in the code. Can you spot skip connections? A particularly worthy exercise for you is to draw a diagram that shows how the model is laid out, just by looking at the code.

Now that we have found a U-Net implementation that fits the bill, we need to adapt it so that it works well for our needs. In general, it's a good idea to keep an eye out for situations where we can use something off the shelf. It's important to have a sense of what models exist, how they're implemented and trained, and whether any parts can be scavenged and applied to the project we're working on at any given moment. While that broader knowledge is something that comes with time and experience, it's a good idea to start building that toolbox now.

13.4.1 Adapting an off-the-shelf model to our project

We will now make some changes to the classic U-Net, justifying them along the way. A useful exercise for you will be to compare results between the *vanilla* model and the one after the tweaks, preferably removing one at a time to see the effect of each change (this is also called an *ablation study* in research circles).

First, we're going to pass the input through batch normalization. This way, we won't have to normalize the data ourselves in the dataset; and, more importantly, we will get normalization statistics (read mean and standard deviation) estimated over individual batches. This means when a batch is *dull* for some reason—that is, when there is nothing to see in all the CT crops fed into the network—it will be scaled more

[4] The implementation included here differs from the official paper by using average pooling instead of max pooling to downsample. The most recent version on GitHub has changed to use max pool.

strongly. The fact that samples in batches are picked randomly at every epoch will minimize the chances of a dull sample ending up in an all-dull batch, and hence those dull samples getting overemphasized.

Second, since the output values are unconstrained, we are going to pass the output through an `nn.Sigmoid` layer to restrict the output to the range [0, 1]. Third, we will reduce the total depth and number of filters we allow our model to use. While this is jumping ahead of ourselves a bit, the capacity of the model using the standard parameters far outstrips our dataset size. This means we're unlikely to find a pretrained model that matches our exact needs. Finally, although this is not a modification, it's important to note that our output is a single channel, with each pixel of output representing the model's estimate of the probability that the pixel in question is part of a nodule.

This wrapping of U-Net can be done rather simply by implementing a model with three attributes: one each for the two features we want to add, and one for the U-Net itself—which we can treat just like any prebuilt module here. We will also pass any keyword arguments we receive into the U-Net constructor.

Listing 13.1 model.py:17, `class UNetWrapper`

kwarg is a dictionary containing all keyword arguments passed to the constructor.

BatchNorm2d wants us to specify the number of input channels, which we take from the keyword argument.

The U-Net: a small thing to include here, but it's really doing all the work.

Just as for the classifier in chapter 11, we use our custom weight initialization. The function is copied over, so we will not show the code again.

```
class UNetWrapper(nn.Module):
  def __init__(self, **kwargs):
    super().__init__()

    self.input_batchnorm = nn.BatchNorm2d(kwargs['in_channels'])
    self.unet = UNet(**kwargs)
    self.final = nn.Sigmoid()

    self._init_weights()
```

The `forward` method is a similarly straightforward sequence. We could use an instance of `nn.Sequential` as we saw in chapter 8, but we'll be explicit here for both clarity of code and clarity of stack traces.[5]

Listing 13.2 model.py:50, `UNetWrapper.forward`

```
def forward(self, input_batch):
  bn_output = self.input_batchnorm(input_batch)
  un_output = self.unet(bn_output)
  fn_output = self.final(un_output)
  return fn_output
```

Note that we're using `nn.BatchNorm2d` here. This is because U-Net is fundamentally a two-dimensional segmentation model. We could adapt the implementation to use 3D

[5] In the unlikely event our code throws any exceptions—which it clearly won't, will it?

convolutions, in order to use information across slices. The memory usage of a straightforward implementation would be considerably greater: that is, we would have to chop up the CT scan. Also, the fact that pixel spacing in the Z direction is much larger than in-plane makes a nodule less likely to be present across many slices. These considerations make a fully 3D approach less attractive for our purposes. Instead, we'll adapt our 3D data to be segmented a slice at a time, providing adjacent slices for context (for example, detecting that a bright lump is indeed a blood vessel gets much easier alongside neighboring slices). Since we're sticking with presenting the data in 2D, we'll use channels to represent the adjacent slices. Our treatment of the third dimension is similar to how we applied a fully connected model to images in chapter 7: the model will have to relearn the adjacency relationships we're throwing away along the axial direction, but that's not difficult for the model to accomplish, especially with the limited number of slices given for context owing to the small size of the target structures.

13.5 Updating the dataset for segmentation

Our source data for this chapter remains unchanged: we're consuming CT scans and annotation data about them. But our model expects input and will produce output of a different form than we had previously. As we hint at in step 2B of figure 13.9, our previous dataset produced 3D data, but we need to produce 2D data now.

The original U-Net implementation did not use padded convolutions, which means while the output segmentation map was smaller than the input, every pixel of that output had a fully populated receptive field. None of the input pixels that fed

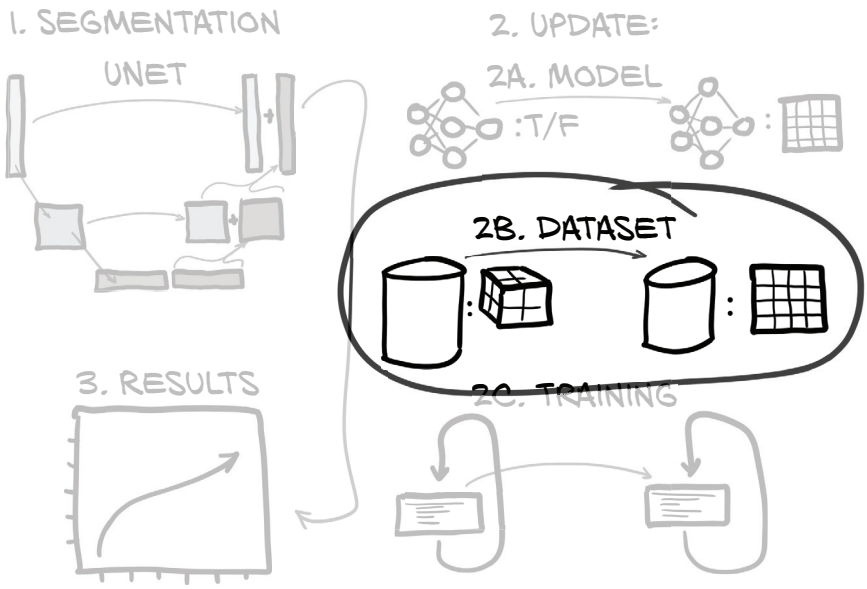


Figure 13.9 The outline of this chapter, with a focus on the changes needed for our segmentation dataset

into the determination of that output pixel were padded, fabricated, or otherwise incomplete. Thus the output of the original U-Net will tile perfectly, so it can be used with images of any size (except at the edges of the input image, where some context will be missing by definition).

There are two problems with us taking the same pixel-perfect approach for our problem. The first is related to the interaction between convolution and downsampling, and the second is related to the nature of our data being three-dimensional.

13.5.1 U-Net has very specific input size requirements

The first issue is that the sizes of the input and output patches for U-Net are very specific. In order to have the two-pixel loss per convolution line up evenly before and after downsampling (especially when considering the further convolutional shrinkage at that lower resolution), only certain input sizes will work. The U-Net paper used 572 × 572 image patches, which resulted in 388 × 388 output maps. The input images are bigger than our 512 × 512 CT slices, and the output is quite a bit smaller! That would mean any nodules near the edge of the CT scan slice wouldn't be segmented at all. Although this setup works well when dealing with very large images, it's not ideal for our use case.

We will address this issue by setting the `padding` flag of the U-Net constructor to `True`. This will mean we can use input images of any size, and we will get output of the same size. We may lose some fidelity near the edges of the image, since the receptive field of pixels located there will include regions that have been artificially padded, but that's a compromise we decide to live with.

13.5.2 U-Net trade-offs for 3D vs. 2D data

The second issue is that our 3D data doesn't line up exactly with U-Net's 2D expected input. Simply taking our 512 × 512 × 128 image and feeding it into a converted-to-3D U-Net class won't work, because we'll exhaust our GPU memory. Each image is 2^9 by 2^9 by 2^7, with 2^2 bytes per voxel. The first layer of U-Net is 64 channels, or 2^6. That's an exponent of 9 + 9 + 7 + 2 + 6 = 33, or 8 GB *just for the first convolutional layer*. There are two convolutional layers (16 GB); and then each downsampling halves the resolution but doubles the channels, which is another 2 GB for each layer after the first downsample (remember, halving the resolution results in one-eighth the data, since we're working with 3D data). So we've hit 20 GB before we even get to the second downsample, much less anything on the upsample side of the model or anything dealing with autograd.

> **NOTE** There are a number of clever and innovative ways to get around these problems, and we in no way suggest that this is the only approach that will ever work.[6] We do feel that this approach is one of the simplest that gets the job done to the level we need for our project in this book. We'd rather keep things simple so that we can focus on the fundamental concepts; the clever stuff can come later, once you've mastered the basics.

[6] For example, Stanislav Nikolov et al., "Deep Learning to Achieve Clinically Applicable Segmentation of Head and Neck Anatomy for Radiotherapy," https://arxiv.org/pdf/1809.04430.pdf.

As anticipated, instead of trying to do things in 3D, we're going to treat each slice as a 2D segmentation problem and cheat our way around the issue of context in the third dimension by providing neighboring slices as separate channels. Instead of the traditional "red," "green," and "blue" channels that we're familiar with from photographic images, our main channels will be "two slices above," "one slice above," "the slice we're actually segmenting," "one slice below," and so on.

This approach isn't without trade-offs, however. We lose the direct spatial relationship between slices when represented as channels, as all channels will be linearly combined by the convolution kernels with no notion of them being one or two slices away, above or below. We also lose the wider receptive field in the depth dimension that would come from a true 3D segmentation with downsampling. Since CT slices are often thicker than the resolution in rows and columns, we do get a somewhat wider view than it seems at first, and this should be enough, considering that nodules typically span a limited number of slices.

Another aspect to consider, that is relevant for both the current and fully 3D approaches, is that we are now ignoring the exact slice thickness. This is something our model will eventually have to learn to be robust against, by being presented with data with different slice spacings.

In general, there isn't an easy flowchart or rule of thumb that can give canned answers to questions about which trade-offs to make, or whether a given set of compromises compromise too much. Careful experimentation is key, however, and systematically testing hypothesis after hypothesis can help narrow down which changes and approaches are working well for the problem at hand. Although it's tempting to make a flurry of changes while waiting for the last set of results to compute, *resist that impulse.*

That's important enough to repeat: *do not test multiple modifications at the same time.* There is far too high a chance that one of the changes will interact poorly with the other, and you'll be left without solid evidence that either one is worth investigating further. With that said, let's start building out our segmentation dataset.

13.5.3 Building the ground truth data

The first thing we need to address is that we have a mismatch between our human-labeled training data and the actual output we want to get from our model. We have annotated points, but we want a per-voxel mask that indicates whether any given voxel is part of a nodule. We'll have to build that mask ourselves from the data we have and then do some manual checking to make sure the routine that builds the mask is performing well.

Validating these manually constructed heuristics at scale can be difficult. We aren't going to attempt to do anything comprehensive when it comes to making sure each and every nodule is properly handled by our heuristics. If we had more resources, approaches like "collaborate with (or pay) someone to create and/or verify everything by hand" might be an option, but since this isn't a well-funded endeavor, we'll rely on checking a handful of samples and using a very simple "does the output look reasonable?" approach.

To that end, we'll design our approaches and our APIs to make it easy to investigate the intermediate steps that our algorithms are going through. While this might result in slightly clunky function calls returning huge tuples of intermediate values, being able to easily grab results and plot them in a notebook makes the clunk worth it.

BOUNDING BOXES

We are going to begin by converting the nodule locations that we have into bounding boxes that cover the entire nodule (note that we'll only do this for *actual nodules*). If we assume that the nodule locations are roughly centered in the mass, we can trace outward from that point in all three dimensions until we hit low-density voxels, indicating that we've reached normal lung tissue (which is mostly filled with air). Let's follow this algorithm in figure 13.10.

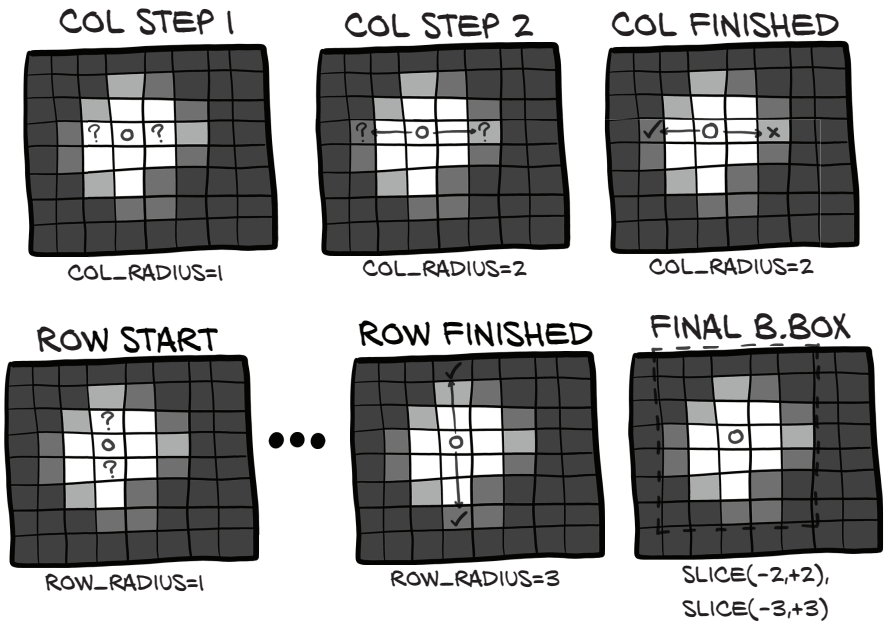


Figure 13.10 An algorithm for finding a bounding box around a lung nodule

We start the origin of our search (O in the figure) at the voxel at the annotated center of our nodule. We then examine the density of the voxels adjacent to our origin on the column axis, marked with a question mark (?). Since both of the examined voxels contain dense tissue, shown here in lighter colors, we continue our search. After incrementing our column search distance to 2, we find that the left voxel has a density below our threshold, and so we stop our search at 2.

Next, we perform the same search in the row direction. Again, we start at the origin, and this time we search up and down. After our search distance becomes 3, we encounter a low-density voxel in both the upper and lower search locations. We only need one to stop our search!

We'll skip showing the search in the third dimension. Our final bounding box is five voxels wide and seven voxels tall. Here's what that looks like in code, for the index direction.

Listing 13.3 dsets.py:131, `Ct.buildAnnotationMask`

```
center_irc = xyz2irc(
  candidateInfo_tup.center_xyz,
  self.origin_xyz,
  self.vxSize_xyz,
  self.direction_a,
)
ci = int(center_irc.index)
cr = int(center_irc.row)
cc = int(center_irc.col)

index_radius = 2
try:
  while self.hu_a[ci + index_radius, cr, cc] > threshold_hu and \
     self.hu_a[ci - index_radius, cr, cc] > threshold_hu:
     index_radius += 1
except IndexError:
  index_radius -= 1
```

candidateInfo_tup here is the same as we've seen previously: as returned by getCandidateInfoList.

Gets the center voxel indices, our starting point

The search described previously

The safety net for indexing beyond the size of the tensor

We first grab the center data and then do the search in a `while` loop. As a slight complication, our search might fall off the boundary of our tensor. We are not terribly concerned about that case and are lazy, so we just catch the index exception.[7]

Note that we stop incrementing the very approximate `radius` values *after* the density drops below threshold, so our bounding box should contain a one-voxel border of low-density tissue (at least on one side; since nodules can be adjacent to regions like the lung wall, we have to stop searching in both directions when we hit air on either side). Since we check both `center_index + index_radius` and `center_index - index_radius` against that threshold, that one-voxel boundary will only exist on the edge closest to our nodule location. This is why we need those locations to be relatively centered. Since some nodules are adjacent to the boundary between the lung and denser tissue like muscle or bone, we can't trace each direction independently, as some edges would end up incredibly far away from the actual nodule.

We then repeat the same radius-expansion process with `row_radius` and `col_radius` (this code is omitted for brevity). Once that's done, we can set a box in our bounding-box mask array to `True` (we'll see the definition of `boundingBox_ary` in just a moment; it's not surprising).

OK, let's wrap all this up in a function. We loop over all nodules. For each nodule, we perform the search shown earlier (which we elide from listing 13.4). Then, in a Boolean tensor `boundingBox_a`, we mark the bounding box we found.

[7] The bug here is that the wraparound at 0 will go undetected. It does not matter much to us. As an exercise, implement proper bounds checking.

After the loop, we do a bit of cleanup by taking the intersection between the bounding-box mask and the tissue that's denser than our threshold of –700 HU (or 0.3 g/cc). That's going to clip off the corners of our boxes (at least, the ones not embedded in the lung wall), and make it conform to the contours of the nodule a bit better.

Listing 13.4 dsets.py:127, `Ct.buildAnnotationMask`

Starts with an all-False tensor of the same size as the CT

```
    def buildAnnotationMask(self, positiveInfo_list, threshold_hu = -700):
        boundingBox_a = np.zeros_like(self.hu_a, dtype=np.bool)

        for candidateInfo_tup in positiveInfo_list:
            # ... line 169
            boundingBox_a[
                ci - index_radius: ci + index_radius + 1,
                cr - row_radius: cr + row_radius + 1,
                cc - col_radius: cc + col_radius + 1] = True

        mask_a = boundingBox_a & (self.hu_a > threshold_hu)

        return mask_a
```

Loops over the nodules. As a reminder that we are only looking at nodules, we call the variable positiveInfo_list.

Restricts the mask to voxels above our density threshold

After we get the nodule radius (the search itself is left out), we mark the bounding box.

Let's take a look at figure 13.11 to see what these masks look like in practice. Additional images in full color can be found in the p2ch13_explore_data.ipynb notebook.

The bottom-right nodule mask demonstrates a limitation of our rectangular bounding-box approach by including a portion of the lung wall. It's certainly something

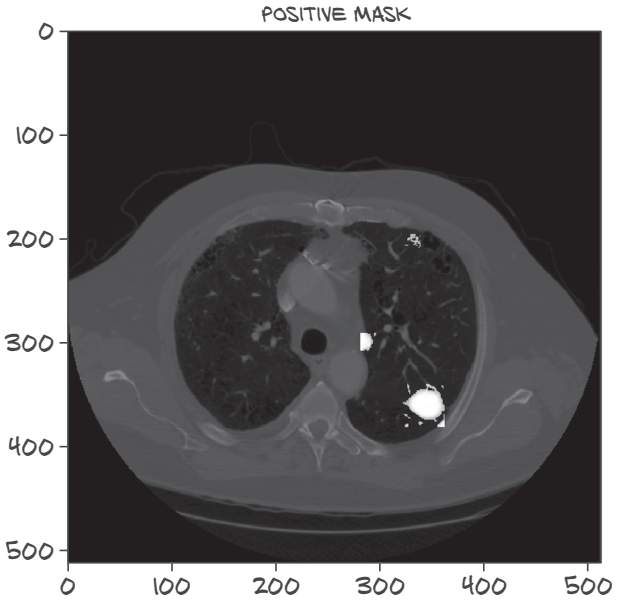


Figure 13.11 Three nodules from `ct.positive_mask`, highlighted in white

we could fix, but since we're not yet convinced that's the best use of our time and attention, we'll let it remain as is for now.[8] Next, we'll go about adding this mask to our CT class.

CALLING MASK CREATION DURING CT INITIALIZATION

Now that we can take a list of nodule information tuples and turn them into at CT-shaped binary "Is this a nodule?" mask, let's embed those masks into our CT object. First, we'll filter our candidates into a list containing only nodules, and then we'll use that list to build the annotation mask. Finally, we'll collect the set of unique array indexes that have at least one voxel of the nodule mask. We'll use this to shape the data we use for validation.

Listing 13.5 dsets.py:99, `Ct.__init__`

```
def __init__(self, series_uid):
  # ... line 116
  candidateInfo_list = getCandidateInfoDict()[self.series_uid]

  self.positiveInfo_list = [
    candidate_tup
    for candidate_tup in candidateInfo_list
  if candidate_tup.isNodule_bool
  ]
  self.positive_mask = self.buildAnnotationMask(self.positiveInfo_list)
  self.positive_indexes = (self.positive_mask.sum(axis=(1,2))
                .nonzero()[0].tolist())
```

Filters for nodules

Gives us a 1D vector (over the slices) with the number of voxels flagged in the mask in each slice

Takes indices of the mask slices that have a nonzero count, which we make into a list

Keen eyes might have noticed the `getCandidateInfoDict` function. The definition isn't surprising; it's just a reformulation of the same information as in the `getCandidateInfoList` function, but pregrouped by `series_uid`.

Listing 13.6 dsets.py:87

```
@functools.lru_cache(1)
def getCandidateInfoDict(requireOnDisk_bool=True):
  candidateInfo_list = getCandidateInfoList(requireOnDisk_bool)
  candidateInfo_dict = {}

  for candidateInfo_tup in candidateInfo_list:
    candidateInfo_dict.setdefault(candidateInfo_tup.series_uid,
                    []).append(candidateInfo_tup)

  return candidateInfo_dict
```

This can be useful to keep Ct init from being a performance bottleneck.

Takes the list of candidates for the series UID from the dict, defaulting to a fresh, empty list if we cannot find it. Then appends the present candidateInfo_tup to it.

[8] Fixing this issue would not do a great deal to teach you about PyTorch.

CACHING CHUNKS OF THE MASK IN ADDITION TO THE CT

In earlier chapters, we cached chunks of CT centered around nodule candidates, since we didn't want to have to read and parse all of a CT's data every time we wanted a small chunk of the CT. We'll want to do the same thing with our new `positive_mask`, so we need to also return it from our `Ct.getRawCandidate` function. This works out to an additional line of code and an edit to the `return` statement.

Listing 13.7 dsets.py:178, `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 = []
  # ... line 203
  ct_chunk = self.hu_a[tuple(slice_list)]
  pos_chunk = self.positive_mask[tuple(slice_list)]      <-- Newly added

  return ct_chunk, pos_chunk, center_irc      <-- New value returned here
```

This will, in turn, be cached to disk by the `getCtRawCandidate` function, which opens the CT, gets the specified raw candidate including the nodule mask, and clips the CT values before returning the CT chunk, mask, and center information.

Listing 13.8 dsets.py:212

```
@raw_cache.memoize(typed=True)
def getCtRawCandidate(series_uid, center_xyz, width_irc):
  ct = getCt(series_uid)
  ct_chunk, pos_chunk, center_irc = ct.getRawCandidate(center_xyz,
                                width_irc)
  ct_chunk.clip(-1000, 1000, ct_chunk)
  return ct_chunk, pos_chunk, center_irc
```

The `prepcache` script precomputes and saves all these values for us, helping keep training quick.

CLEANING UP OUR ANNOTATION DATA

Another thing we're going to take care of in this chapter is doing some better screening on our annotation data. It turns out that several of the candidates listed in candidates.csv are present multiple times. To make it even more interesting, those entries are not exact duplicates of one another. Instead, it seems that the original human annotations weren't sufficiently cleaned before being entered in the file. They might be annotations on the same nodule on different slices, which might even have been beneficial for our classifier.

We'll do a bit of a hand wave here and provide a cleaned up annotation.csv file. In order to fully walk through the provenance of this cleaned file, you'll need to know that the LUNA dataset is derived from another dataset called the Lung Image Database

Consortium image collection (LIDC-IDRI)[9] and includes detailed annotation information from multiple radiologists. We've already done the legwork to get the original LIDC annotations, pull out the nodules, dedupe them, and save them to the file /data/part2/luna/annotations_with_malignancy.csv.

With that file, we can update our `getCandidateInfoList` function to pull our nodules from our new annotations file. First, we loop over the new annotations for the actual nodules. Using the CSV reader,[10] we need to convert the data to the appropriate types before we stick them into our `CandidateInfoTuple` data structure.

Listing 13.9 dsets.py:43, `def getCandidateInfoList`

```
candidateInfo_list = []
with open('data/part2/luna/annotations_with_malignancy.csv', "r") as f:
  for row in list(csv.reader(f))[1:]:                          # For each line in the annotations file that represents one nodule, ...
    series_uid = row[0]
    annotationCenter_xyz = tuple([float(x) for x in row[1:4]])
    annotationDiameter_mm = float(row[4])
    isMal_bool = {'False': False, 'True': True}[row[5]]

    candidateInfo_list.append(                                 # ... we add a record to our list.
      CandidateInfoTuple(
        True,                                                  # isNodule_bool
        True,                                                  # hasAnnotation_bool
        isMal_bool,
        annotationDiameter_mm,
        series_uid,
        annotationCenter_xyz,
      )
    )
```

Similarly, we loop over candidates from candidates.csv as before, but this time we only use the non-nodules. As these are not nodules, the nodule-specific information will just be filled with `False` and `0`.

Listing 13.10 dsets.py:62, `def getCandidateInfoList`

```
with open('data/part2/luna/candidates.csv', "r") as f:         # For each line in the candidates file ...
  for row in list(csv.reader(f))[1:]:
    series_uid = row[0]
    # ... line 72
    if not isNodule_bool:                                      # ... but only the non-nodules (we have the others from earlier) ...
      candidateInfo_list.append(                               # ... we add a candidate record.
        CandidateInfoTuple(
```

[9] Samuel G. Armato 3rd et al., 2011, "The Lung Image Database Consortium (LIDC) and Image Database Resource Initiative (IDRI): A Completed Reference Database of Lung Nodules on CT Scans," *Medical Physics* 38, no. 2 (2011): 915-31, https://pubmed.ncbi.nlm.nih.gov/21452728/. See also Bruce Vendt, LIDC-IDRI, Cancer Imaging Archive, http://mng.bz/mBO4.

[10] If you do this a lot, the `pandas` library that just released 1.0 in 2020 is a great tool to make this faster. We stick with the CSV reader included in the standard Python distribution here.

```
                False,      <-- isNodule_bool
                False,    <-- hasAnnotation_bool
isMal_bool -->  False,
                0.0,
                series_uid,
                candidateCenter_xyz,
            )
        )
```

Other than the addition of the `hasAnnotation_bool` and `isMal_bool` flags (which we won't use in this chapter), the new annotations will slot in and be usable just like the old ones.

> **NOTE** You might be wondering why we haven't discussed the LIDC before now. As it turns out, the LIDC has a large amount of tooling that's already been constructed around the underlying dataset, which is specific to the LIDC. You could even get ready-made masks from PyLIDC. That tooling presents a somewhat unrealistic picture of what sort of support a given dataset might have, since the LIDC is anomalously well supported. What we've done with the LUNA data is much more typical and provides for better learning, since we're spending our time manipulating the raw data rather than learning an API that someone else cooked up.

13.5.4 Implementing Luna2dSegmentationDataset

Compared to previous chapters, we are going to take a different approach to the training and validation split in this chapter. We will have two classes: one acting as a general base class suitable for validation data, and one subclassing the base for the training set, with randomization and a cropped sample.

While this approach is somewhat more complicated in some ways (the classes aren't perfectly encapsulated, for example), it actually simplifies the logic of selecting randomized training samples and the like. It also becomes extremely clear which code paths impact both training and validation, and which are isolated to training only. Without this, we found that some of the logic can become nested or intertwined in ways that make it hard to follow. This is important because our training data will look significantly different from our validation data!

> **NOTE** Other class arrangements are also viable; we considered having two entirely separate `Dataset` subclasses, for example. Standard software engineering design principles apply, so try to keep your structure relatively simple, and try to not copy and paste code, but don't invent complicated frameworks to prevent having to duplicate three lines of code.

The data that we produce will be two-dimensional CT slices with multiple channels. The extra channels will hold adjacent slices of CT. Recall figure 4.2, shown here as figure 13.12; we can see that each slice of CT scan can be thought of as a 2D grayscale image.

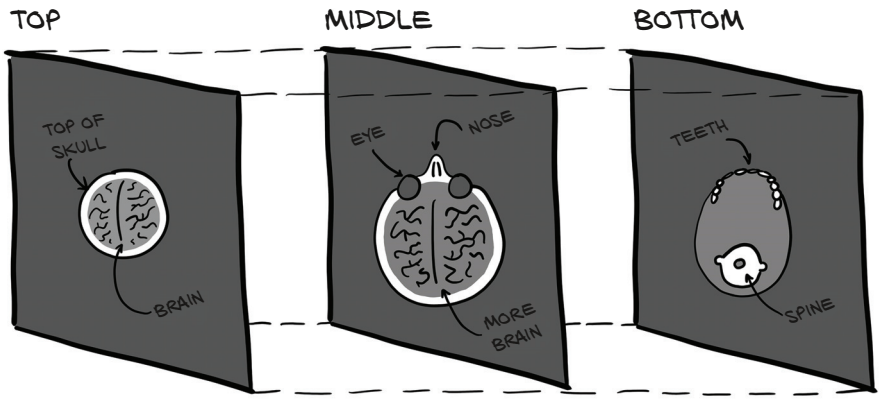


Figure 13.12 Each slice of a CT scan represents a different position in space.

How we combine those slices is up to us. For the input to our classification model, we treated those slices as a 3D array of data and used 3D convolutions to process each sample. For our segmentation model, we are going to instead treat each slice as a single channel, and produce a multichannel 2D image. Doing so will mean that we are treating each slice of CT scan as if it was a color channel of an RGB image, like we saw in figure 4.1, repeated here as figure 13.13. Each input slice of the CT will get stacked together and consumed just like any other 2D image. The channels of our stacked CT image won't correspond to colors, but nothing about 2D convolutions requires the input channels to be colors, so it works out fine.

For validation, we'll need to produce one sample per slice of CT that has an entry in the positive mask, for each validation CT we have. Since different CT scans can have different slice counts,[11] we're going to introduce a new function that caches the

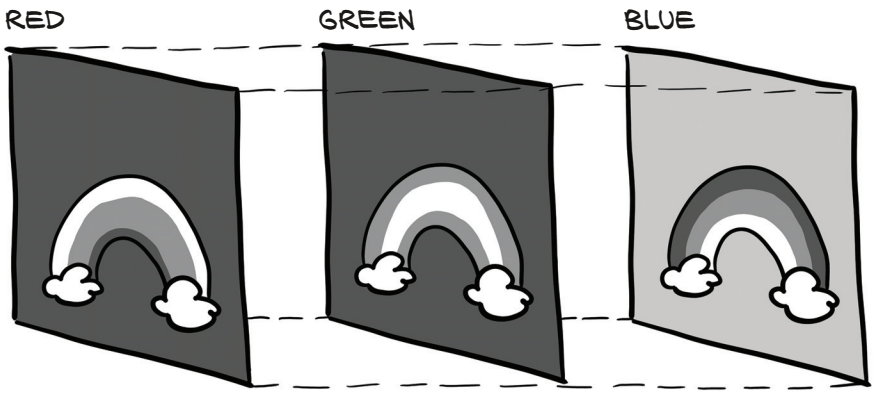


Figure 13.13 Each channel of a photographic image represents a different color.

[11] Most CT scanners produce 512 × 512 slices, and we're not going to worry about the ones that do something different.

size of each CT scan and its positive mask to disk. We need this to be able to quickly construct the full size of a validation set without having to load each CT at `Dataset` initialization. We'll continue to use the same caching decorator as before. Populating this data will also take place during the prepcache.py script, which we must run once before we start any model training.

Listing 13.11 dsets.py:220

```
@raw_cache.memoize(typed=True)
def getCtSampleSize(series_uid):
  ct = Ct(series_uid)
  return int(ct.hu_a.shape[0]), ct.positive_indexes
```

The majority of the `Luna2dSegmentationDataset.__init__` method is similar to what we've seen before. We have a new `contextSlices_count` parameter, as well as an `augmentation_dict` similar to what we introduced in chapter 12.

The handling for the flag indicating whether this is meant to be a training or validation set needs to change somewhat. Since we're no longer training on individual nodules, we will have to partition the list of series, taken as a whole, into training and validation sets. This means an entire CT scan, along with all nodule candidates it contains, will be in either the training set or the validation set.

Listing 13.12 dsets.py:242, .__init__

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

Starting with a series list containing all our series, we keep only every val_stride-th element, starting with 0.

If we are training, we delete every val_stride-th element instead.

Speaking of validation, we're going to have two different modes we can validate our training with. First, when `fullCt_bool` is `True`, we will use every slice in the CT for our dataset. This will be useful when we're evaluating end-to-end performance, since we need to pretend that we're starting off with no prior information about the CT. We'll use the second mode for validation during training, which is when we're limiting ourselves to only the CT slices that have a positive mask present.

As we now only want certain CT series to be considered, we loop over the series UIDs we want and get the total number of slices and the list of interesting ones.

Listing 13.13 dsets.py:250, .__init__

```
self.sample_list = []
for series_uid in self.series_list:
```

```
index_count, positive_indexes = getCtSampleSize(series_uid)

if self.fullCt_bool:
  self.sample_list += [(series_uid, slice_ndx)
              for slice_ndx in range(index_count)]
else:
  self.sample_list += [(series_uid, slice_ndx)
              for slice_ndx in positive_indexes]
```

Here we extend sample_list with every slice of the CT by using range …

… while here we take only the interesting slices.

Doing it this way will keep our validation relatively quick and ensure that we're getting complete stats for true positives and false negatives, but we're making the assumption that other slices will have false positive and true negative stats relatively similar to the ones we evaluate during validation.

Once we have the set of `series_uid` values we'll be using, we can filter our `candidateInfo_list` to contain only nodule candidates with a `series_uid` that is included in that set of series. Additionally, we'll create another list that has only the positive candidates so that during training, we can use those as our training samples.

Listing 13.14 dsets.py:261, .__init__

```
self.candidateInfo_list = getCandidateInfoList()

series_set = set(self.series_list)
self.candidateInfo_list = [cit for cit in self.candidateInfo_list
              if cit.series_uid in series_set]

self.pos_list = [nt for nt in self.candidateInfo_list
          if nt.isNodule_bool]
```

This is cached.

Makes a set for faster lookup

Filters out the candidates from series not in our set

For the data balancing yet to come, we want a list of actual nodules.

Our `__getitem__` implementation will also be a bit fancier by delegating a lot of the logic to a function that makes it easier to retrieve a specific sample. At the core of it, we'd like to retrieve our data in three different forms. First, we have the full slice of the CT, as specified by a `series_uid` and `ct_ndx`. Second, we have a cropped area around a nodule, which we'll use for training data (we'll explain in a bit why we're not using full slices). Finally, the `DataLoader` is going to ask for samples via an integer `ndx`, and the dataset will need to return the appropriate type based on whether it's training or validation.

The base class or subclass `__getitem__` functions will convert from the integer `ndx` to either the full slice or training crop, as appropriate. As mentioned, our validation set's `__getitem__` just calls another function to do the real work. Before that, it wraps the index around into the sample list in order to decouple the epoch size (given by the length of the dataset) from the actual number of samples.

Listing 13.15 dsets.py:281, .__getitem__

The modulo operation does the wrapping.

```
def __getitem__(self, ndx):
  series_uid, slice_ndx = self.sample_list[ndx % len(self.sample_list)]
  return self.getitem_fullSlice(series_uid, slice_ndx)
```

That was easy, but we still need to implement the interesting functionality from the `getItem_fullSlice` method.

Listing 13.16 dsets.py:285, .getitem_fullSlice

```
def getitem_fullSlice(self, series_uid, slice_ndx):
  ct = getCt(series_uid)
  ct_t = torch.zeros((self.contextSlices_count * 2 + 1, 512, 512))

  start_ndx = slice_ndx - self.contextSlices_count
  end_ndx = slice_ndx + self.contextSlices_count + 1
  for i, context_ndx in enumerate(range(start_ndx, end_ndx)):
    context_ndx = max(context_ndx, 0)
    context_ndx = min(context_ndx, ct.hu_a.shape[0] - 1)
    ct_t[i] = torch.from_numpy(ct.hu_a[context_ndx].astype(np.float32))
  ct_t.clamp_(-1000, 1000)

  pos_t = torch.from_numpy(ct.positive_mask[slice_ndx]).unsqueeze(0)

  return ct_t, pos_t, ct.series_uid, slice_ndx
```

Preallocates the output

When we reach beyond the bounds of the ct_a, we duplicate the first or last slice.

Splitting the functions like this means we can always ask a dataset for a specific slice (or cropped training chunk, which we'll see in the next section) indexed by series UID and position. Only for the integer indexing do we go through `__getitem__`, which then gets a sample from the (shuffled) list.

Aside from `ct_t` and `pos_t`, the rest of the tuple we return is all information that we include for debugging and display. We don't need any of it for training.

13.5.5 Designing our training and validation data

Before we get into the implementation for our training dataset, we need to explain why our training data will look different from our validation data. Instead of the full CT slices, we're going to train on 64 × 64 crops around our positive candidates (the actually-a-nodule candidates). These 64 × 64 patches will be taken randomly from a 96 × 96 crop centered on the nodule. We will also include three slices of context in both directions as additional "channels" to our 2D segmentation.

We're doing this to make training more stable, and to converge more quickly. The only reason we know to do this is because we tried to train on whole CT slices, but we found the results unsatisfactory. After some experimentation, we found that the 64 × 64 semirandom crop approach worked well, so we decided to use that for the book.

When you work on your own projects, you'll need to do that kind of experimentation for yourself!

We believe the whole-slice training was unstable essentially due to a class-balancing issue. Since each nodule is so small compared to the whole CT slice, we were right back in a needle-in-a-haystack situation similar to the one we got out of in the last chapter, where our positive samples were swamped by the negatives. In this case, we're talking about pixels rather than nodules, but the concept is the same. By training on crops, we're keeping the number of positive pixels the same and reducing the negative pixel count by several orders of magnitude.

Because our segmentation model is pixel-to-pixel and takes images of arbitrary size, we can get away with training and validating on samples with different dimensions. Validation uses the same convolutions with the same weights, just applied to a larger set of pixels (and so with fewer border pixels to fill in with edge data).

One caveat to this approach is that since our validation set contains orders of magnitude more negative pixels, our model will have a huge false positive rate during validation. There are many more opportunities for our segmentation model to get tricked! It doesn't help that we're going to be pushing for high recall as well. We'll discuss that more in section 13.6.3.

13.5.6 Implementing TrainingLuna2dSegmentationDataset

With that out of the way, let's get back to the code. Here's the training set's `__getitem__`. It looks just like the one for the validation set, except that we now sample from `pos_list` and call `getItem_trainingCrop` with the candidate info tuple, since we need the series and the exact center location, not just the slice.

Listing 13.17 dsets.py:320, `.__getitem__`

```
def __getitem__(self, ndx):
  candidateInfo_tup = self.pos_list[ndx % len(self.pos_list)]
  return self.getitem_trainingCrop(candidateInfo_tup)
```

To implement `getItem_trainingCrop`, we will use a `getCtRawCandidate` function similar to the one we used during classification training. Here, we're passing in a different size crop, but the function is unchanged except for now returning an additional array with a crop of the `ct.positive_mask` as well.

We limit our `pos_a` to the center slice that we're actually segmenting, and then construct our 64 × 64 random crops of the 96 × 96 we were given by `getCtRawCandidate`. Once we have those, we return a tuple with the same items as our validation dataset.

Listing 13.18 dsets.py:324, `.getitem_trainingCrop`

```
def getitem_trainingCrop(self, candidateInfo_tup):
  ct_a, pos_a, center_irc = getCtRawCandidate(
    candidateInfo_tup.series_uid,
    candidateInfo_tup.center_xyz,
```

Gets the candidate with a bit of extra surrounding

```
  (7, 96, 96),
)
pos_a = pos_a[3:4]

row_offset = random.randrange(0,32)
col_offset = random.randrange(0,32)
ct_t = torch.from_numpy(ct_a[:, row_offset:row_offset+64,
                col_offset:col_offset+64]).to(torch.float32)
pos_t = torch.from_numpy(pos_a[:, row_offset:row_offset+64,
                  col_offset:col_offset+64]).to(torch.long)

slice_ndx = center_irc.index

return ct_t, pos_t, candidateInfo_tup.series_uid, slice_ndx
```

Taking a one-element slice keeps the third dimension, which will be the (single) output channel.

With two random numbers between 0 and 31, we crop both CT and mask.

You might have noticed that data augmentation is missing from our dataset implementation. We're going to handle that a little differently this time around: we'll augment our data on the GPU.

13.5.7 Augmenting on the GPU

One of the key concerns when it comes to training a deep learning model is avoiding bottlenecks in your training pipeline. Well, that's not quite true—there will *always* be a bottleneck.[12] The trick is to make sure the bottleneck is at the resource that's the most expensive or difficult to upgrade, and that your usage of that resource isn't wasteful.

Some common places to see bottlenecks are as follows:

- In the data-loading pipeline, either in raw I/O or in decompressing data once it's in RAM. We addressed this with our `diskcache` library usage.
- In CPU preprocessing of the loaded data. This is often data normalization or augmentation.
- In the training loop on the GPU. This is typically where we want our bottleneck to be, since total deep learning system costs for GPUs are usually higher than for storage or CPU.
- Less commonly, the bottleneck can sometimes be the *memory bandwidth* between CPU and GPU. This implies that the GPU isn't doing much work compared to the data size that's being sent in.

Since GPUs can be 50 times faster than CPUs when working on tasks that fit GPUs well, it often makes sense to move those tasks to the GPU from the CPU in cases where CPU usage is becoming high. This is especially true if the data gets expanded during this processing; by moving the smaller input to the GPU first, the expanded data is kept local to the GPU, and less memory bandwidth is used.

In our case, we're going to move data augmentation to the GPU. This will keep our CPU usage light, and the GPU will easily be able to accommodate the additional workload. Far better to have the GPU busy with a small bit of extra work than idle waiting for the CPU to struggle through the augmentation process.

[12] Otherwise, your model would train instantly!

We'll accomplish this by using a second model, similar to all the other subclasses of `nn.Module` we've seen so far in this book. The main difference is that we're not interested in backpropagating gradients through the model, and the `forward` method will be doing decidedly different things. There will be some slight modifications to the actual augmentation routines since we're working with 2D data for this chapter, but otherwise, the augmentation will be very similar to what we saw in chapter 12. The model will consume tensors and produce different tensors, just like the other models we've implemented.

Our model's `__init__` takes the same data augmentation arguments—`flip`, `offset`, and so on—that we used in the last chapter, and assigns them to `self`.

Listing 13.19 model.py:56, `class SegmentationAugmentation`

```
class SegmentationAugmentation(nn.Module):
  def __init__(
      self, flip=None, offset=None, scale=None, rotate=None, noise=None
  ):
    super().__init__()

    self.flip = flip
    self.offset = offset
    # ... line 64
```

Our augmentation `forward` method takes the input and the label, and calls out to build the `transform_t` tensor that will then drive our `affine_grid` and `grid_sample` calls. Those calls should feel very familiar from chapter 12.

Listing 13.20 model.py:68, `SegmentationAugmentation.forward`

```
def forward(self, input_g, label_g):
  transform_t = self._build2dTransformMatrix()
  transform_t = transform_t.expand(input_g.shape[0], -1, -1)
  transform_t = transform_t.to(input_g.device, torch.float32)
  affine_t = F.affine_grid(transform_t[:,:2],
      input_g.size(), align_corners=False)

  augmented_input_g = F.grid_sample(input_g,
      affine_t, padding_mode='border',
      align_corners=False)
  augmented_label_g = F.grid_sample(label_g.to(torch.float32),
      affine_t, padding_mode='border',
      align_corners=False)

  if self.noise:
    noise_t = torch.randn_like(augmented_input_g)
    noise_t *= self.noise

    augmented_input_g += noise_t

  return augmented_input_g, augmented_label_g > 0.5
```

Note that we're augmenting 2D data.

The first dimension of the transformation is the batch, but we only want the first two rows of the 3 × 3 matrices per batch item.

We need the same transformation applied to CT and mask, so we use the same grid. Because grid_sample only works with floats, we convert here.

Just before returning, we convert the mask back to Booleans by comparing to 0.5. The interpolation that grid_sample results in fractional values.

Now that we know what we need to do with `transform_t` to get our data out, let's take a look at the `_build2dTransformMatrix` function that actually creates the transformation matrix we use.

Listing 13.21 model.py:90, ._build2dTransformMatrix

```
def _build2dTransformMatrix(self):
  transform_t = torch.eye(3)

  for i in range(2):
    if self.flip:
      if random.random() > 0.5:
        transform_t[i,i] *= -1
  # ... line 108
  if self.rotate:
    angle_rad = random.random() * math.pi * 2
    s = math.sin(angle_rad)
    c = math.cos(angle_rad)

    rotation_t = torch.tensor([
      [c, -s, 0],
      [s, c, 0],
      [0, 0, 1]])

    transform_t @= rotation_t

  return transform_t
```

Creates a 3 × 3 matrix, but we will drop the last row later.

Again, we're augmenting 2D data here.

Takes a random angle in radians, so in the range 0 .. 2{pi}

Rotation matrix for the 2D rotation by the random angle in the first two dimensions

Applies the rotation to the transformation matrix using the Python matrix multiplication operator

Other than the slight differences to deal with 2D data, our GPU augmentation code looks very similar to our CPU augmentation code. That's great, because it means we're able to write code that doesn't have to care very much about where it runs. The primary difference isn't in the core implementation: it's how we wrapped that implementation into a `nn.Module` subclass. While we've been thinking about models as exclusively a deep learning tool, this shows us that with PyTorch, tensors can be used quite a bit more generally. Keep this in mind when you start your next project—the range of things you can accomplish with a GPU-accelerated tensor is pretty large!

13.6 Updating the training script for segmentation

We have a model. We have data. We need to use them, and you won't be surprised when step 2C of figure 13.14 suggests we should train our new model with the new data.

To be more precise about the process of training our model, we will update three things affecting the outcome from the training code we got in chapter 12:

- We need to instantiate the new model (unsurprisingly).
- We will introduce a new loss: the Dice loss.
- We will also look at an optimizer other than the venerable SGD we've used so far. We'll stick with a popular one and use Adam.

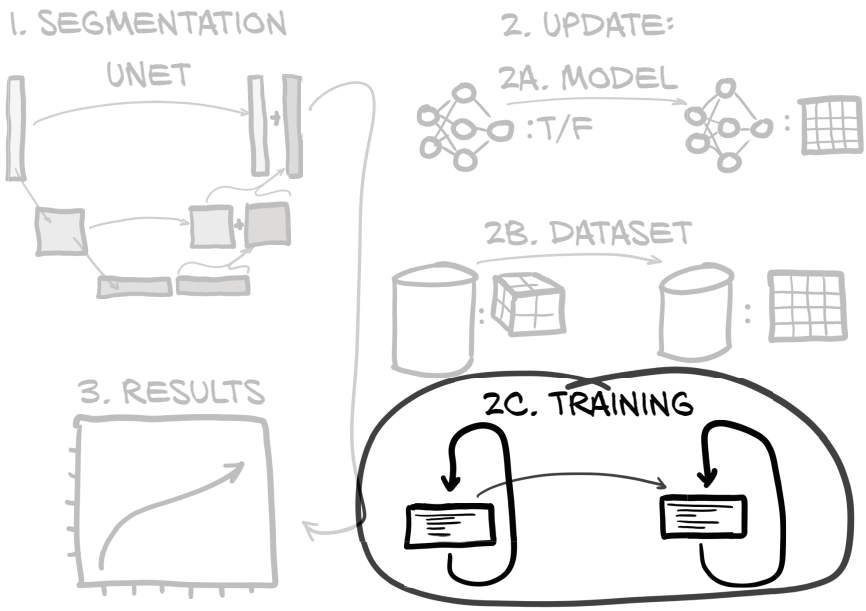


Figure 13.14 The outline of this chapter, with a focus on the changes needed for our training loop

But we will also step up our bookkeeping, by

- Logging images for visual inspection of the segmentation to TensorBoard
- Performing more metrics logging in TensorBoard
- Saving our best model based on the validation

Overall, the training script p2ch13/training.py is even more similar to what we used for classification training in chapter 12 than the adapted code we've seen so far. Any significant changes will be covered here in the text, but be aware that some of the minor tweaks are skipped. For the full story, check the source.

13.6.1 Initializing our segmentation and augmentation models

Our `initModel` method is very unsurprising. We are using the `UNetWrapper` class and giving it our configuration parameters—which we will look at in detail shortly. Also, we now have a second model for augmentation. Just like before, we can move the model to the GPU if desired and possibly set up multi-GPU training using `DataParallel`. We skip these administrative tasks here.

Listing 13.22 training.py:133, `.initModel`

```
def initModel(self):
  segmentation_model = UNetWrapper(
    in_channels=7,
```

```
    n_classes=1,
    depth=3,
    wf=4,
    padding=True,
    batch_norm=True,
    up_mode='upconv',
  )

  augmentation_model = SegmentationAugmentation(**self.augmentation_dict)

  # ... line 154
  return segmentation_model, augmentation_model
```

For input into `UNet`, we've got seven input channels: 3 + 3 context slices, and 1 slice that is the focus for what we're actually segmenting. We have one output class indicating whether this voxel is part of a nodule. The `depth` parameter controls how deep the U goes; each downsampling operation adds 1 to the depth. Using `wf=5` means the first layer will have `2**wf == 32` filters, which doubles with each downsampling. We want the convolutions to be padded so that we get an output image the same size as our input. We also want batch normalization inside the network after each activation function, and our upsampling function should be an upconvolution layer, as implemented by `nn.ConvTranspose2d` (see util/unet.py, line 123).

13.6.2 Using the Adam optimizer

The Adam optimizer (https://arxiv.org/abs/1412.6980) is an alternative to using SGD when training our models. Adam maintains a separate learning rate for each parameter and automatically updates that learning rate as training progresses. Due to these automatic updates, we typically won't need to specify a non-default learning rate when using Adam, since it will quickly determine a reasonable learning rate by itself.

Here's how we instantiate `Adam` in code.

Listing 13.23 training.py:156, `.initOptimizer`

```
def initOptimizer(self):
  return Adam(self.segmentation_model.parameters())
```

It's generally accepted that Adam is a reasonable optimizer to start most projects with.[13] There is often a configuration of stochastic gradient descent with Nesterov momentum that will outperform Adam, but finding the correct hyperparameters to use when initializing SGD for a given project can be difficult and time consuming.

There have been a large number of variations on Adam—AdaMax, RAdam, Ranger, and so on—that each have strengths and weaknesses. Delving into the details of those is outside the scope of this book, but we think that it's important to know that those alternatives exist. We'll use Adam in chapter 13.

[13] See http://cs231n.github.io/neural-networks-3.

13.6.3 Dice loss

The Sørensen-Dice coefficient (https://en.wikipedia.org/wiki/S%C3%B8rensen%E2%80%93Dice_coefficient), also known as the *Dice loss*, is a common loss metric for segmentation tasks. One advantage of using Dice loss over a per-pixel cross-entropy loss is that Dice handles the case where only a small portion of the overall image is flagged as positive. As we recall from chapter 11 in section 11.10, unbalanced training data can be problematic when using cross-entropy loss. That's exactly the situation we have here—most of a CT scan isn't a nodule. Luckily, with Dice, that won't pose as much of a problem.

The Sørensen-Dice coefficient is based on the ratio of correctly segmented pixels to the sum of the predicted and actual pixels. Those ratios are laid out in figure 13.15. On the left, we see an illustration of the Dice score. It is twice the joint area (*true positives*, striped) divided by the sum of the entire predicted area and the entire ground-truth marked area (the overlap being counted twice). On the right are two prototypical examples of high agreement/high Dice score and low agreement/low Dice score.

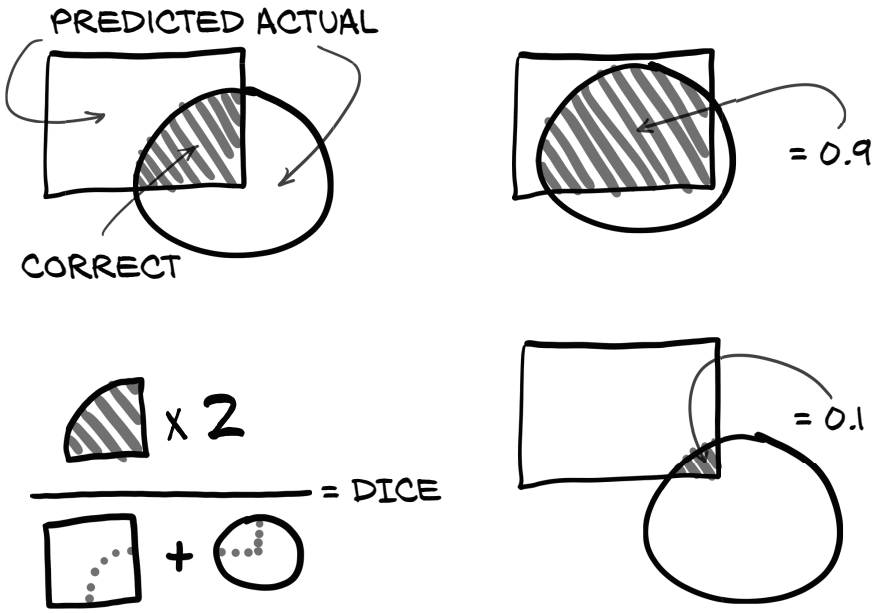


Figure 13.15 The ratios that make up the Dice score

That might sound familiar; it's the same ratio that we saw in chapter 12. We're basically going to be using a per-pixel F1 score!

> **NOTE** This is a per-pixel F1 score *where the "population" is one image's pixels*. Since the population is entirely contained within one training sample, we can use it for training directly. In the classification case, the F1 score is not calculable over a single minibatch, and, hence, we cannot use it for training directly.

Since our `label_g` is effectively a Boolean mask, we can multiply it with our predictions to get our true positives. Note that we aren't treating `prediction_devtensor` as a Boolean here. A loss defined with it wouldn't be differentiable. Instead, we're replacing the number of true positives with the sum of the predicted values for the pixels where the ground truth is 1. This converges to the same thing as the predicted values approach 1, but sometimes the predicted values will be uncertain predictions in the 0.4 to 0.6 range. Those undecided values will contribute roughly the same amount to our gradient updates, no matter which side of 0.5 they happen to fall on. A Dice coefficient utilizing continuous predictions is sometimes referred to as *soft Dice*.

There's one tiny complication. Since we're wanting a loss to minimize, we're going to take our ratio and subtract it from 1. Doing so will invert the slope of our loss function so that in the high-overlap case, our loss is low; and in the low-overlap case, it's high. Here's what that looks like in code.

Listing 13.24 training.py:315, `.diceLoss`

Sums over everything except the batch dimension to get the positively labeled, (softly) positively detected, and (softly) correct positives per batch item

The Dice ratio. To avoid problems when we accidentally have neither predictions nor labels, we add 1 to both numerator and denominator.

```
def diceLoss(self, prediction_g, label_g, epsilon=1):
  diceLabel_g = label_g.sum(dim=[1,2,3])
  dicePrediction_g = prediction_g.sum(dim=[1,2,3])
  diceCorrect_g = (prediction_g * label_g).sum(dim=[1,2,3])

  diceRatio_g = (2 * diceCorrect_g + epsilon) \
    / (dicePrediction_g + diceLabel_g + epsilon)

  return 1 - diceRatio_g
```

To make it a loss, we take 1 – Dice ratio, so lower loss is better.

We're going to update our `computeBatchLoss` function to call `self.diceLoss`. Twice. We'll compute the normal Dice loss for the training sample, as well as for only the pixels included in `label_g`. By multiplying our predictions (which, remember, are floating-point values) times the label (which are effectively Booleans), we'll get pseudo-predictions that got every negative pixel "exactly right" (since all the values for those pixels are multiplied by the false-is-zero values from `label_g`). The only pixels that will generate loss are the false negative pixels (everything that should have been predicted true, but wasn't). This will be helpful, since recall is incredibly important for our overall project; after all, we can't classify tumors properly if we don't detect them in the first place!

Listing 13.25 training.py:282, `.computeBatchLoss`

```
def computeBatchLoss(self, batch_ndx, batch_tup, batch_size, metrics_g,
                     classificationThreshold=0.5):
  input_t, label_t, series_list, _slice_ndx_list = batch_tup

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

Transfers to GPU

```
if self.segmentation_model.training and self.augmentation_dict:
  input_g, label_g = self.augmentation_model(input_g, label_g)

prediction_g = self.segmentation_model(input_g)

diceLoss_g = self.diceLoss(prediction_g, label_g)
fnLoss_g = self.diceLoss(prediction_g * label_g, label_g)
# ... line 313
return diceLoss_g.mean() + fnLoss_g.mean() * 8
```

Augments as needed if we are training. In validation, we would skip this.

Runs the segmentation model …

… and applies our fine Dice loss

Oops. What is this?

Let's talk a bit about what we're doing with our return statement of `diceLoss_g.mean() + fnLoss_g.mean() * 8`.

Loss weighting

In chapter 12, we discussed shaping our dataset so that our classes were not wildly imbalanced. That helped training converge, since the positive and negative samples present in each batch were able to counteract the general pull of the other, and the model had to learn to discriminate between them to improve. We're approximating that same balance here by cropping down our training samples to include fewer non-positive pixels; but it's incredibly important to have high recall, and we need to make sure that as we train, we're providing a loss that reflects that fact.

We are going to have a *weighted loss* that favors one class over the other. What we're saying by multiplying `fnLoss_g` by 8 is that getting the entire population of our positive pixels right is eight times more important than getting the entire population of negative pixels right (nine, if you count the one in `diceLoss_g`). Since the area covered by the positive mask is much, much smaller than the whole 64 × 64 crop, that also means each individual positive pixel wields that much more influence when it comes to backpropagation.

We're willing to trade away many correctly predicted negative pixels in the general Dice loss to gain one correct pixel in the false negative loss. Since the general Dice loss is a strict superset of the false negative loss, the only correct pixels available to make that trade are ones that start as true negatives (all of the true positive pixels are already included in the false negative loss, so there's no trade to be made).

Since we're willing to sacrifice huge swaths of true negative pixels in the pursuit of having better recall, we should expect a large number of false positives in general.[14] We're doing this because recall is very, very important to our use case, and we'd much rather have some false positives than even a single false negative.

We should note that this approach only works when using the Adam optimizer. When using SGD, the push to overpredict would lead to every pixel coming back as positive. Adam's ability to fine-tune the learning rate means stressing the false negative loss doesn't become overpowering.

[14] Roxie would be proud!

COLLECTING METRICS

Since we're going to purposefully skew our numbers for better recall, let's see just how tilted things will be. In our classification `computeBatchLoss`, we compute various per-sample values that we used for metrics and the like. We also compute similar values for the overall segmentation results. These true positive and other metrics were previously computed in `logMetrics`, but due to the size of the result data (recall that each single CT slice from the validation set is a quarter-million pixels!), we need to compute these summary stats live in the `computeBatchLoss` function.

Listing 13.26 training.py:297, `.computeBatchLoss`

```
start_ndx = batch_ndx * batch_size
end_ndx = start_ndx + input_t.size(0)

with torch.no_grad():
  predictionBool_g = (prediction_g[:, 0:1]
            > classificationThreshold).to(torch.float32)

  tp = (   predictionBool_g *  label_g).sum(dim=[1,2,3])
  fn = ((1 - predictionBool_g) *  label_g).sum(dim=[1,2,3])
  fp = (   predictionBool_g * (~label_g)).sum(dim=[1,2,3])

  metrics_g[METRICS_LOSS_NDX, start_ndx:end_ndx] = diceLoss_g
  metrics_g[METRICS_TP_NDX, start_ndx:end_ndx] = tp
  metrics_g[METRICS_FN_NDX, start_ndx:end_ndx] = fn
  metrics_g[METRICS_FP_NDX, start_ndx:end_ndx] = fp
```

We threshold the prediction to get "hard" Dice but convert to float for the later multiplication.

Computing true positives, false positives, and false negatives is similar to what we did when computing the Dice loss.

We store our metrics to a large tensor for future reference. This is per batch item rather than averaged over the batch.

As we discussed at the beginning of this section, we can compute our true positives and so on by multiplying our prediction (or its negation) and our label (or its negation) together. Since we're not as worried about the exact values of our predictions here (it doesn't really matter if we flag a pixel as 0.6 or 0.9—as long as it's over the threshold, we'll call it part of a nodule candidate), we are going to create `predictionBool_g` by comparing it to our threshold of 0.5.

13.6.4 Getting images into TensorBoard

One of the nice things about working on segmentation tasks is that the output is easily represented visually. Being able to eyeball our results can be a huge help for determining whether a model is progressing well (but perhaps needs more training), or if it has gone off the rails (so we need to stop wasting our time with further training). There are many ways we could package up our results as images, and many ways we could display them. TensorBoard has great support for this kind of data, and we already have TensorBoard `SummaryWriter` instances integrated with our training runs, so we're going to use TensorBoard. Let's see what it takes to get everything hooked up.

We'll add a `logImages` function to our main application class and call it with both our training and validation data loaders. While we are at it, we will make another

change to our training loop: we're only going to perform validation and image logging on the first and then every fifth epoch. We do this by checking the epoch number against a new constant, `validation_cadence`.

When training, we're trying to balance a few things:

- Getting a rough idea of how our model is training without having to wait very long
- Spending the bulk of our GPU cycles training, rather than validating
- Making sure we are still performing well on the validation set

The first point means we need to have relatively short epochs so that we get to call `logMetrics` more often. The second, however, means we want to train for a relatively long time before calling `doValidation`. The third means we need to call `doValidation` regularly, rather than once at the end of training or something unworkable like that. By only doing validation on the first and then every fifth epoch, we can meet all of those goals. We get an early signal of training progress, spend the bulk of our time training, and have periodic check-ins with the validation set as we go along.

Listing 13.27 training.py:210, `SegmentationTrainingApp.main`

```
def main(self):
  # ... line 217
  self.validation_cadence = 5
  for epoch_ndx in range(1, self.cli_args.epochs + 1):
    # ... line 228
    trnMetrics_t = self.doTraining(epoch_ndx, train_dl)
    self.logMetrics(epoch_ndx, 'trn', trnMetrics_t)

    if epoch_ndx == 1 or epoch_ndx % self.validation_cadence == 0:
      # ... line 239
      self.logImages(epoch_ndx, 'trn', train_dl)
      self.logImages(epoch_ndx, 'val', val_dl)
```

Our outermost loop, over the epochs

Logs the (scalar) metrics from training after each epoch

Trains for one epoch

Only every validation cadence-th interval …

… we validate the model and log images.

There isn't a single right way to structure our image logging. We are going to grab a handful of CTs from both the training and validation sets. For each CT, we will select 6 evenly spaced slices, end to end, and show both the ground truth and our model's output. We chose 6 slices only because TensorBoard will show 12 images at a time, and we can arrange the browser window to have a row of label images over the model output. Arranging things this way makes it easy to visually compare the two, as we can see in figure 13.16.

Also note the small slider-dot on the `prediction` images. That slider will allow us to view previous versions of the images with the same label (such as val/0_prediction_3, but at an earlier epoch). Being able to see how our segmentation output changes over time can be useful when we're trying to debug something or make tweaks to achieve a specific result. As training progresses, TensorBoard will limit the number of images

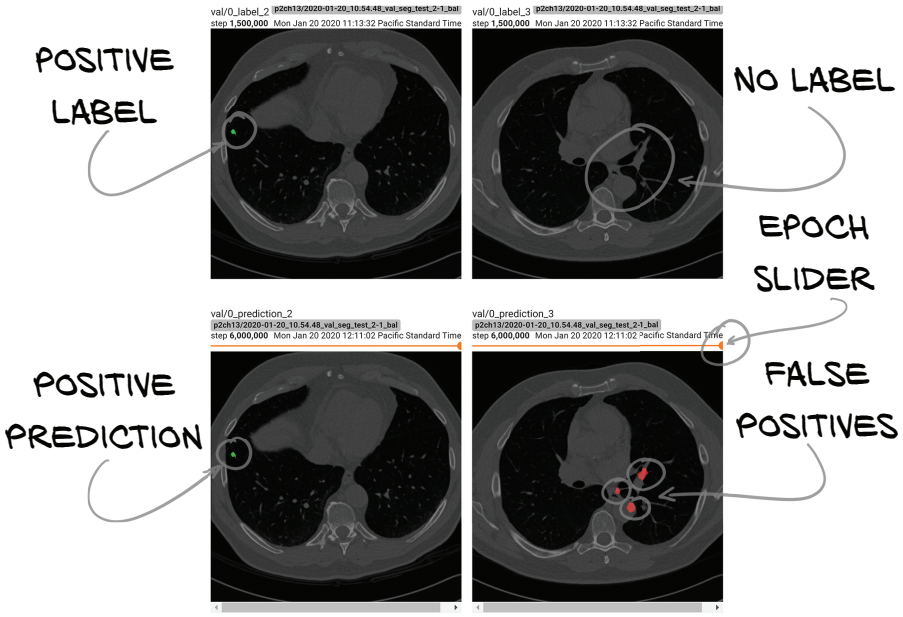


Figure 13.16 Top row: label data for training. Bottom row: output from the segmentation

viewable from the slider to 10, probably to avoid overwhelming the browser with a huge number of images.

The code that produces this output starts by getting 12 series from the pertinent data loader and 6 images from each series.

Listing 13.28 training.py:326, `.logImages`

```
def logImages(self, epoch_ndx, mode_str, dl):
  self.segmentation_model.eval()                    <-- Sets the model to eval

  images = sorted(dl.dataset.series_list)[:12]      <-- Takes (the same) 12 CTs by bypassing the data loader and using the dataset directly. The series list might be shuffled, so we sort.
  for series_ndx, series_uid in enumerate(images):
    ct = getCt(series_uid)

    for slice_ndx in range(6):                      <-- Selects six equidistant slices throughout the CT
      ct_ndx = slice_ndx * (ct.hu_a.shape[0] - 1) // 5
      sample_tup = dl.dataset.getitem_fullSlice(series_uid, ct_ndx)

      ct_t, label_t, series_uid, ct_ndx = sample_tup
```

After that, we feed `ct_t` it into the model. This looks very much like what we see in `computeBatchLoss`; see p2ch13/training.py for details if desired.

Once we have `prediction_a`, we need to build an `image_a` that will hold RGB values to display. We're using `np.float32` values, which need to be in a range from 0 to 1.

Our approach will cheat a little by adding together various images and masks to get data in the range 0 to 2, and then multiplying the entire array by 0.5 to get it back into the right range.

Listing 13.29 training.py:346, `.logImages`

CT intensity is assigned to all RGB channels to provide a grayscale base image.

```
ct_t[:-1,:,:] /= 2000
ct_t[:-1,:,:] += 0.5

ctSlice_a = ct_t[dl.dataset.contextSlices_count].numpy()

image_a = np.zeros((512, 512, 3), dtype=np.float32)
image_a[:,:,:] = ctSlice_a.reshape((512,512,1))
image_a[:,:,0] += prediction_a & (1 - label_a)
image_a[:,:,0] += (1 - prediction_a) & label_a
image_a[:,:,1] += ((1 - prediction_a) & label_a) * 0.5

image_a[:,:,1] += prediction_a & label_a
image_a *= 0.5
image_a.clip(0, 1, image_a)
```

False positives are flagged as red and overlaid on the image.

False negatives are orange.

True positives are green.

Our goal is to have a grayscale CT at half intensity, overlaid with predicted-nodule (or, more correctly, nodule-candidate) pixels in various colors. We're going to use red for all pixels that are incorrect (false positives *and* false negatives). This will mostly be false positives, which we don't care about too much (since we're focused on recall). `1 - label_a` inverts the label, and that multiplied by the `prediction_a` gives us only the predicted pixels that aren't in a candidate nodule. False negatives get a half-strength mask added to green, which means they will show up as orange (1.0 red and 0.5 green renders as orange in RGB). Every correctly predicted pixel inside a nodule is set to green; since we got those pixels right, no red will be added, and so they will render as pure green.

After that, we renormalize our data to the `0…1` range and clamp it (in case we start displaying augmented data here, which would cause speckles when the noise was outside our expected CT range). All that remains is to save the data to TensorBoard.

Listing 13.30 training.py:361, `.logImages`

```
writer = getattr(self, mode_str + '_writer')
writer.add_image(
  f'{mode_str}/{series_ndx}_prediction_{slice_ndx}',
  image_a,
  self.totalTrainingSamples_count,
  dataformats='HWC',
)
```

This looks very similar to the `writer.add_scalar` calls we've seen before. The `dataformats='HWC'` argument tells TensorBoard that the order of axes in our image has our RGB channels as the third axis. Recall that our network layers often specify outputs that are $B \times C \times H \times W$, and we could put that data directly into TensorBoard as well if we specified `'CHW'`.

We also want to save the ground truth that we're using to train, which will form the top row of our TensorBoard CT slices we saw earlier in figure 13.16. The code for that is similar enough to what we just saw that we'll skip it. Again, check p2ch13/training.py if you want the details.

13.6.5 Updating our metrics logging

To give us an idea how we are doing, we compute per-epoch metrics: in particular, true positives, false negatives, and false positives. This is what the following listing does. Nothing here will be particularly surprising.

Listing 13.31 training.py:400, `.logMetrics`

```
sum_a = metrics_a.sum(axis=1)
allLabel_count = sum_a[METRICS_TP_NDX] + sum_a[METRICS_FN_NDX]
metrics_dict['percent_all/tp'] = \
  sum_a[METRICS_TP_NDX] / (allLabel_count or 1) * 100
metrics_dict['percent_all/fn'] = \
  sum_a[METRICS_FN_NDX] / (allLabel_count or 1) * 100
metrics_dict['percent_all/fp'] = \
  sum_a[METRICS_FP_NDX] / (allLabel_count or 1) * 100
```

Can be larger than 100% since we're comparing to the total number of pixels labeled as candidate nodules, which is a tiny fraction of each image

We are going to start scoring our models as a way to determine whether a particular training run is the best we've seen so far. In chapter 12, we said we'd be using the F1 score for our model ranking, but our goals are different here. We need to make sure our recall is as high as possible, since we can't classify a potential nodule if we don't find it in the first place!

We will use our recall to determine the "best" model. As long as the F1 score is reasonable for that epoch,[15] we just want to get recall as high as possible. Screening out any false positives will be the responsibility of the classification model.

Listing 13.32 training.py:393, `.logMetrics`

```
def logMetrics(self, epoch_ndx, mode_str, metrics_t):
  # ... line 453
  score = metrics_dict['pr/recall']

  return score
```

[15] And yes, "reasonable" is a bit of a dodge. "Nonzero" is a good starting place, if you'd like something more specific.

When we add similar code to our classification training loop in the next chapter, we'll use the F1 score.

Back in the main training loop, we'll keep track of the `best_score` we've seen so far in this training run. When we save our model, we'll include a flag that indicates whether this is the best score we've seen so far. Recall from section 13.6.4 that we're only calling the `doValidation` function for the first and then every fifth epochs. That means we're only going to check for a best score on those epochs. That shouldn't be a problem, but it's something to keep in mind if you need to debug something happening on epoch 7. We do this checking just before we save the images.

Listing 13.33 training.py:210, `SegmentationTrainingApp.main`

```
def main(self):
  best_score = 0.0
  for epoch_ndx in range(1, self.cli_args.epochs + 1):
      # if validation is wanted
      # ... line 233
      valMetrics_t = self.doValidation(epoch_ndx, val_dl)
      score = self.logMetrics(epoch_ndx, 'val', valMetrics_t)
      best_score = max(score, best_score)

      self.saveModel('seg', epoch_ndx, score == best_score)
```

The epoch-loop we already saw

Computes the score. As we saw earlier, we take the recall.

Now we only need to write saveModel. The third parameter is whether we want to save it as best model, too.

Let's take a look at how we persist our model to disk.

13.6.6 Saving our model

PyTorch makes it pretty easy to save our model to disk. Under the hood, `torch.save` uses the standard Python `pickle` library, which means we could pass our model instance in directly, and it would save properly. That's not considered the ideal way to persist our model, however, since we lose some flexibility.

Instead, we will save only the *parameters* of our model. Doing this allows us to load those parameters into any model that expects parameters of the same shape, even if the class doesn't match the model those parameters were saved under. The save-parameters-only approach allows us to reuse and remix our models in more ways than saving the entire model.

We can get at our model's parameters using the `model.state_dict()` function.

Listing 13.34 training.py:480, `.saveModel`

```
def saveModel(self, type_str, epoch_ndx, isBest=False):
  # ... line 496
  model = self.segmentation_model
  if isinstance(model, torch.nn.DataParallel):
    model = model.module
```

Gets rid of the DataParallel wrapper, if it exists

```
  state = {
    'sys_argv': sys.argv,
    'time': str(datetime.datetime.now()),
    'model_state': model.state_dict(),    <-- The important part
    'model_name': type(model).__name__,
    'optimizer_state' : self.optimizer.state_dict(),    <-- Preserves momentum, and so on
    'optimizer_name': type(self.optimizer).__name__,
    'epoch': epoch_ndx,
    'totalTrainingSamples_count': self.totalTrainingSamples_count,
  }
  torch.save(state, file_path)
```

We set `file_path` to something like `data-unversioned/part2/models/p2ch13/seg_2019-07-10_02.17.22_ch12.50000.state`. The `.50000.` part is the number of training samples we've presented to the model so far, while the other parts of the path are obvious.

> **TIP** By saving the optimizer state as well, we could resume training seamlessly. While we don't provide an implementation of this, it could be useful if your access to computing resources is likely to be interrupted. Details on loading a model and optimizer to restart training can be found in the official documentation (https://pytorch.org/tutorials/beginner/saving_loading_models.html).

If the current model has the best score we've seen so far, we save a second copy of `state` with a .best.state filename. This might get overwritten later by another, higher-score version of the model. By focusing only on this best file, we can divorce customers of our trained model from the details of how each epoch of training went (assuming, of course, that our score metric is of high quality).

Listing 13.35 training.py:514, `.saveModel`

```
if isBest:
  best_path = os.path.join(
    'data-unversioned', 'part2', 'models',
    self.cli_args.tb_prefix,
    f'{type_str}_{self.time_str}_{self.cli_args.comment}.best.state')
  shutil.copyfile(file_path, best_path)

  log.info("Saved model params to {}".format(best_path))

with open(file_path, 'rb') as f:
  log.info("SHA1: " + hashlib.sha1(f.read()).hexdigest())
```

We also output the SHA1 of the model we just saved. Similar to `sys.argv` and the timestamp we put into the state dictionary, this can help us debug exactly what model we're working with if things become confused later (for example, if a file gets renamed incorrectly).

We will update our classification training script in the next chapter with a similar routine for saving the classification model. In order to diagnose a CT, we'll need to have both models.

13.7 Results

Now that we've made all of our code changes, we've hit the last section in step 3 of figure 13.17. It's time to run `python -m p2ch13.training --epochs 20 --augmented final_seg`. Let's see what our results look like!

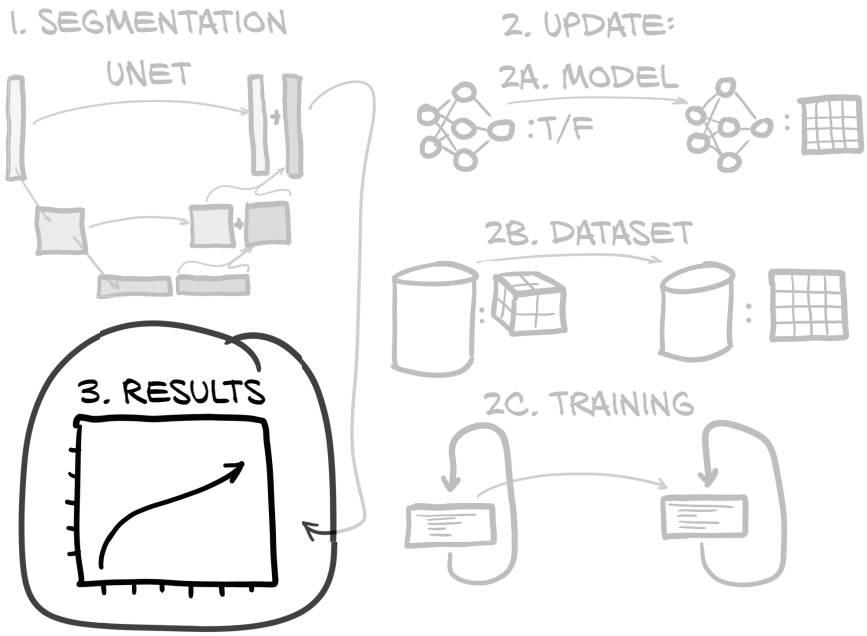


Figure 13.17 The outline of this chapter, with a focus on the results we see from training

Here is what our training metrics look like if we limit ourselves to the epochs we have validation metrics for (we'll be looking at those metrics next, so this will keep it an apples-to-apples comparison):

In these rows, we are particularly interested in the F1 score—it is trending up. Good!

TPs are trending up, too. Great! And FNs and FPs are trending down.

```
E1 trn      0.5235 loss, 0.2276 precision, 0.9381 recall, 0.3663 f1 score
E1 trn_all  0.5235 loss,  93.8% tp, 6.2% fn,     318.4% fp
...
E5 trn      0.2537 loss, 0.5652 precision, 0.9377 recall, 0.7053 f1 score
E5 trn_all  0.2537 loss,  93.8% tp, 6.2% fn,      72.1% fp
...
E10 trn      0.2335 loss, 0.6011 precision, 0.9459 recall, 0.7351 f1 score
E10 trn_all  0.2335 loss,  94.6% tp, 5.4% fn,      62.8% fp
```

```
...
E15 trn      0.2226 loss, 0.6234 precision, 0.9536 recall, 0.7540 f1 score
E15 trn_all  0.2226 loss,  95.4% tp, <2>  4.6% fn,      57.6% fp
 ...
E20 trn      0.2149 loss, 0.6368 precision, 0.9584 recall, 0.7652 f1 score
E20 trn_all  0.2149 loss,  95.8% tp, <2>  4.2% fn,      54.7% fp
```

In these rows, we are particularly interested in the F1 score—it is trending up. Good!

TPs are trending up, too. Great! And FNs and FPs are trending down.

Overall, it looks pretty good. True positives and the F1 score are trending up, false positives and negatives are trending down. That's what we want to see! The validation metrics will tell us whether these results are legitimate. Keep in mind that since we're training on 64 × 64 crops, but validating on whole 512 × 512 CT slices, we are almost certainly going to have drastically different TP:FN:FP ratios. Let's see:

The highest TP rate (great). Note that the TP rate is the same as recall. But FPs are 4495%—that sounds like a lot.

```
E1 val      0.9441 loss, 0.0219 precision, 0.8131 recall, 0.0426 f1 score
E1 val_all  0.9441 loss,  81.3% tp,  18.7% fn,    3637.5% fp

E5 val      0.9009 loss, 0.0332 precision, 0.8397 recall, 0.0639 f1 score
E5 val_all  0.9009 loss,  84.0% tp,  16.0% fn,    2443.0% fp

E10 val      0.9518 loss, 0.0184 precision, 0.8423 recall, 0.0360 f1 score
E10 val_all  0.9518 loss,  84.2% tp,  15.8% fn,    4495.0% fp

E15 val      0.8100 loss, 0.0610 precision, 0.7792 recall, 0.1132 f1 score
E15 val_all  0.8100 loss,  77.9% tp,  22.1% fn,    1198.7% fp

E20 val      0.8602 loss, 0.0427 precision, 0.7691 recall, 0.0809 f1 score
E20 val_all  0.8602 loss,  76.9% tp,  23.1% fn,    1723.9% fp
```

Ouch—false positive rates over 4,000%? Yes, actually, that's expected. Our validation slice area is 2^{18} pixels (512 is 2^9), while our training crop is only 2^{12}. That means we're validating on a slice surface that's $2^6 = 64$ times bigger! Having a false positive count that's also 64 times bigger makes sense. Remember that our true positive rate won't have changed meaningfully, since it would all have been included in the 64 × 64 sample we trained on in the first place. This situation also results in very low precision, and, hence, a low F1 score. That's a natural result of how we've structured the training and validation, so it's not a cause for alarm.

What's problematic, however, is our recall (and, hence, our true positive rate). Our recall plateaus between epochs 5 and 10 and then starts to drop. It's pretty obvious that we begin overfitting very quickly, and we can see further evidence of that in figure 13.18—while the training recall keeps trending upward, the validation recall decreases after 3 million samples. This is how we identified overfitting in chapter 5, in particular figure 5.14.

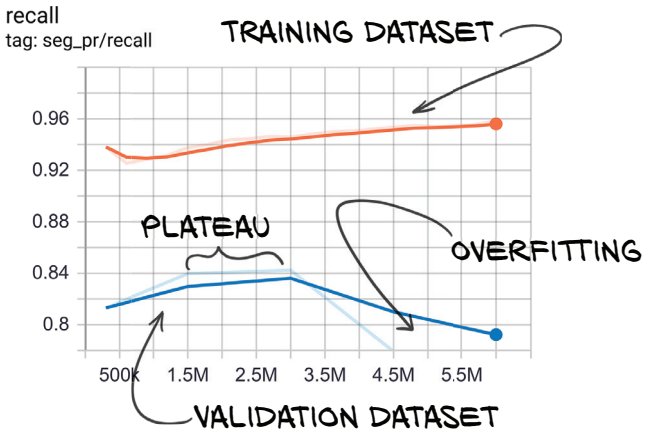


Figure 13.18 The validation set recall, showing signs of overfitting when recall goes down after epoch 10 (3 million samples)

NOTE Always keep in mind that TensorBoard will smooth your data lines by default. The lighter ghost line behind the solid color shows the raw values.

The U-Net architecture has a lot of capacity, and even with our reduced filter and depth counts, it's able to memorize our training set pretty quickly. One upside is that we don't end up needing to train the model for very long!

Recall is our top priority for segmentation, since we'll let issues with precision be handled downstream by the classification models. Reducing those false positives is the entire reason we have those classification models! This skewed situation does mean it is more difficult than we'd like to evaluate our model. We could instead use the F2 score, which weights recall more heavily (or F5, or F10 …), but we'd have to pick an N high enough to almost completely discount precision. We'll skip the intermediates and just score our model by recall, and use our human judgment to make sure a given training run isn't being pathological about it. Since we're training on the Dice loss, rather than directly on recall, it should work out.

This is one of the situations where we are cheating a little, because we (the authors) have already done the training and evaluation for chapter 14, and we know how all of this is going to turn out. There isn't any good way to look at this situation and *know* that the results we're seeing will work. Educated guesses are helpful, but they are no substitute for actually running experiments until something clicks.

As it stands, our results are good enough to use going forward, even if our metrics have some pretty extreme values. We're one step closer to finishing our end-to-end project!

13.8 Conclusion

In this chapter, we've discussed a new way of structuring models for pixel-to-pixel segmentation; introduced U-Net, an off-the-shelf, proven model architecture for those kinds of tasks; and adapted an implementation for our own use. We've also changed our dataset to provide data for our new model's training needs, including small crops

for training and a limited set of slices for validation. Our training loop now has the ability to save images to TensorBoard, and we have moved augmentation from the dataset into a separate model that can operate on the GPU. Finally, we looked at our training results and discussed how even though the false positive rate (in particular) looks different from what we might hope, our results will be acceptable given our requirements for them from the larger project. In chapter 14, we will pull together the various models we've written into a cohesive, end-to-end whole.

13.9 Exercises

1. Implement the model-wrapper approach to augmentation (like what we used for segmentation training) for the classification model.
   a. What compromises did you have to make?
   b. What impact did the change have on training speed?
2. Change the segmentation `Dataset` implementation to have a three-way split for training, validation, and test sets.
   a. What fraction of the data did you use for the test set?
   b. Do performance on the test set and the validation set seem consistent with each other?
   c. How badly does training suffer with the smaller training set?
3. Make the model try to segment malignant versus benign in addition to is-nodule status.
   a. How does your metrics reporting need to change? Your image generation?
   b. What kind of results do you see? Is the segmentation good enough to skip the classification step?
4. Can you train the model on a combination of 64 × 64 crops and whole-CT slices?[16]
5. Can you find additional sources of data to use beyond just the LUNA (or LIDC) data?

13.10 Summary

- Segmentation flags individual pixels or voxels for membership in a class. This is in contrast to classification, which operates at the level of the entire image.
- U-Net was a breakthrough model architecture for segmentation tasks.
- Using segmentation followed by classification, we can implement detection with relatively modest data and computation requirements.
- Naive approaches to 3D segmentation can quickly use too much RAM for current-generation GPUs. Carefully limiting the scope of what is presented to the model can help limit RAM usage.

[16] Hint: Each sample tuple to be batched together must have the same shape for each corresponding tensor, but the next batch could have different samples with different shapes.

- It is possible to train a segmentation model on image crops while validating on whole-image slices. This flexibility can be important for class balancing.
- Loss weighting is an emphasis on the loss computed from certain classes or subsets of the training data, to encourage the model to focus on the desired results. It can complement class balancing and is a useful tool when trying to tweak model training performance.
- TensorBoard can display 2D images generated during training and will save a history of how those models changed over the training run. This can be used to visually track changes to model output as training progresses.
- Model parameters can be saved to disk and loaded back to reconstitute a model that was saved earlier. The exact model implementation can change as long as there is a 1:1 mapping between old and new parameters.

14 End-to-end nodule analysis, and where to go next

This chapter covers

- Connecting segmentation and classification models
- Fine-tuning a network for a new task
- Adding histograms and other metric types to TensorBoard
- Getting from overfitting to generalizing

Over the past several chapters, we have built a decent number of systems that are important components of our project. We started loading our data, built and improved classifiers for nodule candidates, trained segmentation models to find those candidates, handled the support infrastructure needed to train and evaluate those models, and started saving the results of our training to disk. Now it's time to unify the components we have into a cohesive whole, so that we may realize the full goal of our project: it's time to automatically detect cancer.

14.1 Towards the finish line

We can get a hint of the work remaining by looking at figure 14.1. In step 3 (grouping) we see that we still need to build the bridge between the segmentation model from chapter 13 and the classifier from chapter 12 that will tell us whether what the segmentation network found is, indeed, a nodule. On the right is step 5 (nodule analysis and diagnosis), the last step to the overall goal: seeing whether a nodule is cancer. This is another classification task; but to learn something in the process, we'll take a fresh angle at how to approach it by building on the nodule classifier we already have.

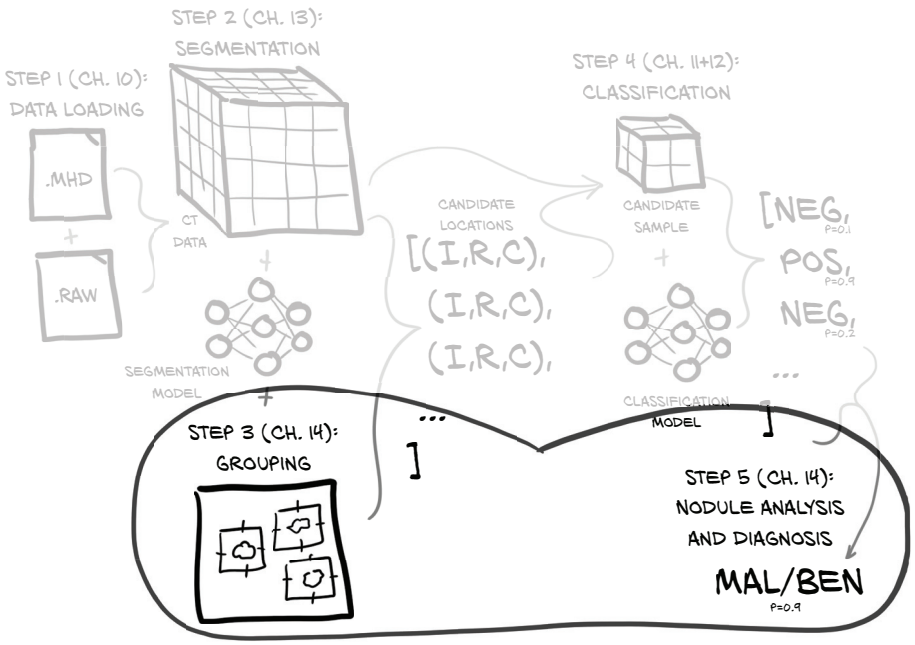


Figure 14.1 Our end-to-end lung cancer detection project, with a focus on this chapter's topics: steps 3 and 5, grouping and nodule analysis

Of course, these brief descriptions and their simplified depiction in figure 14.1 leave out a lot of detail. Let's zoom in a little with figure 14.2 and see what we've got left to accomplish.

As you can see, three important tasks remain. Each item in the following list corresponds to a major line item from figure 14.2:

1. *Generate nodule candidates.* This is step 3 in the overall project. Three tasks go into this step:
   a. *Segmentation*—The segmentation model from chapter 13 will predict if a given pixel is of interest: if we suspect it is part of a nodule. This will be done per 2D slice, and every 2D result will be stacked to form a 3D array of voxels containing nodule candidate predictions.

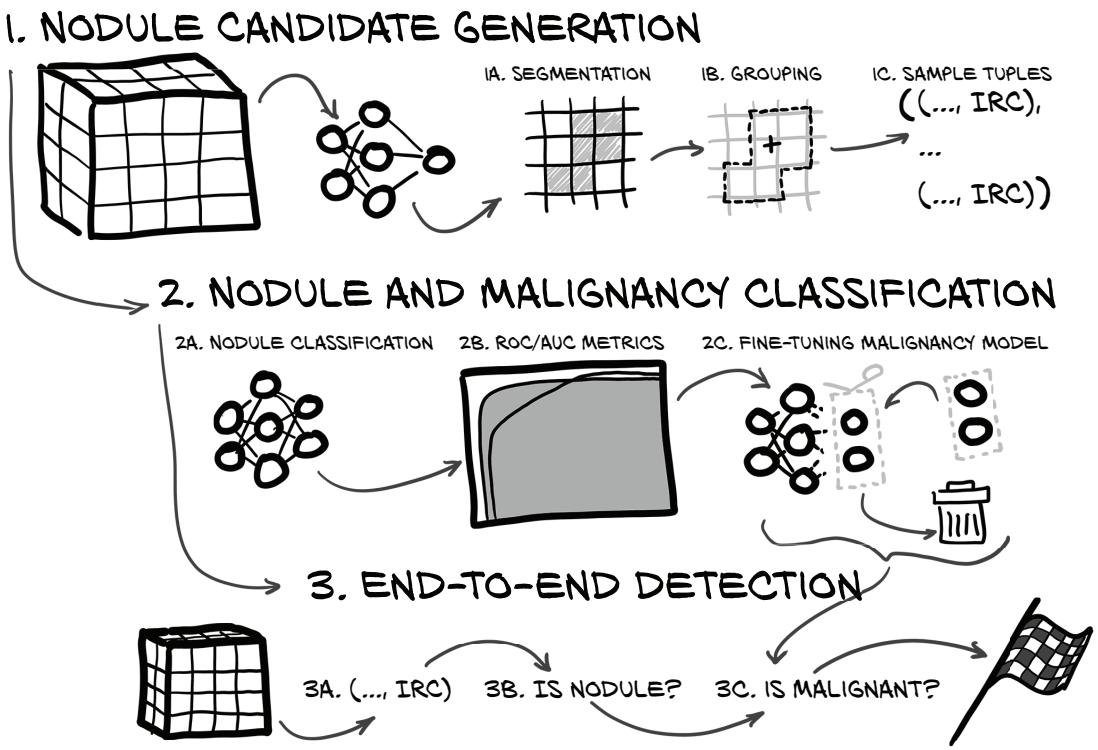


Figure 14.2 A detailed look at the work remaining for our end-to-end project

b *Grouping*—We will group the voxels into nodule candidates by applying a threshold to the predictions, and then grouping connected regions of flagged voxels.

c *Constructing sample tuples*—Each identified nodule candidate will be used to construct a sample tuple for classification. In particular, we need to produce the coordinates (index, row, column) of that nodule's center.

Once this is achieved, we will have an application that takes a raw CT scan from a patient and produces a list of detected nodule candidates. Producing such a list is the task in the LUNA challenge. If this project were to be used clinically (and we reemphasize that our project should not be!), this nodule list would be suitable for closer inspection by a doctor.

2 *Classify nodules and malignancy.* We'll take the nodule candidates we just produced and pass them to the candidate classification step we implemented in chapter 12, and then perform malignancy detection on the candidates flagged as nodules:

a *Nodule classification*—Each nodule candidate from segmentation and grouping will be classified as either nodule or non-nodule. Doing so will allow us to screen out the many normal anatomical structures flagged by our segmentation process.

b *ROC/AUC metrics*—Before we can start our last classification step, we'll define some new metrics for examining the performance of classification models, as well as establish a baseline metric against which to compare our malignancy classifiers.

c *Fine-tuning the malignancy model*—Once our new metrics are in place, we will define a model specifically for classifying benign and malignant nodules, train it, and see how it performs. We will do the training by fine-tuning: a process that cuts out some of the weights of an existing model and replaces them with fresh values that we then adapt to our new task.

At that point we will be within arm's reach of our ultimate goal: to classify nodules into benign and malignant classes and then derive a diagnosis from the CT. Again, diagnosing lung cancer in the real world involves much more than staring at a CT scan, so our performing this diagnosis is more an experiment to see how far we can get using deep learning and imaging data alone.

3 *End-to-end detection*. Finally, we will put all of this together to get to the finish line, combining the components into an end-to-end solution that can look at a CT and answer the question "Are there malignant nodules present in the lungs?"

a *IRC*—We will segment our CT to get nodule candidate samples to classify.

b *Determine the nodules*—We will perform nodule classification on the candidate to determine whether it should be fed into the malignancy classifier.

c *Determine malignancy*—We will perform malignancy classification on the nodules that pass through the nodule classifier to determine whether the patient has cancer.

We've got a lot to do. To the finish line!

NOTE As in the previous chapter, we will discuss the key concepts in detail in the text and leave out the code for repetitive, tedious, or obvious parts. Full details can be found in the book's code repository.

14.2 Independence of the validation set

We are in danger of making a subtle but critical mistake, which we need to discuss and avoid: we have a potential leak from the training set to the validation set! For each of the segmentation and classification models, we took care of splitting the data into a training set and an independent validation set by taking every tenth example for validation and the remainder for training.

However, the split for the classification model was done on the list of nodules, and the split for the segmentation model was done on the list of CT scans. This means we likely have nodules from the segmentation validation set in the training set of the classification model and vice versa. We must avoid that! If left unfixed, this situation could lead to performance figures that would be artificially higher compared to what we

would obtain on an independent dataset. This is called a *leak*, and it would invalidate our validation.

To rectify this potential data leak, we need to rework the classification dataset to also work at the CT scan level, just as we did for the segmentation task in chapter 13. Then we need to retrain the classification model with this new dataset. On the bright side, we didn't save our classification model earlier, so we would have to retrain anyway.

Your takeaway from this should be to keep an eye on the end-to-end process when defining the validation set. Probably the easiest way to do this (and the way it is done for most important datasets) is to make the validation split as explicit as possible—for example, by having two separate directories for training and validation—and then stick to this split for your entire project. When you need to redo the split (for example, when you need to add stratification of the dataset split by some criterion), you need to retrain all of your models with the newly split dataset.

So what we did for you was to take `LunaDataset` from chapters 10–12 and copy over getting the candidate list and splitting it into test and validation datasets from `Luna2dSegmentationDataset` in chapter 13. As this is very mechanical, and there is not much to learn from the details (you are a dataset pro by now), we won't show the code in detail.

We'll retrain our classification model by rerunning the training for the classifier:[1]

```
$ python3 -m p2ch14.training --num-workers=4 --epochs 100 nodule-nonnodule
```

After 100 epochs, we achieve about 95% accuracy for positive samples and 99% for negative ones. As the validation loss isn't seen to be trending upward again, we could train the model longer to see if things continued to improve.

After 90 epochs, we reach the maximal F1 score and have 99.2% validation accuracy, albeit only 92.8% on the actual nodules. We'll take this model, even though we might also try to trade a bit of overall accuracy for better accuracy on the malignant nodules (in between, the model got 95.4% accuracy on actual nodules for 98.9% total accuracy). This will be good enough for us, and we are ready to bridge the models.

14.3 Bridging CT segmentation and nodule candidate classification

Now that we have a segmentation model saved from chapter 13 and a classification model we just trained in the previous section, figure 14.3, steps 1a, 1b, and 1c show that we're ready to work on writing the code that will convert our segmentation output into sample tuples. We are doing the *grouping*: finding the dashed outline around the highlight of step 1b in figure 14.3. Our input is the *segmentation*: the voxels flagged by the segmentation model in 1a. We want to find 1c, the coordinates of the center of mass of each "lump" of flagged voxels: the index, row, and column of the 1b plus mark is what we need to provide in the list of sample tuples as output.

[1] You can also use the p2_run_everything notebook.

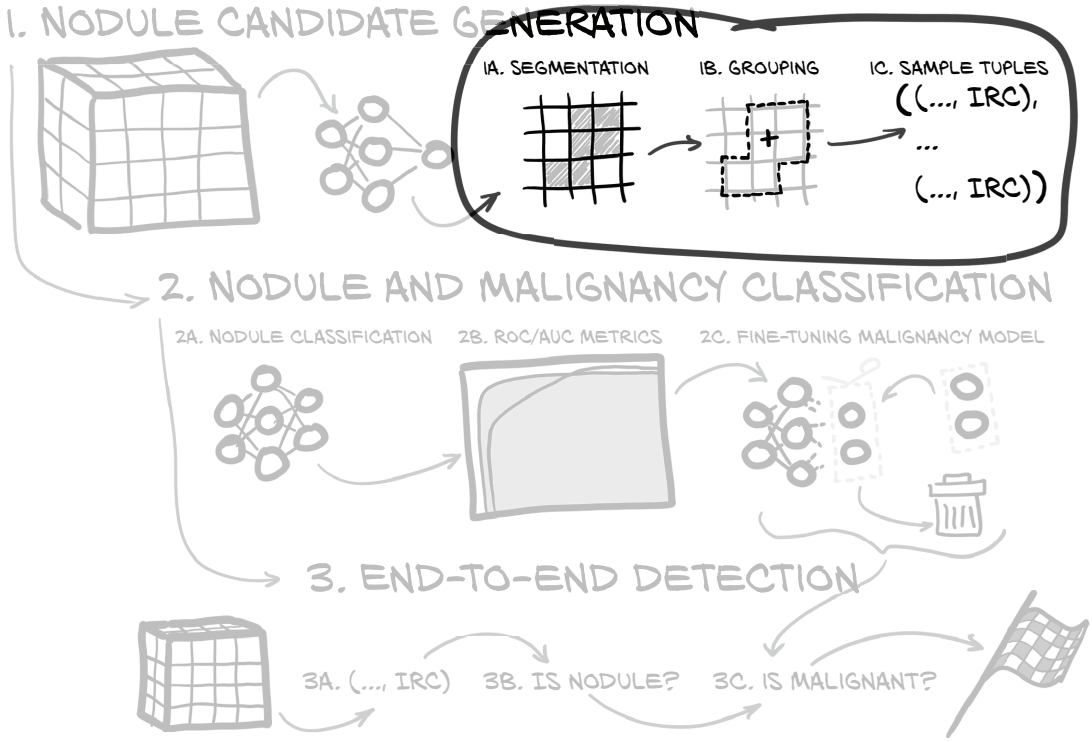


Figure 14.3 Our plan for this chapter, with a focus on grouping segmented voxels into nodule candidates

Running the models will naturally look very similar to how we handled them during training and validation (validation in particular). The difference here is the loop over the CTs. For each CT, we segment *every* slice and then take all the segmented output as the input to grouping. The output from grouping will be fed into a nodule classifier, and the nodules that survive that classification will be fed into a malignancy classifier.

This is accomplished by the following outer loop over the CTs, which for each CT segments, groups, classifies candidates, and provides the classifications for further processing.

Listing 14.1 nodule_analysis.py:324, `NoduleAnalysisApp.main`

```
for _, series_uid in series_iter:
  ct = getCt(series_uid)
  mask_a = self.segmentCt(ct, series_uid)

  candidateInfo_list = self.groupSegmentationOutput(
    series_uid, ct, mask_a)
  classifications_list = self.classifyCandidates(
    ct, candidateInfo_list)
```

Loops over the series UIDs

Gets the CT (step 1 in the big picture)

Runs our segmentation model on it (step 2)

Groups the flagged voxels in the output (step 3)

Runs our nodule classifier on them (step 4)

We'll break down the `segmentCt`, `groupSegmentationOutput`, and `classifyCandidates` methods in the following sections.

14.3.1 Segmentation

First up, we are going to perform segmentation on every slice of the entire CT scan. As we need to feed a given patient's CT slice by slice, we build a `Dataset` that loads a CT with a single `series_uid` and returns each slice, one per `__getitem__` call.

> **NOTE** The segmentation step in particular can take quite a while when executed on the CPU. Even though we gloss over it here, the code will use the GPU if available.

Other than the more expansive input, the main difference is what we do with the output. Recall that the output is an array of per-pixel probabilities (that is, in the range 0…1) that the given pixel is part of a nodule. While iterating over the slices, we collect the slice-wise predictions in a mask array that has the same shape as our CT input. Afterward, we threshold the predictions to get a binary array. We will use a threshold of 0.5, but if we wanted to, we could experiment with thresholding to trade getting more true positives for an increase in false positives.

We also include a small cleanup step using the erosion operation from `scipy.ndimage.morphology`. It deletes one layer of boundary voxels and only keeps the inner ones—those for which all eight neighboring voxels in the axis direction are also flagged. This makes the flagged area smaller and causes very small components (smaller than $3 \times 3 \times 3$ voxels) to vanish. Put together with the loop over the data loader, which we instruct to feed us all slices from a single CT, we have the following.

Listing 14.2 nodule_analysis.py:384, `.segmentCt`

```
    def segmentCt(self, ct, series_uid):
        with torch.no_grad():
            output_a = np.zeros_like(ct.hu_a, dtype=np.float32)
            seg_dl = self.initSegmentationDl(series_uid)
            for input_t, _, _, slice_ndx_list in seg_dl:

                input_g = input_t.to(self.device)
                prediction_g = self.seg_model(input_g)

                for i, slice_ndx in enumerate(slice_ndx_list):
                    output_a[slice_ndx] = prediction_g[i].cpu().numpy()

            mask_a = output_a > 0.5
            mask_a = morphology.binary_erosion(mask_a, iterations=1)

        return mask_a
```

We do not need gradients here, so we don't build the graph.

This array will hold our output: a float array of probability annotations.

We get a data loader that lets us loop over our CT in batches.

After moving the input to the GPU …

… we run the segmentation model …

… and copy each element to the output array.

Thresholds the probability outputs to get a binary output, and then applies binary erosion as cleanup

This was easy enough, but now we need to invent the grouping.

14.3.2 Grouping voxels into nodule candidates

We are going to use a simple connected-components algorithm for grouping our suspected nodule voxels into chunks to feed into classification. This grouping approach labels connected components, which we will accomplish using `scipy.ndimage.measurements.label`. The `label` function will take all nonzero pixels that share an edge with another nonzero pixel and mark them as belonging to the same group. Since our output from the segmentation model has mostly blobs of highly adjacent pixels, this approach matches our data well.

Listing 14.3 nodule_analysis.py:401

```
def groupSegmentationOutput(self, series_uid,  ct, clean_a):
  candidateLabel_a, candidate_count = measurements.label(clean_a)
  centerIrc_list = measurements.center_of_mass(
    ct.hu_a.clip(-1000, 1000) + 1001,
    labels=candidateLabel_a,
    index=np.arange(1, candidate_count+1),
  )
```

Assigns each voxel the label of the group it belongs to

Gets the center of mass for each group as index, row, column coordinates

The output array `candidateLabel_a` is the same shape as `clean_a`, which we used for input, but it has 0 where the background voxels are, and increasing integer labels 1, 2, …, with one number for each of the connected blobs of voxels making up a nodule candidate. Note that the labels here are *not* the same as labels in a classification sense! These are just saying "This blob of voxels is blob 1, this blob over here is blob 2, and so on."

SciPy also sports a function to get the centers of mass of the nodule candidates: `scipy.ndimage.measurements.center_of_mass`. It takes an array with per-voxel densities, the integer labels from the `label` function we just called, and a list of which of those labels need to have a center calculated. To match the function's expectation that the mass is non-negative, we offset the (clipped) `ct.hu_a` by 1,001. Note that this leads to all flagged voxels carrying some weight, since we clamped the lowest air value to –1,000 HU in the native CT units.

Listing 14.4 nodule_analysis.py:409

```
candidateInfo_list = []
for i, center_irc in enumerate(centerIrc_list):
  center_xyz = irc2xyz(
    center_irc,
    ct.origin_xyz,
    ct.vxSize_xyz,
    ct.direction_a,
  )
```

Converts the voxel coordinates to real patient coordinates

```
    candidateInfo_tup = \
      CandidateInfoTuple(False, False, False, 0.0, series_uid, center_xyz)
    candidateInfo_list.append(candidateInfo_tup)

return candidateInfo_list
```

Builds our candidate info tuple and appends it to the list of detections

As output, we get a list of three arrays (one each for the index, row, and column) the same length as our `candidate_count`. We can use this data to populate a list of `candidateInfo_tup` instances; we have grown attached to this little data structure, so we stick our results into the same kind of list we've been using since chapter 10. As we don't really have suitable data for the first four values (`isNodule_bool`, `hasAnnotation_bool`, `isMal_bool`, and `diameter_mm`), we insert placeholder values of a suitable type. We then convert our coordinates from voxels to physical coordinates in a loop, creating the list. It might seem a bit silly to move our coordinates away from our array-based index, row, and column, but all of the code that consumes `candidateInfo_tup` instances expects `center_xyz`, not `center_irc`. We'd get wildly wrong results if we tried to swap one for the other!

Yay—we've conquered step 3, getting nodule locations from the voxel-wise detections! We can now crop out the suspected nodules and feed them to our classifier to weed out some more false positives.

14.3.3 Did we find a nodule? Classification to reduce false positives

As we started part 2 of this book, we described the job of a radiologist looking through CT scans for signs of cancer thus:

> *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."*

We've spent time and energy discussing the proverbial needles; let's discuss the hay for a moment by looking at figure 14.4. Our job, so to speak, is to fork away as much hay as we can from in front of our glassy-eyed radiologist, so that they can refocus their highly trained attention where it can do the most good.

Let's look at how much we are discarding at each step while we perform our end-to-end diagnosis. The arrows in figure 14.4 show the data as it flows from the raw CT voxels through our project to our final malignancy determination. Each arrow that ends with an X indicates a swath of data discarded by the previous step; the arrow pointing to the next step represents the data that survived the culling. Note that the numbers here are *very* approximate.

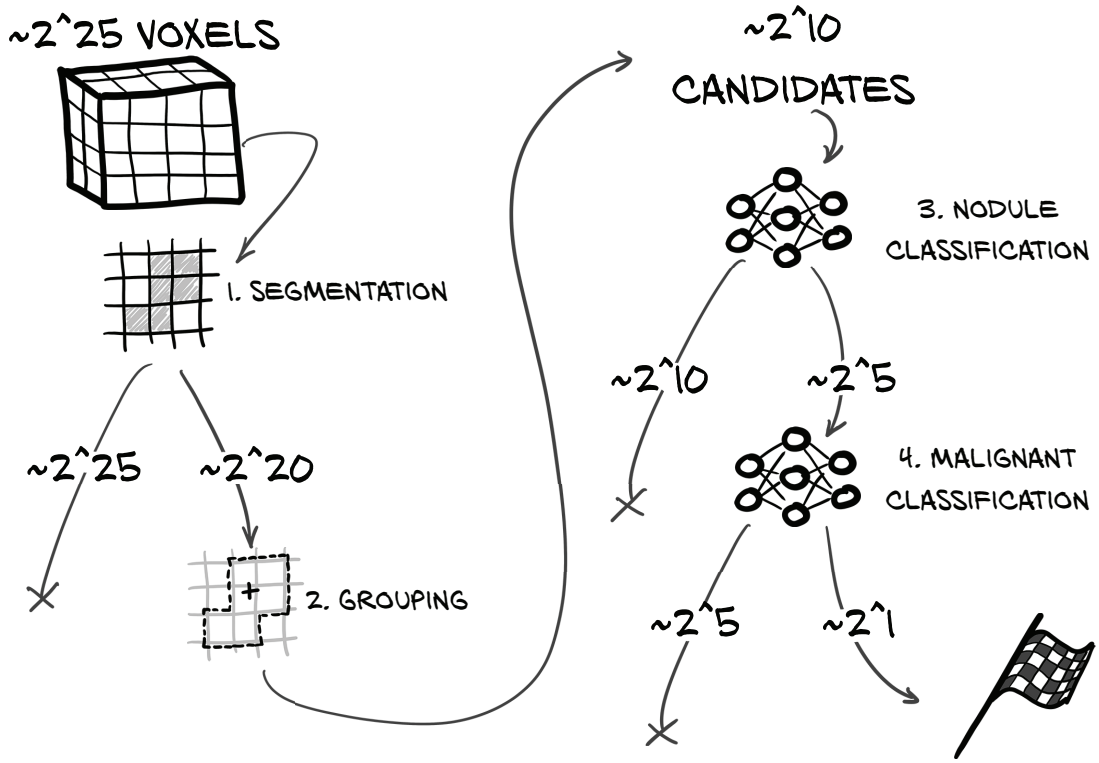


Figure 14.4 The steps of our end-to-end detection project, and the rough order of magnitude of data removed at each step

Let's go through the steps in figure 14.4 in more detail:

1. *Segmentation*—Segmentation starts with the entire CT: hundreds of slices, or about 33 million (2^{25}) voxels (give or take quite a lot). About 2^{20} voxels are flagged as being of interest; this is orders of magnitude smaller than the total input, which means we're throwing out 97% of the voxels (that's the 2^{25} on the left leading to the X).
2. *Grouping*. While grouping doesn't remove anything explicitly, it does reduce the number of items we're considering, since we consolidate voxels into nodule candidates. The grouping produces about 1,000 candidates (2^{10}) from 1 million voxels. A nodule of $16 \times 16 \times 2$ voxels would have a total of 2^{10} voxels.[2]
3. *Nodule classification*. This process throws away the majority of the remaining ~2^{10} items. From our thousands of nodule candidates, we're left with tens of nodules: about 2^5.
4. *Malignant classification*. Finally, the malignancy classifier takes tens of nodules (2^5) and finds the one or two (2^1) that are cancer.

[2] The size of any given nodule is highly variable, obviously.

Each step along the way allows us to discard a huge amount of data that our model is confident is irrelevant to our cancer-detection goal. We went from millions of data points to a handful of tumors.

Fully automated vs. assistive systems

There is a difference between a fully automated system and one that is designed to augment a human's abilities. For our automated system, once a piece of data is flagged as irrelevant, it is gone forever. When presenting data for a human to consume, however, we should allow them to peel back some of the layers and look at the near misses, as well as annotate our findings with a degree of confidence. Were we designing a system for clinical use, we'd need to carefully consider our exact intended use and make sure our system design supported those use cases well. Since our project is fully automated, we can move forward without having to consider how best to surface the near misses and the unsure answers.

Now that we have identified regions in the image that our segmentation model considers probable candidates, we need to crop these candidates from the CT and feed them into the classification module. Happily, we have `candidateInfo_list` from the previous section, so all we need to do is make a `DataSet` from it, put it into a `DataLoader`, and iterate over it. Column 1 of the probability predictions is the predicted probability that this is a nodule and is what we want to keep. Just as before, we collect the output from the entire loop.

Listing 14.5 nodule_analysis.py:357, `.classifyCandidates`

```
    def classifyCandidates(self, ct, candidateInfo_list):
      cls_dl = self.initClassificationDl(candidateInfo_list)
      classifications_list = []
      for batch_ndx, batch_tup in enumerate(cls_dl):
        input_t, _, _, series_list, center_list = batch_tup

        input_g = input_t.to(self.device)
        with torch.no_grad():
          _, probability_nodule_g = self.cls_model(input_g)
          if self.malignancy_model is not None:
            _, probability_mal_g = self.malignancy_model(input_g)
          else:
            probability_mal_g = torch.zeros_like(probability_nodule_g)

        zip_iter = zip(center_list,
          probability_nodule_g[:,1].tolist(),
          probability_mal_g[:,1].tolist())
        for center_irc, prob_nodule, prob_mal in zip_iter:
          center_xyz = irc2xyz(center_irc,
            direction_a=ct.direction_a,
```

Again, we get a data loader to loop over, this time based on our candidate list.

Sends the inputs to the device

Runs the inputs through the nodule vs. non-nodule network

If we have a malignancy model, we run that, too.

Does our bookkeeping, constructing a list of our results

```
            origin_xyz=ct.origin_xyz,
            vxSize_xyz=ct.vxSize_xyz,
          )
          cls_tup = (prob_nodule, prob_mal, center_xyz, center_irc)
          classifications_list.append(cls_tup)
    return classifications_list
```

This is great! We can now threshold the output probabilities to get a list of things our model thinks are actual nodules. In a practical setting, we would probably want to output them for a radiologist to inspect. Again, we might want to adjust the threshold to err a bit on the safe side: that is, if our threshold was 0.3 instead of 0.5, we would present a few more candidates that turn out not to be nodules, while reducing the risk of missing actual nodules.

Listing 14.6 nodule_analysis.py:333, `NoduleAnalysisApp.main`

If we don't pass run_validation, we print individual information …

```
    if not self.cli_args.run_validation:
      print(f"found nodule candidates in {series_uid}:")
      for prob, prob_mal, center_xyz, center_irc in classifications_list:
        if prob > 0.5:
          s = f"nodule prob {prob:.3f}, "
          if self.malignancy_model:
            s += f"malignancy prob {prob_mal:.3f}, "
          s += f"center xyz {center_xyz}"
          print(s)

    if series_uid in candidateInfo_dict:
      one_confusion = match_and_score(
        classifications_list, candidateInfo_dict[series_uid]
      )
      all_confusion += one_confusion
      print_confusion(
        series_uid, one_confusion, self.malignancy_model is not None
      )

print_confusion(
  "Total", all_confusion, self.malignancy_model is not None
)
```

… for all candidates found by the segmentation where the classifier assigned a nodule probability of 50% or more.

If we have the ground truth data, we compute and print the confusion matrix and also add the current results to the total.

Let's run this for a given CT from the validation set:[3]

```
$ python3.6 -m p2ch14.nodule_analysis 1.3.6.1.4.1.14519.5.2.1.6279.6001
➥ .592821488053137951302246128864
...
found nodule candidates in 1.3.6.1.4.1.14519.5.2.1.6279.6001.5928214880
➥ 53137951302246128864:
```

[3] We chose this series specifically because it has a nice mix of results.

This candidate is assigned a 53% probability of being malignant, so it barely makes the probability threshold of 50%. The malignancy classification assigns a very low (3%) probability.

```
nodule prob 0.533, malignancy prob 0.030, center xyz XyzTuple
➥ (x=-128.857421875, y=-80.349609375, z=-31.300007820129395)
nodule prob 0.754, malignancy prob 0.446, center xyz XyzTuple
➥ (x=-116.396484375, y=-168.142578125, z=-238.30000233650208)
...
nodule prob 0.974, malignancy prob 0.427, center xyz XyzTuple
➥ (x=121.494140625, y=-45.798828125, z=-211.3000030517578)
nodule prob 0.700, malignancy prob 0.310, center xyz XyzTuple
➥ (x=123.759765625, y=-44.666015625, z=-211.3000030517578)
...
```

Detected as a nodule with very high confidence and assigned a 42% probability of malignancy

The script found 16 nodule candidates in total. Since we're using our validation set, we have a full set of annotations and malignancy information for each CT, which we can use to create a confusion matrix with our results. The rows are the truth (as defined by the annotations), and the columns show how our project handled each case:

Scan ID

Prognosis: Complete Miss means the segmentation didn't find a nodule, Filtered Out is the classifier's work, and Predicted Nodules are those it marked as nodules.

```
1.3.6.1.4.1.14519.5.2.1.6279.6001.592821488053137951302246128864
                  |    Complete Miss |     Filtered Out |     Pred. Nodule
    Non-Nodules |                  |             1088 |                15
         Benign |                1 |                0 |                 0
      Malignant |                0 |                0 |                 1
```

The rows contain the ground truth.

The Complete Miss column is when our segmenter did not flag a nodule at all. Since the segmenter was not trying to flag non-nodules, we leave that cell blank. Our segmenter was trained to have high recall, so there are a large number of non-nodules, but our nodule classifier is well equipped to screen those out.

So we found the 1 malignant nodule in this scan, but missed a 17th benign one. In addition, 15 false positive non-nodules made it through the nodule classifier. The filtering by the classifier brought the false positives down from over 1,000! As we saw earlier, 1,088 is about $O(2^{10})$, so that lines up with what we expect. Similarly, 15 is about $O(2^4)$, which isn't far from the $O(2^5)$ we ballparked.

Cool! But what's the larger picture?

14.4 Quantitative validation

Now that we have anecdotal evidence that the thing we built might be working on one case, let's take a look at the performance of our model on the entire validation set. Doing so is simple: we run our validation set through the previous prediction and check how many nodules we get, how many we miss, and how many candidates are erroneously identified as nodules.

We run the following, which should take half an hour to an hour when run on the GPU. After coffee (or a full-blown nap), here is what we get:

```
$ python3 -m p2ch14.nodule_analysis --run-validation

...
Total
                 |    Complete Miss |     Filtered Out |     Pred. Nodule
     Non-Nodules |                  |           164893 |             2156
          Benign |               12 |                3 |               87
       Malignant |                1 |                6 |               45
```

We detected 132 of the 154 nodules, or 85%. Of the 22 we missed, 13 were not considered candidates by the segmentation, so this would be the obvious starting point for improvements.

About 95% of the detected nodules are false positives. This is of course not great; on the other hand, it's a lot less critical—having to look at 20 nodule candidates to find one nodule will be much easier than looking at the entire CT. We will go into this in more detail in section 14.7.2, but we want to stress that rather than treating these mistakes as a black box, it's a good idea to investigate the misclassified cases and see if they have commonalities. Are there characteristics that differentiate them from the samples that were correctly classified? Can we find anything that could be used to improve our performance?

For now, we're going to accept our numbers as is: not bad, but not perfect. The exact numbers may differ when you run your self-trained model. Toward the end of this chapter, we will provide some pointers to papers and techniques that can help improve these numbers. With inspiration and some experimentation, we are confident that you can achieve better scores than we show here.

14.5 Predicting malignancy

Now that we have implemented the nodule-detection task of the LUNA challenge and can produce our own nodule predictions, we ask ourselves the logical next question: can we distinguish malignant nodules from benign ones? We should say that even with a good system, diagnosing malignancy would probably take a more holistic view of the patient, additional non-CT context, and eventually a biopsy, rather than just looking at single nodules in isolation on a CT scan. As such, this seems to be a task that is likely to be performed by a doctor for some time to come.

14.5.1 Getting malignancy information

The LUNA challenge focuses on nodule detection and does not come with malignancy information. The LIDC-IDRI dataset (http://mng.bz/4A4R) has a superset of the CT scans used for the LUNA dataset and includes additional information about the degree of malignancy of the identified tumors. Conveniently, there is a PyLIDC library that can be installed easily, as follows:

```
$ pip3 install pylidc
```

The `pylicd` library gives us ready access to the additional malignancy information we want. Just like matching the annotations with the candidates by location as we did in chapter 10, we need to associate the annotation information from LIDC with the coordinates of the LUNA candidates.

In the LIDC annotations, the malignancy information is encoded per nodule and diagnosing radiologist (up to four looked at the same nodule) using an ordinal five-value scale from 1 (highly unlikely) through moderately unlikely, indeterminate, and moderately suspicious, and ending with 5 (highly suspicious).[4] These annotations are based on the image alone and subject to assumptions about the patient. To convert the list of numbers to a single Boolean yes/no, we will consider nodules to be malignant when at least two radiologists rated that nodule as "moderately suspicious" or greater. Note that this criterion is somewhat arbitrary; indeed, the literature has many different ways of dealing with this data, including predicting the five steps, using averages, or removing nodules from the dataset where the rating radiologists were uncertain or disagreed.

The technical aspects of combining the data are the same as in chapter 10, so we skip showing the code here (it is in the code repository for this chapter) and will use the extended CSV file. We will use the dataset in a way very similar to what we did for the nodule classifier, except that we now only need to process actual nodules and use whether a given nodule is malignant or not as the label to predict. This is structurally very similar to the balancing we used in chapter 12, but instead of sampling from `pos_list` and `neg_list`, we sample from `mal_list` and `ben_list`. Just as we did for the nodule classifier, we want to keep the training data balanced. We put this into the `MalignancyLunaDataset` class, which subclasses the `LunaDataset` but is otherwise very similar.

For convenience, we create a `dataset` command-line argument in training.py and dynamically use the dataset class specified on the command line. We do this by using Python's `getattr` function. For example, if `self.cli_args.dataset` is the string `MalignancyLunaDataset`, it will get `p2ch14.dsets.MalignancyLunaDataset` and assign this type to `ds_cls`, as we can see here.

Listing 14.7 training.py:154, `.initTrainDl`

```
ds_cls = getattr(p2ch14.dsets, self.cli_args.dataset)

train_ds = ds_cls(
  val_stride=10,
  isValSet_bool=False,
  ratio_int=1,
)
```

Dynamic class-name lookup

Recall that this is the one-to-one balancing of the training data, here between benign and malignant.

14.5.2 An area under the curve baseline: Classifying by diameter

It is always good to have a baseline to see what performance is better than nothing. We could go for better than random, but here we can use the diameter as a predictor for malignancy—larger nodules are more likely to be malignant. Step 2b of figure 14.5 hints at a new metric we can use to compare classifiers.

[4] See the PyLIDC documentation for full details: http://mng.bz/Qyv6.

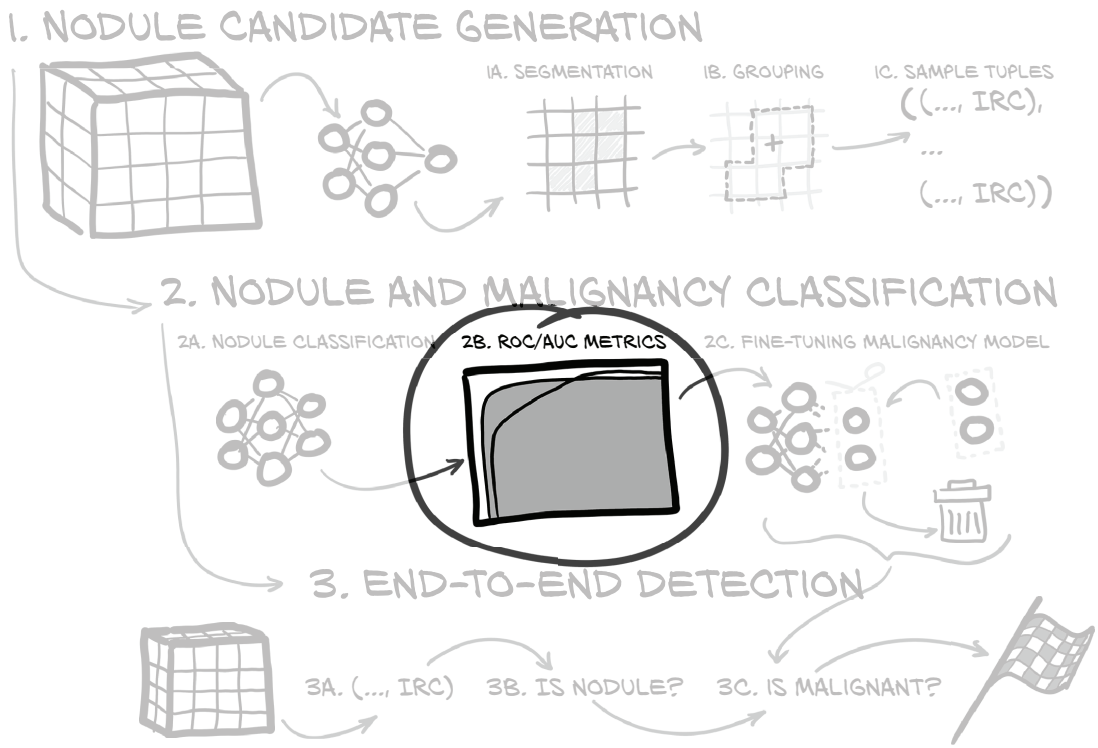


Figure 14.5 The end-to-end project we are implementing in this chapter, with a focus on the ROC graph

We could use the nodule diameter as the sole input to a hypothetical classifier predicting whether a nodule is malignant. It wouldn't be a very good classifier, but it turns out that saying "Everything bigger than this threshold X is malignant" is a better predictor of malignancy than we might expect. Of course, picking the right threshold is key—there's a sweet spot that gets all the huge tumors and none of the tiny specks, and roughly splits the uncertain area that's a jumble of larger benign nodules and smaller malignant ones.

As we might recall from chapter 12, our true positive, false positive, true negative, and false negative counts change based on what threshold value we choose. As we decrease the threshold over which we predict that a nodule is malignant, we will increase the number of true positives, but also the number of false positives. The *false positive rate* (FPR) is FP / (FP + TN), while the *true positive rate* (TPR) is TP / (TP + FN), which you might also remember from chapter 12 as the recall.

Let's set a range for our threshold. The lower bound will be the value at which *all* of our samples are classified as positive, and the upper bound will be the opposite, where all samples are classified as negative. At one extreme, our FPR and TPR will both be zero, since there won't be *any* positives; and at the other, both will be one, since TNs and FNs won't exist (everything is positive!).

No one true way to measure false positives: Precision vs. false positive rate

The FPR here and the precision from chapter 12 are rates (between 0 and 1) that measure things that are not quite opposites. As we discussed, precision is TP / (TP + FP) and measures how many of the samples predicted to be positive will actually be positive. The FPR is FP / (FP + TN) and measures how many of the actually negative samples are predicted to be positive. For heavily imbalanced datasets (like the nodule versus non-nodule classification), our model might achieve a very good FPR (which is closely related to the cross-entropy criterion as a loss) while the precision—and thus the F1 score—is still very poor. A low FPR means we're weeding out a lot of what we're not interested in, but if we are looking for that proverbial needle, we still have mostly hay.

For our nodule data, that's from 3.25 mm (the smallest nodule) to 22.78 mm (the largest). If we pick a threshold value somewhere between those two values, we can then compute FPR(threshold) and TPR(threshold). If we set the FPR value to *X* and TPR to *Y*, we can plot a point that represents that threshold; and if we instead plot the FPR versus TPR for every possible threshold, we get a diagram called the *receiver operating characteristic* (ROC) shown in figure 14.6. The shaded area is the *area under the (ROC) curve*, or AUC. It is between 0 and 1, and higher is better.[5]

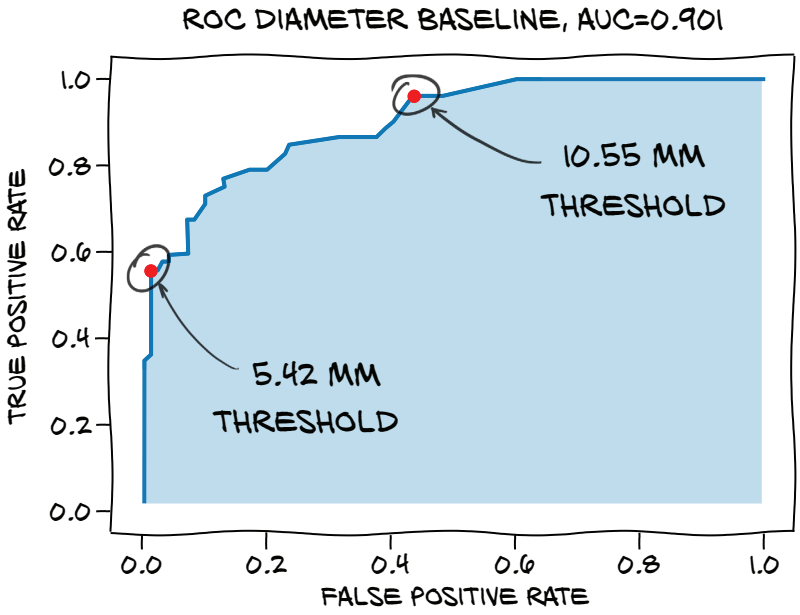


Figure 14.6 Receiver operating characteristic (ROC) curve for our baseline

[5] Note that random predictions on a balanced dataset would result in an AUC of 0.5, so that gives us a floor for how good our classifier must be.

Here, we also call out two specific threshold values: diameters of 5.42 mm and 10.55 mm. We chose those two values because they give us somewhat reasonable endpoints for the range of thresholds we might consider, were we to need to pick a single threshold. Anything smaller than 5.42 mm, and we'd only be dropping our TPR. Larger than 10.55 mm, and we'd just be flagging malignant nodules as benign for no gain. The best threshold for this classifier will probably be in the middle somewhere.

How do we actually compute the values shown here? We first grab the candidate info list, filter out the annotated nodules, and get the malignancy label and diameter. For convenience, we also get the number of benign and malignant nodules.

Listing 14.8 p2ch14_malben_baseline.ipynb

Takes the regular dataset and in particular the list of benign and malignant nodules

```
# In[2]:
ds = p2ch14.dsets.MalignantLunaDataset(val_stride=10, isValSet_bool=True)
nodules = ds.ben_list + ds.mal_list
is_mal = torch.tensor([n.isMal_bool for n in nodules])
diam  = torch.tensor([n.diameter_mm for n in nodules])
num_mal = is_mal.sum()
num_ben = len(is_mal) - num_mal
```

Gets lists of malignancy status and diameter

For normalization of the TPR and FPR , we take the number of malignant and benign nodules.

To compute the ROC curve, we need an array of the possible thresholds. We get this from `torch.linspace`, which takes the two boundary elements. We wish to start at zero predicted positives, so we go from maximal threshold to minimal. This is the 3.25 to 22.78 we already mentioned:

```
# In[3]:
threshold = torch.linspace(diam.max(), diam.min())
```

We then build a two-dimensional tensor in which the rows are per threshold, the columns are per-sample information, and the value is whether this sample is predicted as positive. This Boolean tensor is then filtered by whether the label of the sample is malignant or benign. We sum the rows to count the number of `True` entries. Dividing by the number of malignant or benign nodules gives us the TPR and FPR—the two coordinates for the ROC curve:

Indexing by None adds a dimension of size 1, just like .unsqueeze(ndx). This gets us a 2D tensor of whether a given nodule (in a column) is classified as malignant for a given diameter (in the row).

```
# In[4]:
predictions = (diam[None] >= threshold[:, None])
tp_diam = (predictions & is_mal[None]).sum(1).float() / num_mal
fp_diam = (predictions & ~is_mal[None]).sum(1).float() / num_ben
```

With the predictions matrix, we can compute the TPRs and FPRs for each diameter by summing over the columns.

To compute the area under this curve, we use numeric integration by the trapezoidal rule (https://en.wikipedia.org/wiki/Trapezoidal_rule), where we multiply the average TPRs (on the Y-axis) between two points by the difference of the two FPRs (on the X-axis)—the area of trapezoids between two points of the graph. Then we sum the area of the trapezoids:

```
# In[5]:
fp_diam_diff =  fp_diam[1:] - fp_diam[:-1]
tp_diam_avg  = (tp_diam[1:] + tp_diam[:-1])/2
auc_diam = (fp_diam_diff * tp_diam_avg).sum()
```

Now, if we run `pyplot.plot(fp_diam, tp_diam, label=f"diameter baseline, AUC={auc_diam:.3f}")` (along with the appropriate figure setup we see in cell 8), we get the plot we saw in figure 14.6.

14.5.3 Reusing preexisting weights: Fine-tuning

One way to quickly get results (and often also get by with much less data) is to start not from random initializations but from a network trained on some task with related data. This is called *transfer learning* or, when training only the last few layers, *fine-tuning*. Looking at the highlighted part in figure 14.7, we see that in step 2c, we're going to cut out the last bit of the model and replace it with something new.

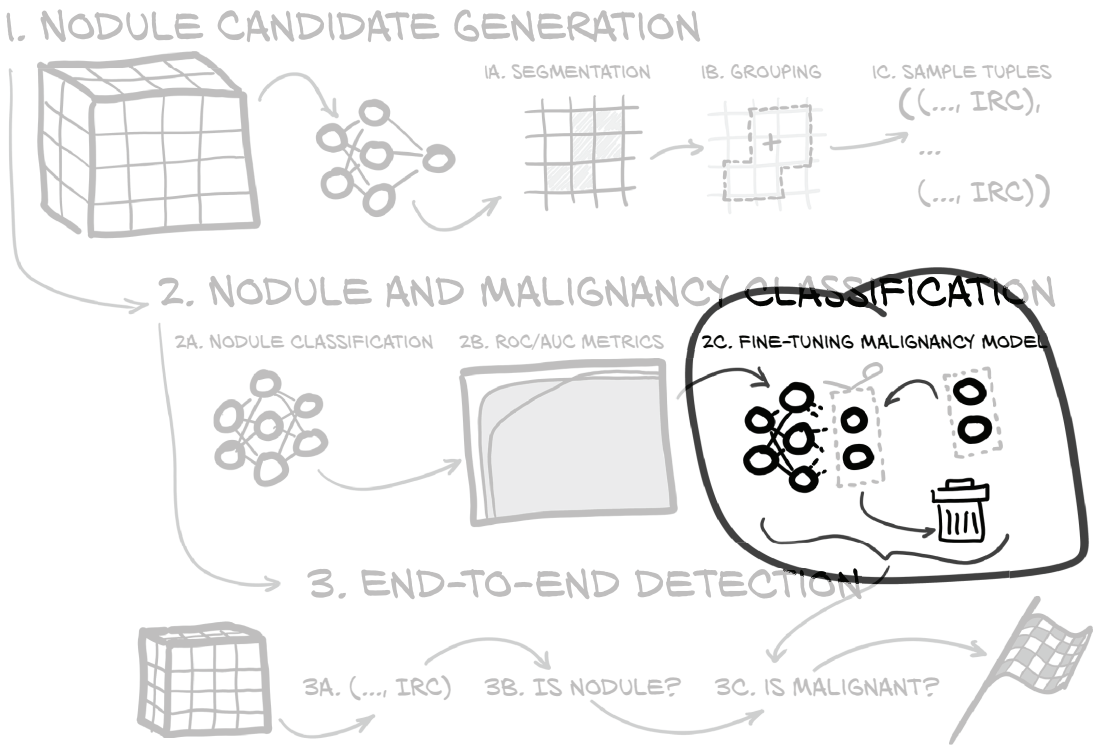


Figure 14.7 The end-to-end project we're implementing in this chapter, with a focus on fine-tuning

Recall from chapter 8 that we could interpret the intermediate values as features extracted from the image—features could be edges or corners that the model detects or indications of any pattern. Before deep learning, it was very common to use handcrafted features similar to what we briefly experimented with when starting with convolutions. Deep learning has the network derive features useful for the task at hand, such as discrimination between classes, from the data. Now, fine-tuning has us mix the ancient ways (almost a *decade* ago!) of using preexisting features and the new way of using learned features. We treat some (often large) part of the network as a fixed *feature extractor* and only train a relatively small part on top of it.

This generally works very well. Pretrained networks trained on ImageNet as we saw in chapter 2 are very useful as feature extractors for many tasks dealing with natural images—sometimes they also work amazingly for completely different inputs, from paintings or imitations thereof in style transfer to audio spectrograms. There are cases when this strategy works less well. For example, one of the common data augmentation strategies in training models on ImageNet is randomly flipping the images—a dog looking right is the same class as one looking left. As a result, the features between flipped images are very similar. But if we now try to use the pretrained model for a task where left or right matters, we will likely encounter accuracy problems. If we want to identify traffic signs, *turn left here* is quite different than *turn right here*; but a network building on ImageNet-based features will probably make lots of wrong assignments between the two classes.[6]

In our case, we have a network that has been trained on similar data: the nodule classification network. Let's try using that.

For the sake of exposition, we will stay very basic in our fine-tuning approach. In the model architecture in figure 14.8, the two bits of particular interest are highlighted: the last convolutional block and the `head_linear` module. The simplest fine-tuning is to cut out the `head_linear` part—in truth, we are just keeping the random initialization. After we try that, we will also explore a variant where we retrain both `head_linear` and the last convolutional block.

We need to do the following:

- Load the weights of the model we wish to start with, except for the last linear layer, where we want to keep the initialization.
- Disable gradients for the parameters we do not want to train (everything except parameters with names starting with `head`).

When we do fine-tuning training on more than `head_linear`, we still only reset `head_linear` to random, because we believe the previous feature-extraction layers might

[6] You can try it yourself with the venerable German Traffic Sign Recognition Benchmark dataset at http://mng.bz/XPZ9.

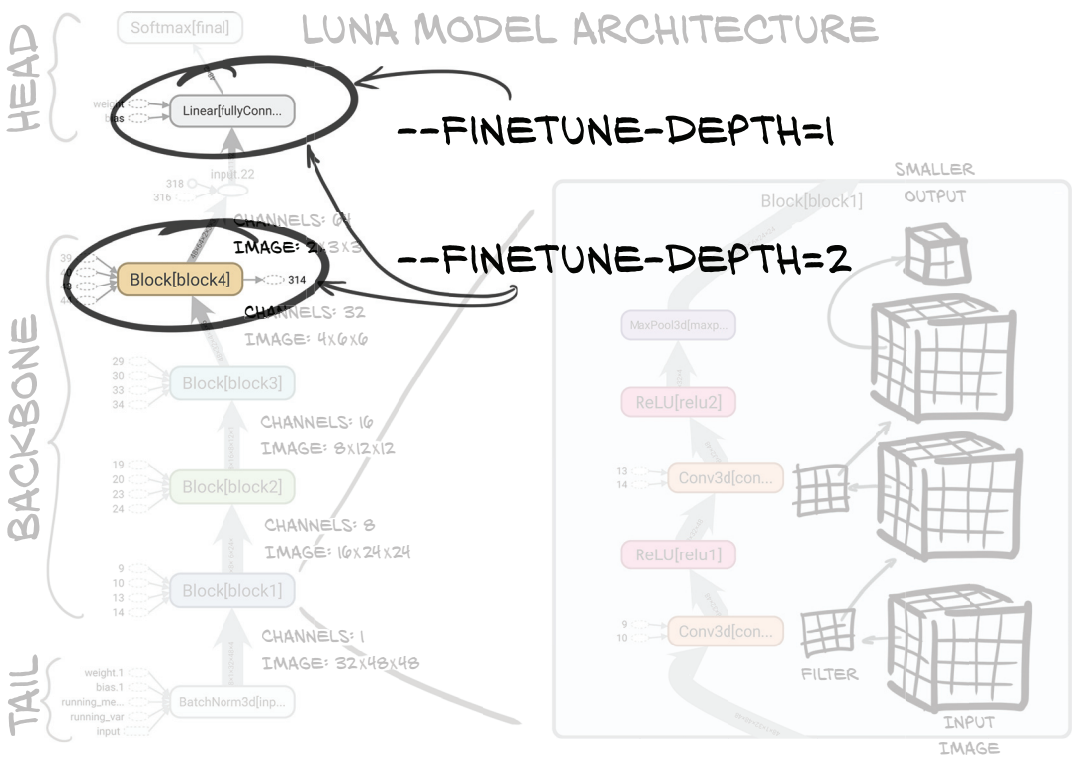


Figure 14.8 The model architecture from chapter 11, with the depth-1 and depth-2 weights highlighted

not be ideal for our problem, but we expect them to be a reasonable starting point. This is easy: we add some loading code into our model setup.

Listing 14.9 training.py:124, `.initModel`

```
d = torch.load(self.cli_args.finetune, map_location='cpu')
model_blocks = [
  n for n, subm in model.named_children()
  if len(list(subm.parameters())) > 0
]
finetune_blocks = model_blocks[-self.cli_args.finetune_depth:]
model.load_state_dict(
  {
    k: v for k,v in d['model_state'].items()
    if k.split('.')[0] not in model_blocks[-1]
  },
```

Filters out top-level modules that have parameters (as opposed to the final activation)

Takes the last finetune_depth blocks. The default (if fine-tuning) is 1.

Filters out the last block (the final linear part) and does not load it. Starting from a fully initialized model would have us begin with (almost) all nodules labeled as malignant, because that output means "nodule" in the classifier we start from.

```
  strict=False,
)
for n, p in model.named_parameters():
  if n.split('.')[0] not in finetune_blocks:
    p.requires_grad_(False)
```

Passing strict=False lets us load only some weights of the module (with the filtered ones missing).

For all but finetune_blocks, we do not want gradients.

We're set! We can train only the head by running this:

```
python3 -m p2ch14.training \
    --malignant \
    --dataset MalignantLunaDataset \
    --finetune data/part2/models/cls_2020-02-06_14.16.55_final-nodule-
➥ nonnodule.best.state \
    --epochs 40 \
    malben-finetune
```

Let's run our model on the validation set and get the ROC curve, shown in figure 14.9. It's a lot better than random, but given that we're not outperforming the baseline, we need to see what is holding us back.

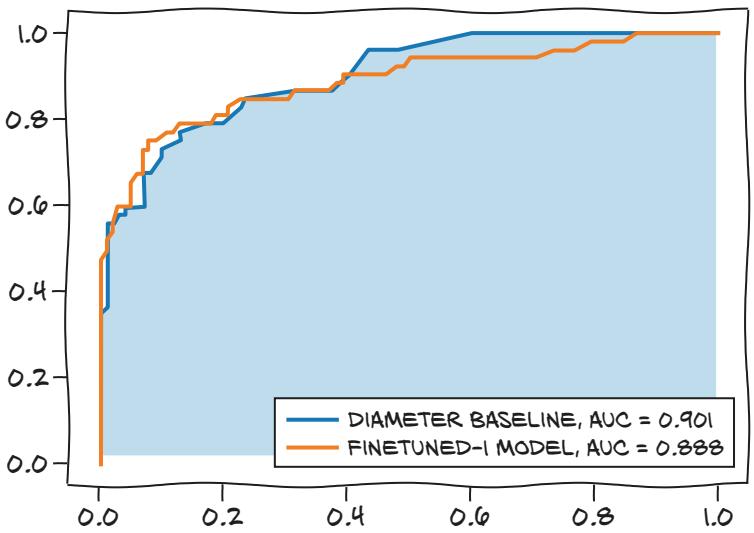


Figure 14.9 ROC curve for our fine-tuned model with a retrained final linear layer. Not too bad, but not quite as good as the baseline.

Figure 14.10 shows the TensorBoard graphs for our training. Looking at the validation loss, we see that while the AUC slowly increases and the loss decreases, even the training loss seems to plateau at a somewhat high level (say, 0.3) instead of trending toward zero. We could run a longer training to check whether it is just very slow; but comparing this to the loss progression discussed in chapter 5—in particular, figure 5.14—we can see our loss value has not flatlined as badly as case A in the figure, but our problem with

losses stagnating is qualitatively similar. Back then, case A indicated that we did not have enough capacity, so we should consider the following three possible causes:

- Features (the output of the last convolution) obtained by training the network on nodule versus non-nodule classification are not useful for malignancy detection.
- The capacity of the head—the only part we are training—is not large enough.
- The network might have too little capacity overall.

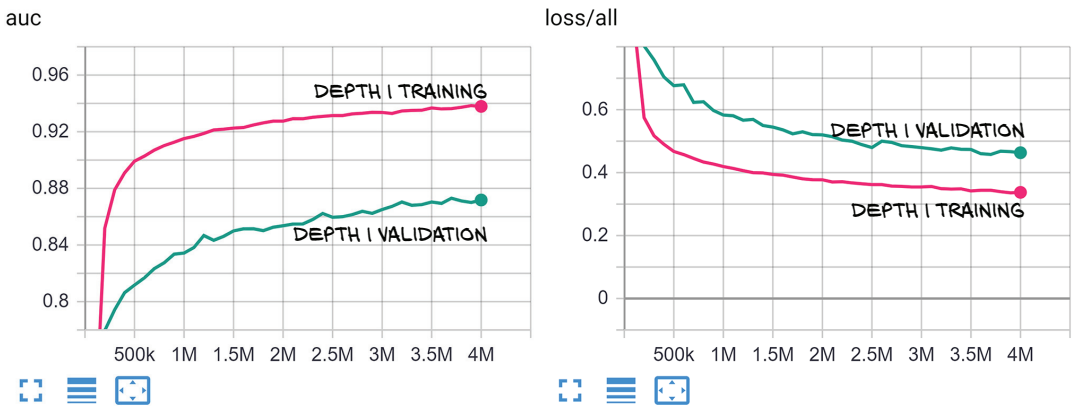


Figure 14.10 AUC (left) and loss (right) for the fine-tuning of the last linear layer

If training only the fully connected part in fine-tuning is not enough, the next thing to try is to include the last convolutional block in the fine-tuning training. Happily, we introduced a parameter for that, so we can include the `block4` part into our training:

```
python3 -m p2ch14.training \
    --malignant \
    --dataset MalignantLunaDataset \
    --finetune data/part2/models/cls_2020-02-06_14.16.55_final-nodule-
➥ nonnodule.best.state \
    --finetune-depth 2 \
    --epochs 10 \
    malben-finetune-twolayer
```

This CLI parameter is new.

Once done, we can check our new best model against the baseline. Figure 14.11 looks more reasonable! We flag about 75% of the malignant nodules with almost no false positives. This is clearly better than the 65% the diameter baseline can give us. Trying to push beyond 75%, our model's performance falls back to the baseline. When we go back to the classification problem, we will want to pick a point on the ROC curve to balance true positives versus false positives.

We are roughly on par with the baseline, and we will be content with that. In section 14.7, we hint at the many things that you can explore to improve these results, but that didn't fit in this book.

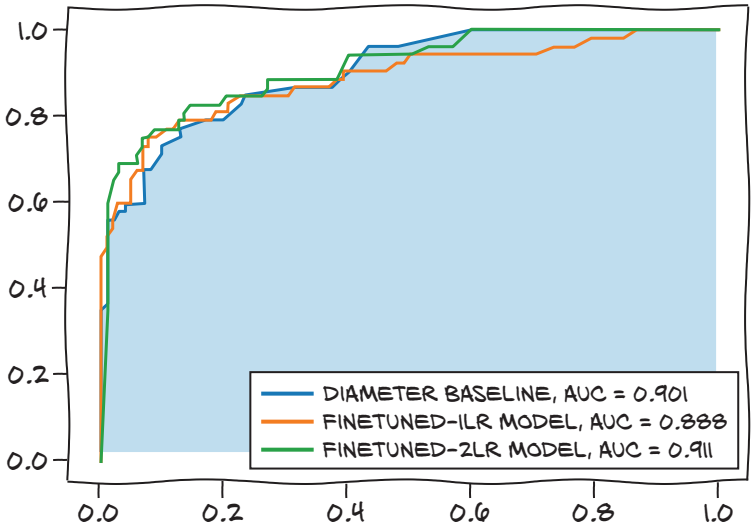


Figure 14.11 ROC curve for our modified model. Now we're getting really close to the baseline.

Looking at the loss curves in figure 14.12, we see that our model is now overfitting very early; thus the next step would be to check into further regularization methods. We will leave that for you.

There are more refined methods of fine-tuning. Some advocate gradually unfreeze the layers, starting from the top. Others propose to train the later layers with the usual learning rate and use a smaller rate for the lower layers. PyTorch does natively support using different optimization parameters like learning rates, weight decay, and momentum for different parameters by separating them in several *parameter groups* that are just that: lists of parameters with separate hyperparameters (https://pytorch.org/docs/stable/optim.html#per-parameter-options).

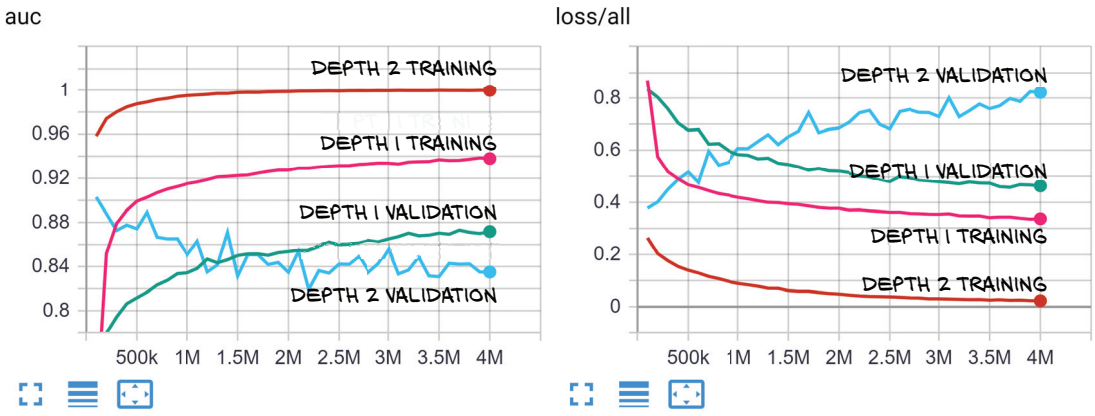


Figure 14.12 AUC (left) and loss (right) for the fine-tuning of the last convolutional block and the fully connected layer

14.5.4 More output in TensorBoard

While we are retraining the model, it might be worth looking at a few more outputs we could add to TensorBoard to see how we are doing. For histograms, TensorBoard has a premade recording function. For ROC curves, it does not, so we have an opportunity to meet the Matplotlib interface.

HISTOGRAMS

We can take the predicted probabilities for malignancy and make a histogram of them. Actually, we make two: one for (according to the ground truth) benign and one for malignant nodules. These histograms give us a fine-grained view into the outputs of the model and let us see if there are large clusters of output probabilities that are completely wrong.

> **NOTE** In general, shaping the data you display is an important part of getting quality information from the data. If you have many extremely confident correct classifications, you might want to exclude the leftmost bin. Getting the right things onscreen will typically require some iteration of careful thought and experimentation. Don't hesitate to tweak what you're showing, but also take care to remember if you change the definition of a particular metric without changing the name. It can be easy to compare apples to oranges unless you're disciplined about naming schemes or removing now-invalid runs of data.

We first create some space in the tensor `metrics_t` holding our data. Recall that we defined the indices somewhere near the top.

Listing 14.10 training.py:31

```
METRICS_LABEL_NDX=0
METRICS_PRED_NDX=1
METRICS_PRED_P_NDX=2
METRICS_LOSS_NDX=3
METRICS_SIZE = 4
```

Our new index, carrying the prediction probabilities (rather than prethresholded predictions)

Once that's done, we can call `writer.add_histogram` with a label, the data, and the `global_step` counter set to our number of training samples presented; this is similar to the scalar call earlier. We also pass in `bins` set to a fixed scale.

Listing 14.11 training.py:496, `.logMetrics`

```
bins = np.linspace(0, 1)

writer.add_histogram(
  'label_neg',
  metrics_t[METRICS_PRED_P_NDX, negLabel_mask],
  self.totalTrainingSamples_count,
  bins=bins
)
writer.add_histogram(
  'label_pos',
```

```
  metrics_t[METRICS_PRED_P_NDX, posLabel_mask],
  self.totalTrainingSamples_count,
  bins=bins
)
```

Now we can take a look at our prediction distribution for benign samples and how it evolves over each epoch. We want to examine two main features of the histograms in figure 14.13. As we would expect if our network is learning anything, in the top row of benign samples and non-nodules, there is a mountain on the left where the network is very confident that what it sees is not malignant. Similarly, there is a mountain on the right for the malignant samples.

But looking closer, we see the capacity problem of fine-tuning only one layer. Focusing on the top-left series of histograms, we see the mass to the left is somewhat spread out and does not seem to reduce much. There is even a small peak round 1.0, and quite a bit of probability mass is spread out across the entire range. This reflects the loss that didn't want to decrease below 0.3.

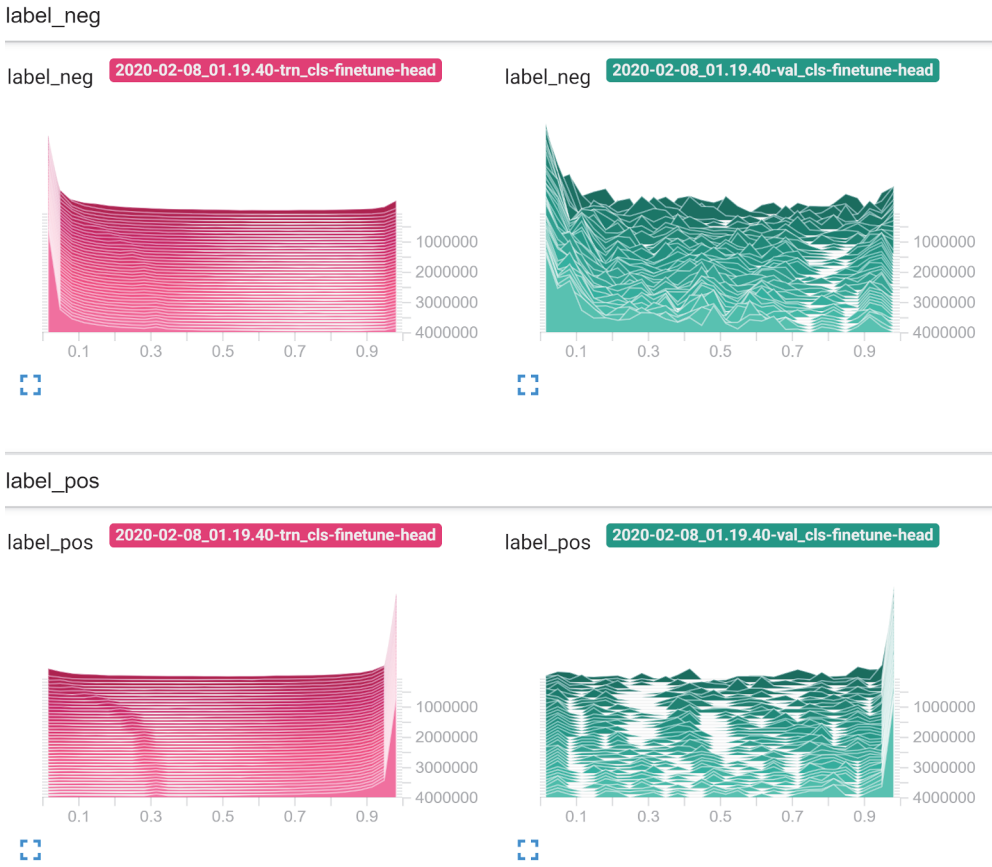


Figure 14.13 TensorBoard histogram display for fine-tuning the head only

Given this observation on the training loss, we would not have to look further, but let's pretend for a moment that we do. In the validation results on the right side, it appears that the probability mass away from the "correct" side is larger for the non-malignant samples in the top-right diagram than for the malignant ones in the bottom-right diagram. So the network gets non-malignant samples wrong more often than malignant ones. This might have us look into rebalancing the data to show more non-malignant samples. But again, this is when we pretend there was nothing wrong with the training on the left side. We typically want to fix training first!

For comparison, let's take a look at the same graph for our depth 2 fine-tuning (figure 14.14). On the training side (the left two diagrams), we have very sharp peaks at the correct answer and not much else. This reflects that training works well.

On the validation side, we now see that the most pronounced artifact is the little peak at 0 predicted probability for malignancy in the bottom-right histogram. So our systematic problem is that we're misclassifying malignant samples as non-malignant. (This is the reverse of what we had earlier!) This is the overfitting we saw with two-layer

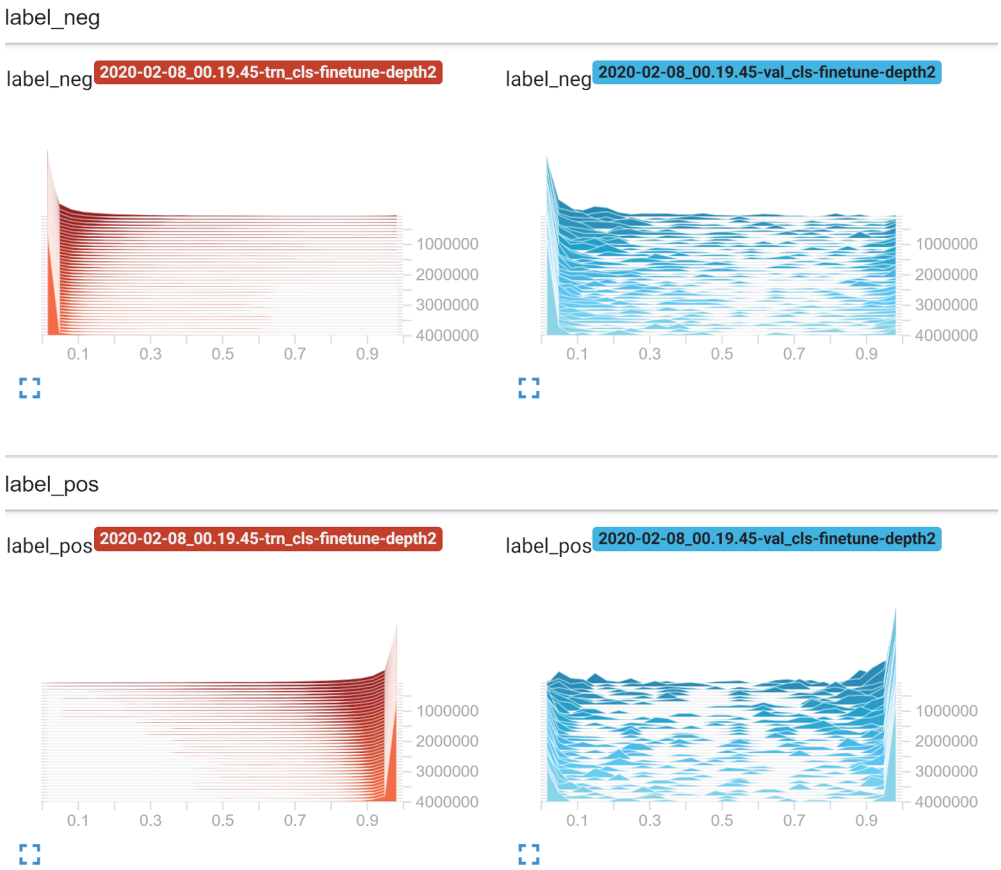


Figure 14.14 TensorBoard histogram display for fine-tuning with depth 2

fine-tuning. It probably would be good to pull up a few images of that type to see what's happening.

ROC AND OTHER CURVES IN TENSORBOARD

As mentioned earlier, TensorBoard does not natively support drawing ROC curves. We can, however, use the ability to export any graph from Matplotlib. The data preparation looks just like in section 14.5.2: we use the data that we also plotted in the histogram to compute the TPR and FPR—`tpr` and `fpr`, respectively. We again plot our data, but this time we keep track of `pyplot.figure` and pass it to the `SummaryWriter` method `add_figure`.

Listing 14.12 training.py:482, `.logMetrics`

Sets up a new Matplotlib figure. We usually don't need it because it is implicitly done in Matplotlib, but here we do.

Uses arbitrary pyplot functions

Adds our figure to TensorBoard

```
fig = pyplot.figure()
pyplot.plot(fpr, tpr)
writer.add_figure('roc', fig, self.totalTrainingSamples_count)
```

Because this is given to TensorBoard as an image, it appears under that heading. We didn't draw the comparison curve or anything else, so as not to distract you from the actual function call, but we could use any Matplotlib facilities here. In figure 14.15, we see again that the depth-2 fine-tuning (left) overfits, while the head-only fine-tuning (right) does not.

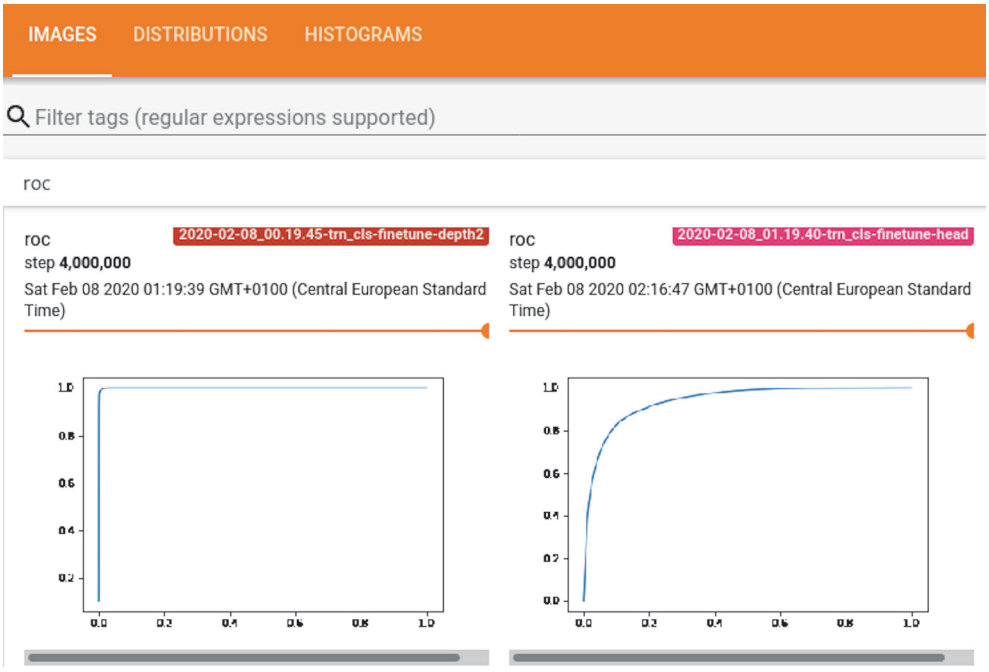


Figure 14.15 Training ROC curves in TensorBoard. A slider lets us go through the iterations.

14.6 What we see when we diagnose

Following along with steps 3a, 3b, and 3c in figure 14.16, we now need to run the full pipeline from the step 3a segmentation on the left to the step 3c malignancy model on the right. The good news is that almost all of our code is in place already! We just need to stitch it together: the moment has come to actually write and run our end-to-end diagnosis script.

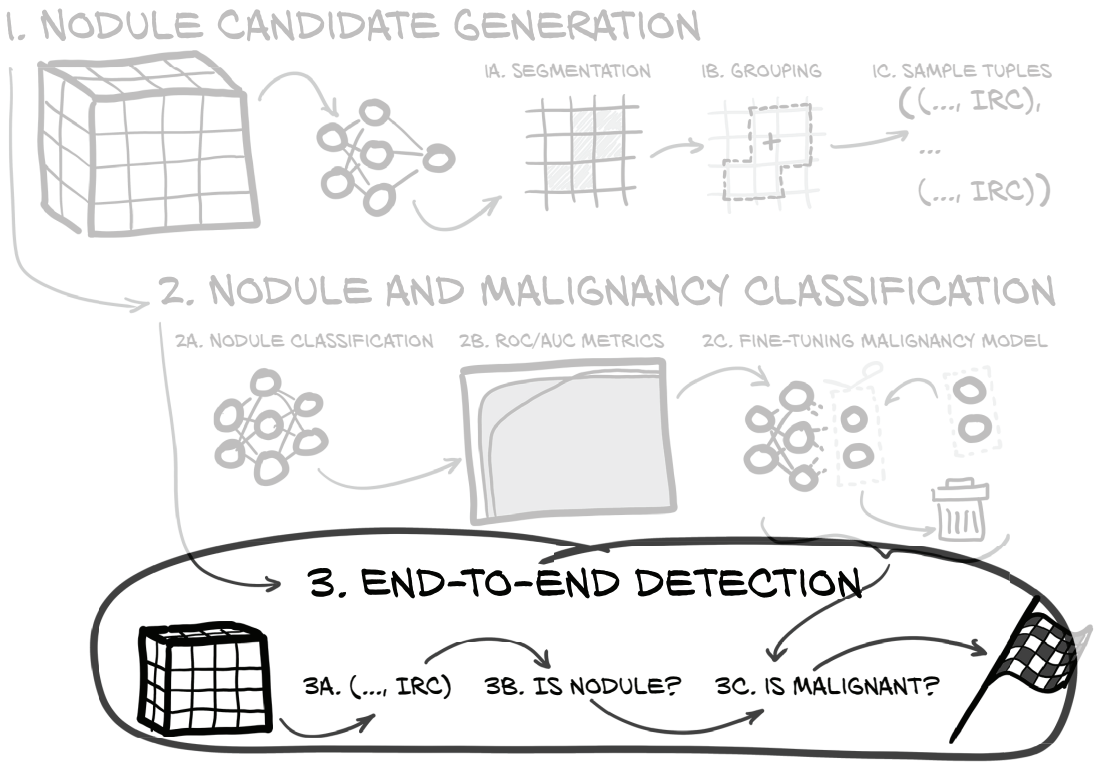


Figure 14.16 The end-to-end project we are implementing in this chapter, with a focus on end-to-end detection

We saw our first hints at handling the malignancy model back in the code in section 14.3.3. If we pass an argument `--malignancy-path` to the `nodule_analysis` call, it runs the malignancy model found at this path and outputs the information. This works for both a single scan and the `--run-validation` variant.

Be warned that the script will probably take a while to finish; even just the 89 CTs in the validation set took about 25 minutes.[7]

[7] Most of the delay is from SciPy's processing of the connected components. At the time of writing, we are not aware of an accelerated implementation.

Let's see what we get:

```
Total
              | Complete Miss | Filtered Out | Pred. Benign | Pred. Malignant
 Non-Nodules |               |       164893 |         1593 |             563
      Benign |            12 |            3 |           70 |              17
   Malignant |             1 |            6 |            9 |              36
```

Not too bad! We detect about 85% of the nodules and correctly flag about 70% of the malignant ones, end to end.[8] While we have a lot of false positives, it would seem that having 16 of them per true nodule reduces what needs to be looked at (well, if it were not for the 30% false negatives). As we already warned in chapter 9, this isn't at the level where you could collect millions of funding for your medical AI startup,[9] but it's a pretty reasonable starting point. In general, we should be pretty happy that we're getting results that are clearly meaningful; and of course our *real* goal has been to study deep learning along the way.

We might next choose to look at the nodules that are actually misclassified. Keep in mind that for our task at hand, even the radiologists who annotated the dataset differed in opinion. We might stratify our validation set by how clearly they identified a nodule as malignant.

14.6.1 Training, validation, and test sets

There is one caveat that we must mention. While we didn't explicitly train our model on the validation set, although we ran this risk at the beginning of the chapter, we did *choose* the epoch of training to use based on the model's performance on the validation set. That's a bit of a data leak, too. In fact, we should expect our real-world performance to be slightly worse than this, as it's unlikely that whatever model performs best on our validation set will perform equally well on every other unseen set of data (on average, at least).

For this reason, practitioners often split data into *three* sets:

- A *training set*, exactly as we've done here
- A *validation set*, used to determine which epoch of evolution of the model to consider "best"
- A *test set*, used to actually predict performance for the model (as chosen by the validation set) on unseen, real-world data

Adding a third set would have led us to pull another nontrivial chunk of our training data, which would have been somewhat painful, given how badly we had to fight overfitting already. It would also have complicated the presentation, so we purposely left it out. Were this a project with the resources to get more data and an imperative to build the best possible system to use in the wild, we'd have to make a different decision here and actively seek more data to use as an independent test set.

---

[8] Recall that our earlier "75% with almost no false positives" ROC number was looking at malignancy classification in isolation. Here we are filtering out seven malignant nodules before we even get to the malignancy classifier.

[9] If it were, we'd have done that instead of writing this book!

The general message is that there are subtle ways for bias to creep into our models. We should use extra care to control information leakage at every step of the way and verify its absence using independent data as much as possible. The price to pay for taking shortcuts is failing egregiously at a later stage, at the worst possible time: when we're closer to production.

14.7 What next? Additional sources of inspiration (and data)

Further improvements will be difficult to measure at this point. Our classification validation set contains 154 nodules, and our nodule classification model is typically getting at least 150 of them right, with most of the variance coming from epoch-by-epoch training changes. Even if we were to make a significant improvement to our model, we don't have enough fidelity in our validation set to tell whether that change is an improvement for certain! This is also very pronounced in the benign versus malignant classification, where the validation loss zigzags a lot. If we reduced our validation stride from 10 to 5, the size of our validation set would double, at the cost of one-ninth of our training data. That might be worth it if we wanted to try other improvements. Of course, we would also need to address the question of a test set, which would take away from our already limited training data.

We would also want to take a good look at the cases where the network does not perform as well as we'd like, to see if we can identify any pattern. But beyond that, let's talk briefly about some general ways we could improve our project. In a way, this section is like section 8.5 in chapter 8. We will endeavor to fill you with ideas to try; don't worry if you don't understand each in detail.[10]

14.7.1 Preventing overfitting: Better regularization

Reflecting on what we did throughout part 2, in each of the three problems—the classifiers in chapter 11 and section 14.5, as well as the segmentation in chapter 13—we had overfitting models. Overfitting in the first case was catastrophic; we dealt with it by balancing the data and augmentation in chapter 12. This balancing of the data to prevent overfitting has also been the main motivation to train the U-Net on crops around nodules and candidates rather than full slices. For the remaining overfitting, we bailed out, stopping training early when the overfitting started to affect our validation results. This means preventing or reducing overfitting would be a great way to improve our results.

This pattern—get a model that overfits, and then work to reduce that overfitting—can really can be seen as a recipe.[11] So this two-step process should be used when we want to improve on the state we have achieved now.

---

[10] At least one of the authors would love to write an entire book on the topics touched on in this section.

[11] See also Andrej Karparthy's blog post "A Recipe for Training Neural Networks" at https://karpathy.github.io/2019/04/25/recipe for a more elaborate recipe.

Classic regularization and augmentation

You might have noticed that we did not even use all the regularization techniques from chapter 8. For example, dropout would be an easy thing to try.

While we have some augmentation in place, we could go further. One relatively powerful augmentation method we did not attempt to employ is elastic deformations, where we put "digital crumples" into the inputs.[12] This makes for much more variability than rotation and flipping alone and would seem to be applicable to our tasks as well.

More abstract augmentation

So far, our augmentation has been geometrically inspired—we transformed our input to more or less look like something plausible we might see. It turns out that we need not limit ourselves to that type of augmentation.

Recall from chapter 8 that mathematically, the cross-entropy loss we have been using is a measure of the discrepancy between two probability distributions—that of the predictions and the distribution that puts all probability mass on the label and can be represented by the one-hot vector for the label. If overconfidence is a problem for our network, one simple thing we could try is not using the one-hot distribution but rather putting a small probability mass on the "wrong" classes.[13] This is called *label smoothing*.

We can also mess with inputs and labels at the same time. A very general and also easy-to-apply augmentation technique for doing this has been proposed under the name of *mixup*:[14] the authors propose to randomly interpolate both inputs and labels. Interestingly, with a linearity assumption for the loss (which is satisfied by binary cross entropy), this is equivalent to just manipulating the inputs with a weight drawn from an appropriately adapted distribution.[15] Clearly, we don't expect blended inputs to occur when working on real data, but it seems that this mixing encourages stability of the predictions and is very effective.

Beyond a single best model: Ensembling

One perspective we could have on the problem of overfitting is that our model is capable of working the way we want if we knew the right parameters, but we don't actually know them.[16] If we followed this intuition, we might try to come up with several sets of parameters (that is, several models), hoping that the weaknesses of each might compensate for the other. This technique of evaluating several models and combining the output is called *ensembling*. Simply put, we train several models and then, in order to predict, run all of them and average the predictions. When each individual model overfits (or we have taken a snapshot of the model just before we started to see the overfitting), it seems plausible that the models might start to make bad predictions on different inputs, rather than always overfit the same sample first.

---

[12] You can find a recipe (albeit aimed at TensorFlow) at http://mng.bz/Md5Q.
[13] You can use `nn.KLDivLoss` loss for this.
[14] Hongyi Zhang et al., "mixup: Beyond Empirical Risk Minimization," https://arxiv.org/abs/1710.09412.
[15] See Ferenc Huszár's post at http://mng.bz/aRJj/; he also provides PyTorch code.
[16] We might expand that to be outright Bayesian, but we'll just go with this bit of intuition.

In ensembling, we typically use completely separate training runs or even varying model structures. But if we were to make it particularly simple, we could take several snapshots of the model from a single training run—preferably shortly before the end or before we start to observe overfitting. We might try to build an ensemble of these snapshots, but as they will still be somewhat close to each other, we could instead average them. This is the core idea of *stochastic weight averaging*.[17] We need to exercise some care when doing so: for example, when our models use batch normalization, we might want to adjust the statistics, but we can likely get a small accuracy boost even without that.

GENERALIZING WHAT WE ASK THE NETWORK TO LEARN

We could also look at *multitask learning*, where we require a model to learn additional outputs beyond the ones we will then evaluate,[18] which has a proven track record of improving results. We could try to train on nodule versus non-nodule and benign versus malignant at the same time. Actually, the data source for the malignancy data provides additional labeling we could use as additional tasks; see the next section. This idea is closely related to the transfer-learning concept we looked at earlier, but here we would typically train both tasks in parallel rather than first doing one and then trying to move to the next.

If we do not have additional tasks but rather have a stash of additional unlabeled data, we can look into *semi-supervised learning*. An approach that was recently proposed and looks very effective is unsupervised data augmentation.[19] Here we train our model as usual on the data. On the unlabeled data, we make a prediction on an unaugmented sample. We then take that prediction as the target for this sample and train the model to predict that target on the augmented sample as well. In other words, we don't know if the prediction is correct, but we ask the network to produce consistent outputs whether we augment or not.

When we run out of tasks of genuine interest but do not have additional data, we may look at making things up. Making up data is somewhat difficult (although people sometimes use GANs similar to the ones we briefly saw in chapter 2, with some success), so we instead make up tasks. This is when we enter the realm of *self-supervised learning*; the tasks are often called *pretext tasks*. A very popular crop of pretext tasks apply some sort of corruption to some of the inputs. Then we can train a network to reconstruct the original (for example, using a U-Net-like architecture) or train a classifier to detect real from corrupted data while sharing large parts of the model (such as the convolutional layers).

This is still dependent on us coming up with a way to corrupt our inputs. If we don't have such a method in mind and aren't getting the results we want, there are

---

[17] Pavel Izmailov and Andrew Gordon Wilson present an introduction with PyTorch code at http://mng.bz/gywe.

[18] See Sebastian Ruder, "An Overview of Multi-Task Learning in Deep Neural Networks," https://arxiv.org/abs/1706.05098; but this is also a key idea in many areas.

[19] Q. Xie et al., "Unsupervised Data Augmentation for Consistency Training," https://arxiv.org/abs/1904.12848.

other ways to do self-supervised learning. A very generic task would be if the features the model learns are good enough to let the model discriminate between different samples of our dataset. This is called *contrastive learning*.

To make things more concrete, consider the following: we take the extracted features from the current image and a largish number K of other images. This is our *key* set of features. Now we set up a classification pretext task as follows: given the features of the current image, the *query*, to which of the $K + 1$ *key* features does it belong? This might seem trivial at first, but even if there is perfect agreement between the query features and the key features for the correct class, training on this task encourages the feature of the query to be maximally dissimilar from those of the K other images (in terms of being assigned low probability in the classifier output). Of course, there are many details to fill in; we recommend (somewhat arbitrarily) looking at momentum contrast.[20]

14.7.2 Refined training data

We could improve our training data in a few ways. We mentioned earlier that the malignancy classification is actually based on a more nuanced categorization by several radiologists. An easy way to use the data we discarded by making it into the dichotomy "malignant or not?" would be to use the five classes. The radiologists' assessments could then be used as a smoothed label: we could one-hot-encode each one and then average over the assessments of a given nodule. So if four radiologists look at a nodule and two call it "indeterminate," one calls that same nodule "moderately suspicious," and the fourth labels it "highly suspicious," we would train on the cross entropy between the model output and the target probability distribution given by the vector `0 0 0.5 0.25 0.25`. This would be similar to the label smoothing we mentioned earlier, but in a smarter, problem-specific way. We would, however, have to find a new way of evaluating these models, as we lose the simple accuracy, ROC, and AUC notions we have in binary classification.

Another way to use multiple assessments would be to train a number of models instead of one, each trained on the annotations given by an individual radiologist. At inference we would then ensemble the models by, for example, averaging their output probabilities.

In the direction of multiple tasks mentioned earlier, we could again go back to the PyLIDC-provided annotation data, where other classifications are provided for each annotation (subtlety, internal structure, calcification, sphericity, margin definedness, lobulation, spiculation, and texture (https://pylidc.github.io/annotation.html). We might have to learn a lot more about nodules, first, though.

In the segmentation, we could try to see whether the masks provided by PyLIDC work better than those we generated ourselves. Since the LIDC data has annotations from multiple radiologists, it would be possible to group nodules into "high agreement" and "low agreement" groups. It might be interesting to see if that corresponds

[20] K. He et al., "Momentum Contrast for Unsupervised Visual Representation Learning," https://arxiv.org/abs/1911.05722.

to "easy" and "hard" to classify nodules in terms of seeing whether our classifier gets almost all easy ones right and only has trouble on the ones that were more ambiguous to the human experts. Or we could approach the problem from the other side, by defining how difficult nodules are to detect in terms of our model performance: "easy" (correctly classified after an epoch or two of training), "medium" (eventually gotten right), and "hard" (persistently misclassified) buckets.

Beyond readily available data, one thing that would probably make sense is to further partition the nodules by malignancy type. Getting a professional to examine our training data in more detail and flag each nodule with a cancer type, and then forcing the model to report that type, could result in more efficient training. The cost to contract out that work is prohibitive for hobby projects, but paying might make sense in commercial contexts.

Especially difficult cases could also be subject to a limited repeat review by human experts to check for errors. Again, that would require a budget but is certainly within reason for serious endeavors.

14.7.3 Competition results and research papers

Our goal in part 2 was to present a self-contained path from problem to solution, and we did that. But the particular problem of finding and classifying lung nodules has been worked on before; so if you want to dig deeper, you can also see what other people have done.

DATA SCIENCE BOWL 2017

While we have limited the scope of part 2 to the CT scans in the LUNA dataset, there is also a wealth of information available from Data Science Bowl 2017 (www.kaggle.com/c/data-science-bowl-2017), hosted by Kaggle (www.kaggle.com). The data itself is no longer available, but there are many accounts of people describing what worked for them and what did not. For example, some of the Data Science Bowl (DSB) finalists reported that the detailed malignancy level (1…5) information from LIDC was useful during training.

Two highlights you could look at are these:[21]

- Second-place solution write-up by Daniel Hammack and Julian de Wit: http://mng.bz/Md48
- Ninth-place solution write-up by Team Deep Breath: http://mng.bz/aRAX

NOTE Many of the newer techniques we hinted at previously were not yet available to the DSB participants. The three years between the 2017 DSB and this book going to print are an eternity in deep learning!

One idea for a more legitimate test set would be to use the DSB dataset instead of reusing our validation set. Unfortunately, the DSB stopped sharing the raw data, so unless you happen to have access to an old copy, you would need another data source.

[21] Thanks to the Internet Archive for saving them from redesigns.

LUNA PAPERS

The LUNA Grand Challenge has collected several results (https://luna16.grand-challenge.org/Results) that show quite a bit of promise. While not all of the papers provided include enough detail to reproduce the results, many do contain enough information to improve our project. You could review some of the papers and attempt to replicate approaches that seem interesting.

14.8 Conclusion

This chapter concludes part 2 and delivers on the promise we made back in chapter 9: we now have a working, end-to-end system that attempts to diagnose lung cancer from CT scans. Looking back at where we started, we've come a long way and, hopefully, learned a lot. We trained a model to do something interesting and difficult using publicly available data. The key question is, "Will this be good for anything in the real world?" with the follow-up question, "Is this ready for production?" The definition of *production* critically depends on the *intended use*, so if we're wondering whether our algorithm can replace an expert radiologist, this is definitely not the case. We'd argue that this can represent version 0.1 of a tool that could in the future support a radiologist during clinical routine: for instance, by providing a second opinion about something that could have gone unnoticed.

Such a tool would require clearance by regulatory bodies of competence (like the Food and Drug Administration in the United States) in order for it to be employed outside of research contexts. Something we would certainly be missing is an extensive, curated dataset to further train and, even more importantly, validate our work. Individual cases would need to be evaluated by multiple experts in the context of a research protocol; and a proper representation of a wide spectrum of situations, from common presentations to corner cases, would be mandatory.

All these cases, from pure research use to clinical validation to clinical use, would require us to execute our model in an environment amenable to be scaled up. Needless to say, this comes with its own set of challenges, both technical and in terms of process. We'll discuss some of the technical challenges in chapter 15.

14.8.1 Behind the curtain

As we close out the modeling in part 2, we want to pull back the curtain a bit and give you a glimpse at the unvarnished truth of working on deep learning projects. Fundamentally, this book has presented a skewed take on things: a curated set of obstacles and opportunities; a well-tended garden path through the larger wilds of deep learning. We think this semi-organic series of challenges (especially in part 2) makes for a better book, and we hope a better learning experience. It does not, however, make for a more *realistic* experience.

In all likelihood, the vast majority of your experiments will not work out. Not every idea will be a discovery, and not every change will be a breakthrough. Deep learning is fiddly. Deep learning is fickle. And remember that deep learning is literally pushing at the forefront of human knowledge; it's a frontier that we are exploring and mapping

further every day, *right now*. It's an exciting time to be in the field, but as with most fieldwork, you're going to get some mud on your boots.

In the spirit of transparency, here are some things that we tried, that we tripped over, that didn't work, or that at least didn't work well enough to bother keeping:

- Using `HardTanh` instead of `Softmax` for the classification network (it was simpler to explain, but it didn't actually work well).
- Trying to fix the issues caused by `HardTanh` by making the classification network more complicated (skip connections, and so on).
- Poor weight initialization causing training to be unstable, particularly for segmentation.
- Training on full CT slices for segmentation.
- Loss weighting for segmentation with SGD. It didn't work, and Adam was needed for it to be useful.
- True 3D segmentation of CT scans. It didn't work for us, but then DeepMind went and did it anyway.[22] This was before we moved to cropping to nodules, and we ran out of memory, so you might try again based on the current setup.
- Misunderstanding the meaning of the `class` column from the LUNA data, which caused some rewrites partway through authoring the book.
- Accidentally leaving in an "I want results quickly" hack that threw away 80% of the candidate nodules found by the segmentation module, causing the results to look atrocious until we figured out what was going on (that cost an entire weekend!).
- A host of different optimizers, loss functions, and model architectures.
- Balancing the training data in various ways.

There are certainly more that we've forgotten. A lot of things went wrong before they went right! Please learn from our mistakes.

We might also add that for many things in this text, we just picked an approach; we emphatically *do not* imply that other approaches are inferior (many of them are probably better!). Additionally, coding style and project design typically differ a lot between people. In machine learning, it is very common for people to do a lot of programming in Jupyter Notebooks. Notebooks are a great tool to try things quickly, but they come with their own caveats: for example, around how to keep track of what you did. Finally, instead of using the caching mechanism we used with `prepcache`, we could have had a separate preprocessing step that wrote out the data as serialized tensors. Each of these approaches seems to be a matter of taste; even among the three authors, any one of us would do things slightly differently.[23] It is always good to try things and find which one works best for you while remaining flexible when cooperating with your peers.

---

[22] Stanislav Nikolov et al., "Deep Learning to Achieve Clinically Applicable Segmentation of Head and Neck Anatomy for Radiotherapy," https://arxiv.org/pdf/1809.04430.pdf

[23] Oh, the discussions we've had!

14.9 Exercises

1. Implement a test set for classification, or reuse the test set from chapter 13's exercises. Use the validation set to pick the best epochs while training, but use the test set to evaluate the end-to-end project. How well does performance on the validation set line up with performance on the test set?
2. Can you train a single model that is able to do three-way classification, distinguishing among non-nodules, benign modules, and malignant nodules in one pass?
   a. What class-balancing split works best for training?
   b. How does this single-pass model perform, compared to the two-pass approach we are using in the book?
3. We trained our classifier on annotations, but expect it to perform on the output of our segmentation. Use the segmentation model to build a list of non-nodules to use during training instead of the non-nodules provided.
   a. Does the classification model performance improve when trained on this new set?
   b. Can you characterize what kinds of nodule candidates see the biggest changes with the newly trained model?
4. The padded convolutions we use result in less than full context near the edges of the image. Compute the loss for segmented pixels near the edges of the CT scan slice, versus those in the interior. Is there a measurable difference between the two?
5. Try running the classifier on the entire CT by using overlapping $32 \times 48 \times 48$ patches. How does this compare to the segmentation approach?

14.10 Summary

- An unambiguous split between training and validation (and test) sets is crucial. Here, splitting by patient is much less prone to getting things wrong. This is even more true when you have several models in your pipeline.
- Getting from pixel-wise marks to nodules can be achieved using very traditional image processing. We don't want to look down on the classics, but value these tools and use them where appropriate.
- Our diagnosis script performs both segmentation and classification. This allows us to diagnose a CT that we have not seen before, though our current `Dataset` implementation is not configured to accept `series_uid`s from sources other than LUNA.
- Fine-tuning is a great way to fit a model while using a minimum of training data. Make sure the pretrained model has features relevant to your task, and make sure that you retrain a portion of the network with enough capacity.

- TensorBoard allows us to write out many different types of diagrams that help us determine what's going on. But this is not a replacement for looking at data on which our model works particularly badly.
- Successful training seems to involve an overfitting network at some stage, and which we then regularize. We might as well take that as a recipe; and we should probably learn more about regularization.
- Training neural networks is about trying things, seeing what goes wrong, and improving on it. There usually isn't a magic bullet.
- Kaggle is an excellent source of project ideas for deep learning. Many new datasets have cash prizes for the top performers, and older contests have examples that can be used as starting points for further experimentation.

Part 3

Deployment

In part 3, we'll look at how to get our models to the point where they can be used. We saw how to build models in the previous parts: part 1 introduced the building and training of models, and part 2 thoroughly covered an example from start to finish, so the hard work is done.

But no model is useful until you can actually use it. So, now we need to put the models out there and apply them to the tasks they are designed to solve. This part is closer to part 1 in spirit, because it introduces a lot of PyTorch components. As before, we'll focus on applications and tasks we wish to solve rather than just looking at PyTorch for its own sake.

In part 3's single chapter, we'll take a tour of the PyTorch deployment landscape as of early 2020. We'll get to know and use the PyTorch just-in-time compiler (JIT) to export models for use in third-party applications to the C++ API for mobile support.

15 Deploying to production

This chapter covers

- Options for deploying PyTorch models
- Working with the PyTorch JIT
- Deploying a model server and exporting models
- Running exported and natively implemented models from C++
- Running models on mobile

In part 1 of this book, we learned a lot about models; and part 2 left us with a detailed path for creating good models for a particular problem. Now that we have these great models, we need to take them where they can be useful. Maintaining infrastructure for executing inference of deep learning models at scale can be impactful from an architectural as well as cost standpoint. While PyTorch started off as a framework focused on research, beginning with the 1.0 release, a set of production-oriented features were added that today make PyTorch an ideal end-to-end platform from research to large-scale production.

What deploying to production means will vary with the use case:

- Perhaps the most natural deployment for the models we developed in part 2 would be to set up a network service providing access to our models. We'll do this in two versions using lightweight Python web frameworks: Flask (http://flask.pocoo.org) and Sanic (https://sanicframework.org). The first is arguably one of the most popular of these frameworks, and the latter is similar in spirit but takes advantage of Python's new async/await support for asynchronous operations for efficiency.
- We can export our model to a well-standardized format that allows us to ship it using optimized model processors, specialized hardware, or cloud services. For PyTorch models, the Open Neural Network Exchange (ONNX) format fills this role.
- We may wish to integrate our models into larger applications. For this it would be handy if we were not limited to Python. Thus we will explore using PyTorch models from C++ with the idea that this also is a stepping-stone to any language.
- Finally, for some things like the image zebraification we saw in chapter 2, it may be nice to run our model on mobile devices. While it is unlikely that you will have a CT module for your mobile, other medical applications like do-it-yourself skin screenings may be more natural, and the user might prefer running on the device versus having their skin sent to a cloud service. Luckily for us, PyTorch has gained mobile support recently, and we will explore that.

As we learn how to implement these use cases, we will use the classifier from chapter 14 as our first example for serving, and then switch to the zebraification model for the other bits of deployment.

15.1 Serving PyTorch models

We'll begin with what it takes to put our model on a server. Staying true to our hands-on approach, we'll start with the simplest possible server. Once we have something basic that works, we'll take look at its shortfalls and take a stab at resolving them. Finally, we'll look at what is, at the time of writing, the future. Let's get something that listens on the network.[1]

15.1.1 Our model behind a Flask server

Flask is one of the most widely used Python modules. It can be installed using `pip`:[2]

```
pip install Flask
```

[1] To play it safe, do not do this on an untrusted network.

[2] Or `pip3` for Python3. You also might want to run it from a Python virtual environment.

The API can be created by decorating functions.

Listing 15.1 flask_hello_world.py:1

```
from flask import Flask
app = Flask(__name__)

@app.route("/hello")
def hello():
  return "Hello World!"

if __name__ == '__main__':
  app.run(host='0.0.0.0', port=8000)
```

When started, the application will run at port 8000 and expose one route, /hello, that returns the "Hello World" string. At this point, we can augment our Flask server by loading a previously saved model and exposing it through a `POST` route. We will use the nodule classifier from chapter 14 as an example.

We'll use Flask's (somewhat curiously imported) `request` to get our data. More precisely, request.files contains a dictionary of file objects indexed by field names. We'll use JSON to parse the input, and we'll return a JSON string using flask's `jsonify` helper.

Instead of /hello, we will now expose a /predict route that takes a binary blob (the pixel content of the series) and the related metadata (a JSON object containing a dictionary with `shape` as a key) as input files provided with a `POST` request and returns a JSON response with the predicted diagnosis. More precisely, our server takes one sample (rather than a batch) and returns the probability that it is malignant.

In order to get to the data, we first need to decode the JSON to binary, which we can then decode into a one-dimensional array with `numpy.frombuffer`. We'll convert this to a tensor with `torch.from_numpy` and view its actual shape.

The actual handling of the model is just like in chapter 14: we'll instantiate `LunaModel` from chapter 14, load the weights we got from our training, and put the model in `eval` mode. As we are not training anything, we'll tell PyTorch that we will not want gradients when running the model by running in a `with torch.no_grad()` block.

Listing 15.2 flask_server.py:1

```
import numpy as np
import sys
import os
import torch
from flask import Flask, request, jsonify
import json

from p2ch13.model_cls import LunaModel

app = Flask(__name__)

model = LunaModel()
```

Sets up our model, loads the weights, and moves to evaluation mode

```
model.load_state_dict(torch.load(sys.argv[1],
                  map_location='cpu')['model_state'])
model.eval()

def run_inference(in_tensor):
  with torch.no_grad():                                  # No autograd for us.
    # LunaModel takes a batch and outputs a tuple (scores, probs)
    out_tensor = model(in_tensor.unsqueeze(0))[1].squeeze(0)
  probs = out_tensor.tolist()
  out = {'prob_malignant': probs[1]}
  return out

@app.route("/predict", methods=["POST"])                 # We expect a form submission (HTTP POST) at the "/predict" endpoint.
def predict():
  meta = json.load(request.files['meta'])                # Our request will have one file called meta.
  blob = request.files['blob'].read()
  in_tensor = torch.from_numpy(np.frombuffer(
    blob, dtype=np.float32))                             # Converts our data from binary blob to torch
  in_tensor = in_tensor.view(*meta['shape'])
  out = run_inference(in_tensor)
  return jsonify(out)                                    # Encodes our response content as JSON

if __name__ == '__main__':
  app.run(host='0.0.0.0', port=8000)
  print (sys.argv[1])
```

Run the server as follows:

```
python3 -m p3ch15.flask_server
➥ data/part2/models/cls_2019-10-19_15.48.24_final_cls.best.state
```

We prepared a trivial client at cls_client.py that sends a single example. From the code directory, you can run it as

```
python3 p3ch15/cls_client.py
```

It should tell you that the nodule is very unlikely to be malignant. Clearly, our server takes inputs, runs them through our model, and returns the outputs. So are we done? Not quite. Let's look at what could be better in the next section.

15.1.2 What we want from deployment

Let's collect some things we desire for serving models.[3] First, we want to support *modern protocols and their features*. Old-school HTTP is deeply serial, which means when a client wants to send several requests in the same connection, the next requests will only be sent after the previous request has been answered. Not very efficient if you want to send a batch of things. We will partially deliver here—our upgrade to Sanic certainly moves us to a framework that has the ambition to be very efficient.

[3] One of the earliest public talks discussing the inadequacy of Flask serving for PyTorch models is Christian Perone's "PyTorch under the Hood," http://mng.bz/xWdW.

When using GPUs, it is often much more efficient to *batch requests* than to process them one by one or fire them in parallel. So next, we have the task of collecting requests from several connections, assembling them into a batch to run on the GPU, and then getting the results back to the respective requesters. This sounds elaborate and (again, when we write this) seems not to be done very often in simple tutorials. That is reason enough for us to do it properly here! Note, though, that until latency induced by the duration of a model run is an issue (in that waiting for our own run is OK; but waiting for the batch that's running when the request arrives to finish, and then waiting for our run to give results, is prohibitive), there is little reason to run multiple batches on one GPU at a given time. Increasing the maximum batch size will generally be more efficient.

We want to serve several things *in parallel*. Even with asynchronous serving, we need our model to run efficiently on a second thread—this means we want to escape the (in)famous Python global interpreter lock (GIL) with our model.

We also want to do as *little copying* as possible. Both from a memory-consumption and a time perspective, copying things over and over is bad. Many HTTP things are encoded in Base64 (a format restricted to 6 bits per byte to encode binary in more or less alphanumeric strings), and—say, for images—decoding that to binary and then again to a tensor and then to the batch is clearly relatively expensive. We will partially deliver on this—we'll use streaming `PUT` requests to not allocate Base64 strings and to avoid growing strings by successively appending to them (which is terrible for performance for strings as much as tensors). We say we do not deliver completely because we are not truly minimizing the copying, though.

The last desirable thing for serving is *safety*. Ideally, we would have safe decoding. We want to guard against both overflows and resource exhaustion. Once we have a fixed-size input tensor, we should be mostly good, as it is hard to crash PyTorch starting from fixed-sized inputs. The stretch to get there, decoding images and the like, is likely more of a headache, and we make no guarantees. Internet security is a large enough field that we will not cover it at all. We should note that neural networks are known to be susceptible to manipulation of the inputs to generate desired but wrong or unforeseen outputs (known as *adversarial examples*), but this isn't extremely pertinent to our application, so we'll skip it here.

Enough talk. Let's improve on our server.

15.1.3 Request batching

Our second example server will use the Sanic framework (installed via the Python package of the same name). This will give us the ability to serve many requests in parallel using asynchronous processing, so we'll tick that off our list. While we are at it, we will also implement request batching.

Asynchronous programming can sound scary, and it usually comes with lots of terminology. But what we are doing here is just allowing functions to non-blockingly wait for results of computations or events.[4]

[4] Fancy people call these asynchronous function *generators* or sometimes, more loosely, *coroutines*: https://en.wikipedia.org/wiki/Coroutine.

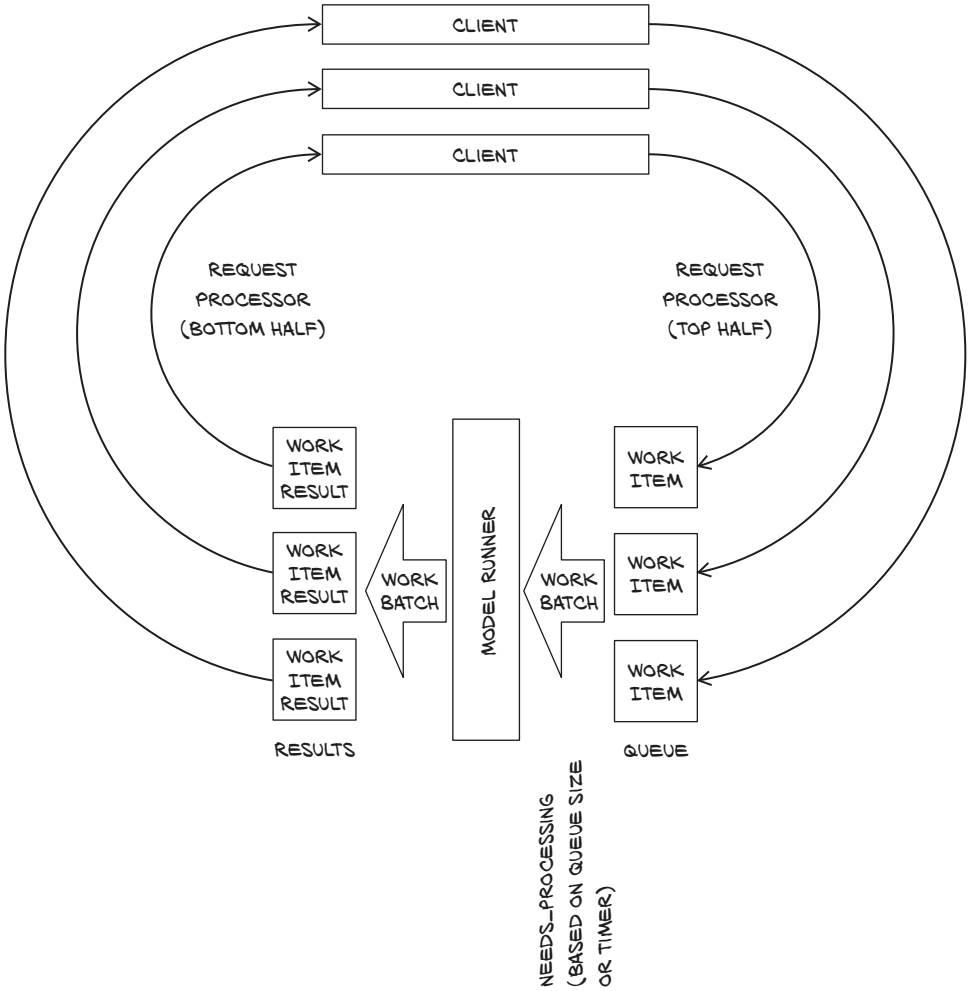


Figure 15.1 Dataflow with request batching

In order to do request batching, we have to decouple the request handling from running the model. Figure 15.1 shows the flow of the data.

At the top of figure 15.1 are the clients, making requests. One by one, these go through the top half of the request processor. They cause work items to be enqueued with the request information. When a full batch has been queued or the oldest request has waited for a specified maximum time, a model runner takes a batch from the queue, processes it, and attaches the result to the work items. These are then processed one by one by the bottom half of the request processor.

IMPLEMENTATION

We implement this by writing two functions. The model runner function starts at the beginning and runs forever. Whenever we need to run the model, it assembles a batch of inputs, runs the model in a second thread (so other things can happen), and returns the result.

The request processor then decodes the request, enqueues inputs, waits for the processing to be completed, and returns the output with the results. In order to appreciate what *asynchronous* means here, think of the model runner as a wastepaper basket. All the figures we scribble for this chapter can be quickly disposed of to the right of the desk. But every once in a while—either because the basket is full or when it is time to clean up in the evening—we need to take all the collected paper out to the trash can. Similarly, we enqueue new requests, trigger processing if needed, and wait for the results before sending them out as the answer to the request. Figure 15.2 shows our two functions in the blocks we execute uninterrupted before handing back to the event loop.

A slight complication relative to this picture is that we have two occasions when we need to process events: if we have accumulated a full batch, we start right away; and when the oldest request reaches the maximum wait time, we also want to run. We solve this by setting a timer for the latter.[5]

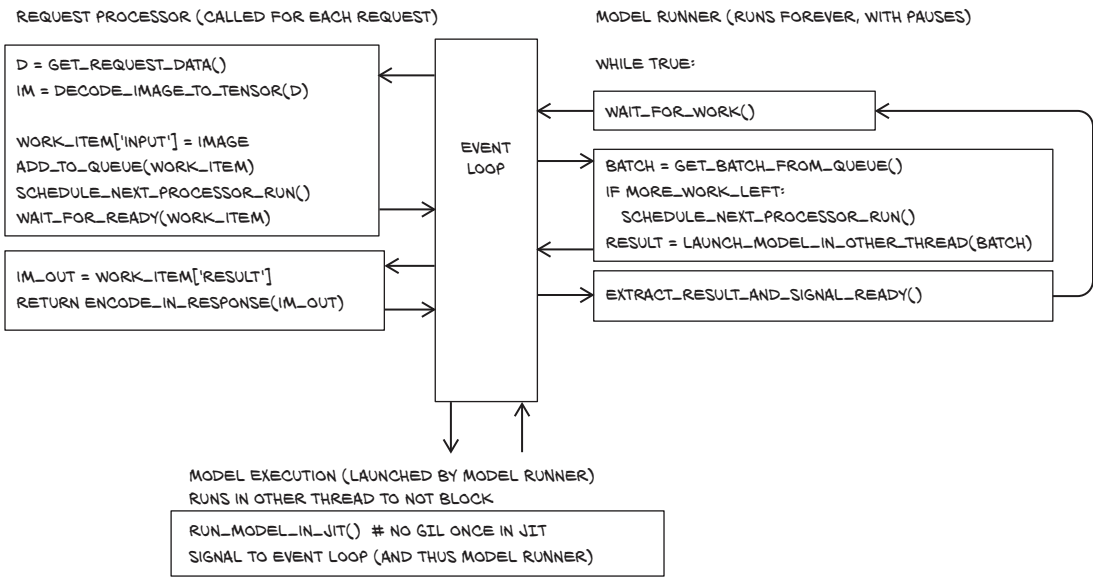


Figure 15.2 Our asynchronous server consists of three blocks: request processor, model runner, and model execution. These blocks are a bit like functions, but the first two will yield to the event loop in between.

[5] An alternative might be to forgo the timer and just run whenever the queue is not empty. This would potentially run smaller "first" batches, but the overall performance impact might not be so large for most applications.

All our interesting code is in a `ModelRunner` class, as shown in the following listing.

Listing 15.3 request_batching_server.py:32, `ModelRunner`

```
class ModelRunner:
  def __init__(self, model_name):
    self.model_name = model_name
    self.queue = []                          <-- The queue

    self.queue_lock = None                   <-- This will become our lock.

    self.model = get_pretrained_model(self.model_name,
                                      map_location=device)   <-- Loads and instantiates the model. This is the (only) thing we will need to change for switching to the JIT. For now, we import the CycleGAN (with the slight modification of standardizing to 0..1 input and output) from p3ch15/cyclegan.py.

    self.needs_processing = None             <-- Our signal to run the model

    self.needs_processing_timer = None       <-- Finally, the timer
```

`ModelRunner` first loads our model and takes care of some administrative things. In addition to the model, we also need a few other ingredients. We enter our requests into a `queue`. This is a just a Python list in which we add work items at the back and remove them in the front.

When we modify the `queue`, we want to prevent other tasks from changing the queue out from under us. To this effect, we introduce a `queue_lock` that will be an `asyncio.Lock` provided by the `asyncio` module. As all `asyncio` objects we use here need to know the event loop, which is only available after we initialize the application, we temporarily set it to `None` in the instantiation. While locking like this may not be strictly necessary because our methods do not hand back to the event loop while holding the lock, and operations on the queue are atomic thanks to the GIL, it does explicitly encode our underlying assumption. If we had multiple workers, we would need to look at locking. One caveat: Python's async locks are not threadsafe. (Sigh.)

`ModelRunner` waits when it has nothing to do. We need to signal it from `RequestProcessor` that it should stop slacking off and get to work. This is done via an `asyncio.Event` called `needs_processing`. `ModelRunner` uses the `wait()` method to wait for the `needs_processing` event. The `RequestProcessor` then uses `set()` to signal, and `ModelRunner` wakes up and `clear()`s the event.

Finally, we need a timer to guarantee a maximal wait time. This timer is created when we need it by using `app.loop.call_at`. It sets the `needs_processing` event; we just reserve a slot now. So actually, sometimes the event will be set directly because a batch is complete or when the timer goes off. When we process a batch before the timer goes off, we will clear it so we don't do too much work.

FROM REQUEST TO QUEUE

Next we need to be able to enqueue requests, the core of the first part of `RequestProcessor` in figure 15.2 (without the decoding and reencoding). We do this in our first `async` method, `process_input`.

Listing 15.4 request_batching_server.py:54

```
async def process_input(self, input):
  our_task = {"done_event": asyncio.Event(loop=app.loop),
        "input": input,
        "time": app.loop.time()}
  async with self.queue_lock:
    if len(self.queue) >= MAX_QUEUE_SIZE:
      raise HandlingError("I'm too busy", code=503)
    self.queue.append(our_task)
    self.schedule_processing_if_needed()

  await our_task["done_event"].wait()
  return our_task["output"]
```

Sets up the task data

With the lock, we add our task and …

… schedule processing. Processing will set needs_processing if we have a full batch. If we don't and no timer is set, it will set one to when the max wait time is up.

Waits (and hands back to the loop using await) for the processing to finish

We set up a little Python dictionary to hold our task's information: the `input` of course, the `time` it was queued, and a `done_event` to be set when the task has been processed. The processing adds an `output`.

Holding the queue lock (conveniently done in an `async with` block), we add our task to the queue and schedule processing if needed. As a precaution, we error out if the queue has become too large. Then all we have to do is wait for our task to be processed, and return it.

> **NOTE** It is important to use the loop time (typically a monotonic clock), which may be different from the `time.time()`. Otherwise, we might end up with events scheduled for processing before they have been queued, or no processing at all.

This is all we need for the request processing (except decoding and encoding).

Running batches from the queue

Next, let's look at the `model_runner` function on the right side of figure 15.2, which does the model invocation.

Listing 15.5 request_batching_server.py:71, `.run_model`

```
async def model_runner(self):
  self.queue_lock = asyncio.Lock(loop=app.loop)
  self.needs_processing = asyncio.Event(loop=app.loop)
  while True:
    await self.needs_processing.wait()
    self.needs_processing.clear()
    if self.needs_processing_timer is not None:
      self.needs_processing_timer.cancel()
      self.needs_processing_timer = None
    async with self.queue_lock:
      # ... line 87
      to_process = self.queue[:MAX_BATCH_SIZE]
      del self.queue[:len(to_process)]
```

Waits until there is something to do

Cancels the timer if it is set

Grabs a batch and schedules the running of the next batch, if needed

```
      self.schedule_processing_if_needed()
    batch = torch.stack([t["input"] for t in to_process], dim=0)
    # we could delete inputs here...

    result = await app.loop.run_in_executor(
      None, functools.partial(self.run_model, batch)
    )
    for t, r in zip(to_process, result):
      t["output"] = r
      t["done_event"].set()
    del to_process
```

Runs the model in a separate thread, moving data to the device and then handing over to the model. We continue processing after it is done.

Adds the results to the work-item and sets the ready event

As indicated in figure 15.2, `model_runner` does some setup and then infinitely loops (but yields to the event loop in between). It is invoked when the app is instantiated, so it can set up `queue_lock` and the `needs_processing` event we discussed earlier. Then it goes into the loop, `await`-ing the `needs_processing` event.

When an event comes, first we check whether a time is set and, if so, clear it, because we'll be processing things now. Then `model_runner` grabs a batch from the queue and, if needed, schedules the processing of the next batch. It assembles the batch from the individual tasks and launches a new thread that evaluates the model using `asyncio`'s `app.loop.run_in_executor`. Finally, it adds the outputs to the tasks and sets `done_event`.

And that's basically it. The web framework—roughly looking like Flask with `async` and `await` sprinkled in—needs a little wrapper. And we need to start the `model_runner` function on the event loop. As mentioned earlier, locking the queue really is not necessary if we do not have multiple runners taking from the queue and potentially interrupting each other, but knowing our code will be adapted to other projects, we stay on the safe side of losing requests.

We start our server with

```
python3 -m p3ch15.request_batching_server data/p1ch2/horse2zebra_0.4.0.pth
```

Now we can test by uploading the image data/p1ch2/horse.jpg and saving the result:

```
curl -T data/p1ch2/horse.jpg
➥ http://localhost:8000/image --output /tmp/res.jpg
```

Note that this server does get a few things right—it batches requests for the GPU and runs asynchronously—but we still use the Python mode, so the GIL hampers running our model in parallel to the request serving in the main thread. It will not be safe for potentially hostile environments like the internet. In particular, the decoding of request data seems neither optimal in speed nor completely safe.

In general, it would be nicer if we could have decoding where we pass the request stream to a function along with a preallocated memory chunk, and the function decodes the image from the stream to us. But we do not know of a library that does things this way.

15.2 Exporting models

So far, we have used PyTorch from the Python interpreter. But this is not always desirable: the GIL is still potentially blocking our improved web server. Or we might want to run on embedded systems where Python is too expensive or unavailable. This is when we export our model. There are several ways in which we can play this. We might go away from PyTorch entirely and move to more specialized frameworks. Or we might stay within the PyTorch ecosystem and use the JIT, a *just in time* compiler for a PyTorch-centric subset of Python. Even when we then run the JITed model in Python, we might be after two of its advantages: sometimes the JIT enables nifty optimizations, or—as in the case of our web server—we just want to escape the GIL, which JITed models do. Finally (but we take some time to get there), we might run our model under `libtorch`, the C++ library PyTorch offers, or with the derived Torch Mobile.

15.2.1 Interoperability beyond PyTorch with ONNX

Sometimes we want to leave the PyTorch ecosystem with our model in hand—for example, to run on embedded hardware with a specialized model deployment pipeline. For this purpose, Open Neural Network Exchange provides an interoperational format for neural networks and machine learning models (https://onnx.ai). Once exported, the model can be executed using any ONNX-compatible runtime, such as ONNX Runtime,[6] provided that the operations in use in our model are supported by the ONNX standard and the target runtime. It is, for example, quite a bit faster on the Raspberry Pi than running PyTorch directly. Beyond traditional hardware, a lot of specialized AI accelerator hardware supports ONNX (https://onnx.ai/supported-tools.html#deployModel).

In a way, a deep learning model is a program with a very specific instruction set, made of granular operations like matrix multiplication, convolution, `relu`, `tanh`, and so on. As such, if we can serialize the computation, we can reexecute it in another runtime that understands its low-level operations. ONNX is a standardization of a format describing those operations and their parameters.

Most of the modern deep learning frameworks support serialization of their computations to ONNX, and some of them can load an ONNX file and execute it (although this is not the case for PyTorch). Some low-footprint ("edge") devices accept an ONNX files as input and generate low-level instructions for the specific device. And some cloud computing providers now make it possible to upload an ONNX file and see it exposed through a REST endpoint.

In order to export a model to ONNX, we need to run a model with a dummy input: the values of the input tensors don't really matter; what matters is that they are the correct shape and type. By invoking the `torch.onnx.export` function, PyTorch

[6] The code lives at https://github.com/microsoft/onnxruntime, but be sure to read the privacy statement! Currently, building ONNX Runtime yourself will get you a package that does not send things to the mothership.

will *trace* the computations performed by the model and serialize them into an ONNX file with the provided name:

```
torch.onnx.export(seg_model, dummy_input, "seg_model.onnx")
```

The resulting ONNX file can now be run in a runtime, compiled to an edge device, or uploaded to a cloud service. It can be used from Python after installing `onnxruntime` or `onnxruntime-gpu` and getting a `batch` as a NumPy array.

Listing 15.6 onnx_example.py

```
import onnxruntime

sess = onnxruntime.InferenceSession("seg_model.onnx")
input_name = sess.get_inputs()[0].name
pred_onnx, = sess.run(None, {input_name: batch})
```

The ONNX runtime API uses sessions to define models and then calls the run method with a set of named inputs. This is a somewhat typical setup when dealing with computations defined in static graphs.

Not all TorchScript operators can be represented as standardized ONNX operators. If we export operations foreign to ONNX, we will get errors about unknown `aten` operators when we try to use the runtime.

15.2.2 PyTorch's own export: Tracing

When interoperability is not the key, but we need to escape the Python GIL or otherwise export our network, we can use PyTorch's own representation, called the *TorchScript graph*. We will see what that is and how the JIT that generates it works in the next section. But let's give it a spin right here and now.

The simplest way to make a TorchScript model is to trace it. This looks exactly like ONNX exporting. This isn't surprising, because that is what the ONNX model uses under the hood, too. Here we just feed dummy inputs into the model using the `torch.jit.trace` function. We import `UNetWrapper` from chapter 13, load the trained parameters, and put the model into evaluation mode.

Before we trace the model, there is one additional caveat: none of the parameters should require gradients, because using the `torch.no_grad()` context manager is strictly a runtime switch. Even if we trace the model within `no_grad` but then run it outside, PyTorch will record gradients. If we take a peek ahead at figure 15.4, we see why: after the model has been traced, we ask PyTorch to execute it. But the traced model will have parameters requiring gradients when executing the recorded operations, and they will make everything require gradients. To escape that, we would have to run the traced model in a `torch.no_grad` context. To spare us this—from experience, it is easy to forget and then be surprised by the lack of performance—we loop through the model parameters and set all of them to not require gradients.

But then all we need to do is call `torch.jit.trace`. [7]

Listing 15.7 trace_example.py

```
import torch
from p2ch13.model_seg import UNetWrapper

seg_dict = torch.load('data-unversioned/part2/models/p2ch13/seg_2019-10-20_15
➥ .57.21_none.best.state', map_location='cpu')
seg_model = UNetWrapper(in_channels=8, n_classes=1, depth=4, wf=3,
➥ padding=True, batch_norm=True, up_mode='upconv')
seg_model.load_state_dict(seg_dict['model_state'])
seg_model.eval()
for p in seg_model.parameters():
    p.requires_grad_(False)

dummy_input = torch.randn(1, 8, 512, 512)
traced_seg_model = torch.jit.trace(seg_model, dummy_input)
```

Sets the parameters to not require gradients

The tracing

The tracing gives us a warning:

```
TracerWarning: Converting a tensor to a Python index might cause the trace
to be incorrect. We can't record the data flow of Python values, so this
value will be treated as a constant in the future. This means the trace
might not generalize to other inputs!
  return layer[:, :, diff_y:(diff_y + target_size[0]), diff_x:(diff_x +
➥ target_size[1])]
```

This stems from the cropping we do in U-Net, but as long as we only ever plan to feed images of size 512 × 512 into the model, we will be OK. In the next section, we'll take a closer look at what causes the warning and how to get around the limitation it highlights if we need to. It will also be important when we want to convert models that are more complex than convolutional networks and U-Nets to TorchScript.

We can save the traced model

```
torch.jit.save(traced_seg_model, 'traced_seg_model.pt')
```

and load it back without needed anything but the saved file, and then we can call it:

```
loaded_model = torch.jit.load('traced_seg_model.pt')
prediction = loaded_model(batch)
```

The PyTorch JIT will keep the model's state from when we saved it: that we had put it into evaluation mode and that our parameters do not require gradients. If we had not taken care of it beforehand, we would need to use `with torch.no_grad():` in the execution.

[7] Strictly speaking, this traces the model as a function. Recently, PyTorch gained the ability to preserve more of the module structure using `torch.jit.trace_module`, but for us, the plain tracing is sufficient.

> **TIP** You can run the JITed and exported PyTorch model without keeping the source. However, we always want to establish a workflow where we automatically go from source model to installed JITed model for deployment. If we do not, we will find ourselves in a situation where we would like to tweak something with the model but have lost the ability to modify and regenerate. Always keep the source, Luke!

15.2.3 Our server with a traced model

Now is a good time to iterate our web server to what is, in this case, our final version. We can export the traced CycleGAN model as follows:

```
python3 p3ch15/cyclegan.py data/p1ch2/horse2zebra_0.4.0.pth
➥ data/p3ch15/traced_zebra_model.pt
```

Now we just need to replace the call to `get_pretrained_model` with `torch.jit.load` in our server (and drop the now-unnecessary `import` of `get_pretrained_model`). This also means our model runs independent of the GIL—and this is what we wanted our server to achieve here. For your convenience, we have put the small modifications in request_batching_jit_server.py. We can run it with the traced model file path as a command-line argument.

Now that we have had a taste of what the JIT can do for us, let's dive into the details!

15.3 Interacting with the PyTorch JIT

Debuting in PyTorch 1.0, the PyTorch JIT is at the center of quite a few recent innovations around PyTorch, not least of which is providing a rich set of deployment options.

15.3.1 What to expect from moving beyond classic Python/PyTorch

Quite often, Python is said to lack speed. While there is some truth to this, the tensor operations we use in PyTorch usually are in themselves large enough that the Python slowness between them is not a large issue. For small devices like smartphones, the memory overhead that Python brings might be more important. So keep in mind that frequently, the speedup gained by taking Python out of the computation is 10% or less.

Another immediate speedup from not running the model in Python only appears in multithreaded environments, but then it can be significant: because the intermediates are not Python objects, the computation is not affected by the menace of all Python parallelization, the GIL. This is what we had in mind earlier and realized when we used a traced model in our server.

Moving from the classic PyTorch way of executing one operation before looking at the next does give PyTorch a holistic view of the calculation: that is, it can consider the calculation in its entirety. This opens the door to crucial optimizations and higher-level transformations. Some of those apply mostly to inference, while others can also provide a significant speedup in training.

Let's use a quick example to give you a taste of why looking at several operations at once can be beneficial. When PyTorch runs a sequence of operations on the GPU, it calls a subprogram (*kernel*, in CUDA parlance) for each of them. Every kernel reads the input from GPU memory, computes the result, and then stores the result. Thus most of the time is typically spent not computing things, but reading from and writing to memory. This can be improved on by reading only once, computing several operations, and then writing at the very end. This is precisely what the PyTorch JIT fuser does. To give you an idea of how this works, figure 15.3 shows the pointwise computation taking place in long short-term memory (LSTM; https://en.wikipedia.org/wiki/Long_short-term_memory) cell, a popular building block for recurrent networks.

The details of figure 15.3 are not important to us here, but there are 5 inputs at the top, 2 outputs at the bottom, and 7 intermediate results represented as rounded indices. By computing all of this in one go in a single CUDA function and keeping the intermediates in registers, the JIT reduces the number of memory reads from 12 to 5 and the number of writes from 9 to 2. These are the large gains the JIT gets us; it can reduce the time to train an LSTM network by a factor of four. This seemingly simple

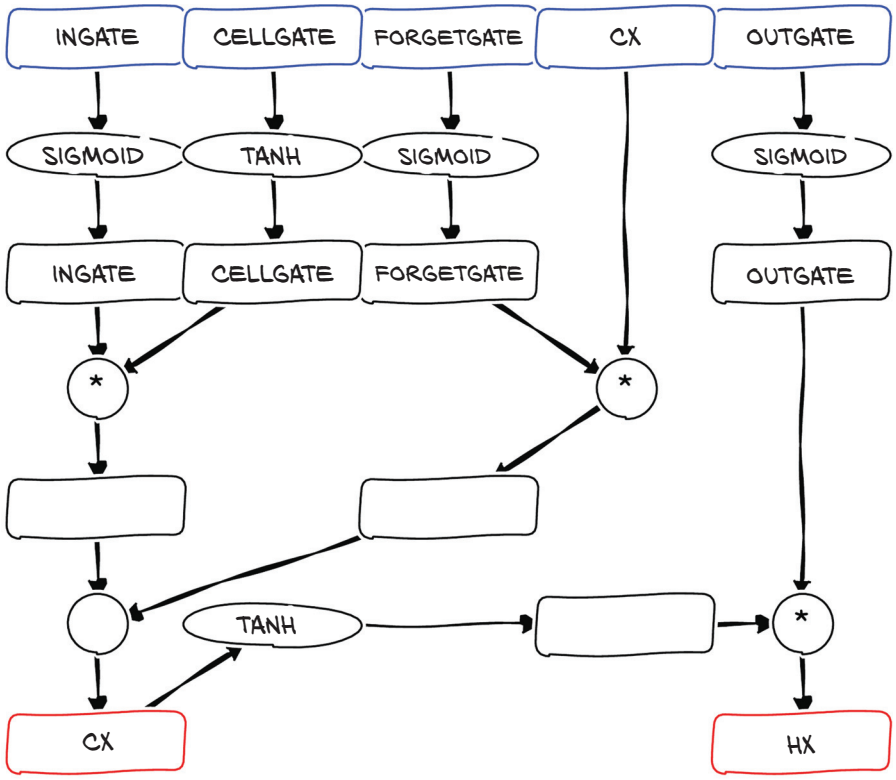


Figure 15.3 LSTM cell pointwise operations. From five inputs at the top, this block computes two outputs at the bottom. The boxes in between are intermediate results that vanilla PyTorch will store in memory but the JIT fuser will just keep in registers.

trick allows PyTorch to significantly narrow the gap between the speed of LSTM and generalized LSTM cells flexibly defined in PyTorch and the rigid but highly optimized LSTM implementation provided by libraries like cuDNN.

In summary, the speedup from using the JIT to escape Python is more modest than we might naively expect when we have been told that Python is awfully slow, but avoiding the GIL is a significant win for multithreaded applications. The large speedups in JITed models come from special optimizations that the JIT enables but that are more elaborate than just avoiding Python overhead.

15.3.2 The dual nature of PyTorch as interface and backend

To understand how moving beyond Python works, it is beneficial to mentally separate PyTorch into several parts. We saw a first glimpse of this in section 1.4. Our PyTorch `torch.nn` modules—which we first saw in chapter 6 and which have been our main tool for modeling ever since—hold the parameters of our network and are implemented using the functional interface: functions taking and returning tensors. These are implemented as a C++ extension, handed over to the C++-level autograd-enabled layer. (This then hands the actual computation to an internal library called ATen, performing the computation or relying on backends to do so, but this is not important.)

Given that the C++ functions are already there, the PyTorch developers made them into an official API. This is the nucleus of LibTorch, which allows us to write C++ tensor operations that look almost like their Python counterparts. As the `torch.nn` modules are Python-only by nature, the C++ API mirrors them in a namespace `torch::nn` that is designed to look a lot like the Python part but is independent.

This would allow us to redo in C++ what we did in Python. But that is not what we want: we want to *export* the model. Happily, there is another interface to the same functions provided by PyTorch: the PyTorch JIT. The PyTorch JIT provides a "symbolic" representation of the computation. This representation is the *TorchScript intermediate representation* (TorchScript IR, or sometimes just TorchScript). We mentioned TorchScript in section 15.2.2 when discussing delayed computation. In the following sections, we will see how to get this representation of our Python models and how they can be saved, loaded, and executed. Similar to what we discussed for the regular PyTorch API, the PyTorch JIT functions to load, inspect, and execute TorchScript modules can also be accessed both from Python and from C++.

In summary, we have four ways of calling PyTorch functions, illustrated in figure 15.4: from both C++ and Python, we can either call functions directly or have the JIT as an intermediary. All of these eventually call the C++ LibTorch functions and from there ATen and the computational backend.

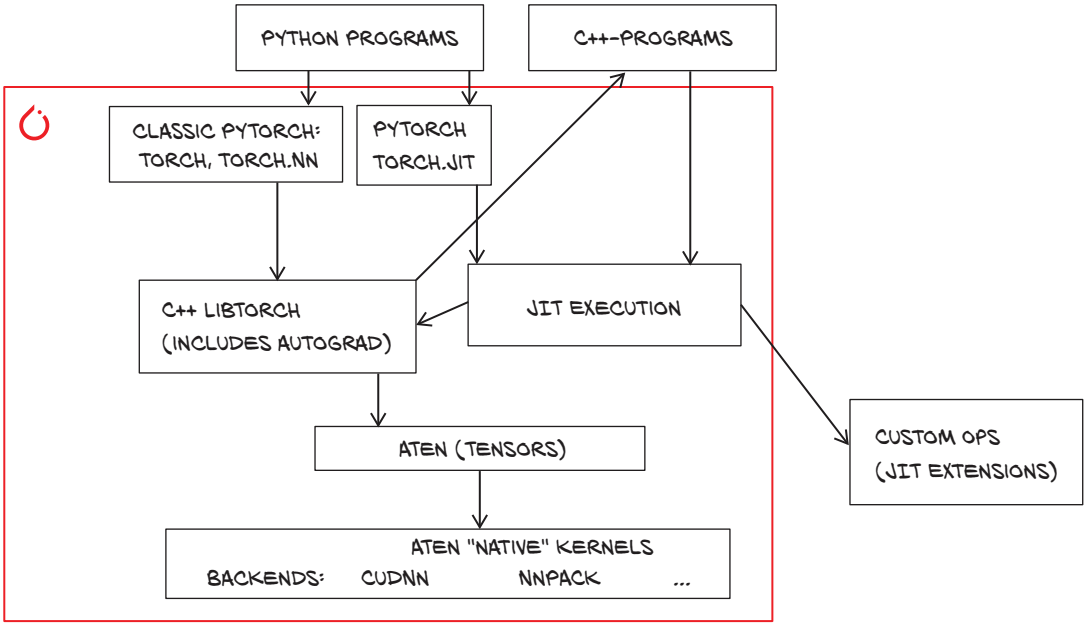


Figure 15.4 Many ways of calling into PyTorch

15.3.3 TorchScript

TorchScript is at the center of the deployment options envisioned by PyTorch. As such, it is worth taking a close look at how it works.

There are two straightforward ways to create a TorchScript model: tracing and scripting. We will look at each of them in the following sections. At a very high level, the two work as follows:

- In *tracing*, which we used in in section 15.2.2, we execute our usual PyTorch model using sample (random) inputs. The PyTorch JIT has hooks (in the C++ autograd interface) for every function that allows it to record the computation. In a way, it is like saying "Watch how I compute the outputs—now you can do the same." Given that the JIT only comes into play when PyTorch functions (and also `nn.Modules`) are called, you can run any Python code while tracing, but the JIT will only notice those bits (and notably be ignorant of control flow). When we use tensor shapes—usually a tuple of integers—the JIT tries to follow what's going on but may have to give up. This is what gave us the warning when tracing the U-Net.
- In *scripting*, the PyTorch JIT looks at the actual Python code of our computation and compiles it into the TorchScript IR. This means that, while we can be sure that every aspect of our program is captured by the JIT, we are restricted to those parts understood by the compiler. This is like saying "I am telling you how to do it—now you do the same." Sounds like programming, really.

We are not here for theory, so let's try tracing and scripting with a very simple function that adds inefficiently over the first dimension:

```
# In[2]:
def myfn(x):
    y = x[0]
    for i in range(1, x.size(0)):
        y = y + x[i]
    return y
```

We can trace it:

```
# In[3]:
inp = torch.randn(5,5)
traced_fn = torch.jit.trace(myfn, inp)
print(traced_fn.code)
```

```
# Out[3]:
def myfn(x: Tensor) -> Tensor:
  y = torch.select(x, 0, 0)
  y0 = torch.add(y, torch.select(x, 0, 1), alpha=1)
  y1 = torch.add(y0, torch.select(x, 0, 2), alpha=1)
  y2 = torch.add(y1, torch.select(x, 0, 3), alpha=1)
  _0 = torch.add(y2, torch.select(x, 0, 4), alpha=1)
  return _0
```

Indexing in the first line of our function

Our loop—but completely unrolled and fixed to 1…4 regardless of the size of x

Scary, but so true!

```
TracerWarning: Converting a tensor to a Python index might cause the trace
to be incorrect. We can't record the data flow of Python values, so this
value will be treated as a constant in the future. This means the
trace might not generalize to other inputs!
```

We see the big warning—and indeed, the code has fixed indexing and additions for five rows, and it would not deal as intended with four or six rows.

This is where scripting helps:

```
# In[4]:
scripted_fn = torch.jit.script(myfn)
print(scripted_fn.code)
```

```
# Out[4]:
def myfn(x: Tensor) -> Tensor:
  y = torch.select(x, 0, 0)
  _0 = torch.__range_length(1, torch.size(x, 0), 1)
  y0 = y
  for _1 in range(_0):
    i = torch.__derive_index(_1, 1, 1)
    y0 = torch.add(y0, torch.select(x, 0, i), alpha=1)
  return y0
```

PyTorch constructs the range length from the tensor size.

Our for loop—even if we have to take the funny-looking next line to get our index i

Our loop body, which is just a tad more verbose

We can also print the scripted graph, which is closer to the internal representation of TorchScript:

```
# In[5]:
xprint(scripted_fn.graph)
# end::cell_5_code[]

# tag::cell_5_output[]
# Out[5]:
graph(%x.1 : Tensor):
  %10 : bool = prim::Constant[value=1]()
  %2 : int = prim::Constant[value=0]()
  %5 : int = prim::Constant[value=1]()
  %y.1 : Tensor = aten::select(%x.1, %2, %2)
  %7 : int = aten::size(%x.1, %2)
  %9 : int = aten::__range_length(%5, %7, %5)
  %y : Tensor = prim::Loop(%9, %10, %y.1)
    block0(%11 : int, %y.6 : Tensor):
      %i.1 : int = aten::__derive_index(%11, %5, %5)
      %18 : Tensor = aten::select(%x.1, %2, %i.1)
      %y.3 : Tensor = aten::add(%y.6, %18, %5)
      -> (%10, %y.3)
  return (%y)
```

Seems a lot more verbose than we need

The first assignment of y

Constructing the range is recognizable after we see the code.

Our for loop returns the value (y) it calculates.

Body of the for loop: selects a slice, and adds to y

In practice, you would most often use `torch.jit.script` in the form of a decorator:

```
@torch.jit.script
def myfn(x):
  ...
```

You could also do this with a custom `trace` decorator taking care of the inputs, but this has not caught on.

Although TorchScript (the language) looks like a subset of Python, there are fundamental differences. If we look very closely, we see that PyTorch has added type specifications to the code. This hints at an important difference: TorchScript is statically typed—every value (variable) in the program has one and only one type. Also, the types are limited to those for which the TorchScript IR has a representation. Within the program, the JIT will usually infer the type automatically, but we need to annotate any non-tensor arguments of scripted functions with their types. This is in stark contrast to Python, where we can assign anything to any variable.

So far, we've traced functions to get scripted functions. But we graduated from just using functions in chapter 5 to using modules a long time ago. Sure enough, we can also trace or script models. These will then behave roughly like the modules we know and love. For both tracing and scripting, we pass an instance of `Module` to `torch.jit.trace` (with sample inputs) or `torch.jit.script` (without sample inputs), respectively. This will give us the `forward` method we are used to. If we want to expose other methods (this only works in `scripting`) to be called from the outside, we decorate them with `@torch.jit.export` in the class definition.

When we said that the JITed modules work like they did in Python, this includes the fact that we can use them for training, too. On the flip side, this means we need to set them up for inference (for example, using the `torch.no_grad()` context) just like our traditional models, to make them do the right thing.

With algorithmically relatively simple models—like the CycleGAN, classification models and U-Net-based segmentation—we can just trace the model as we did earlier. For more complex models, a nifty property is that we can use scripted or traced functions from other scripted or traced code, and that we can use scripted or traced submodules when constructing and tracing or scripting a module. We can also trace functions by calling `nn.Models`, but then we need to set all parameters to not require gradients, as the parameters will be constants for the traced model.

As we have seen tracing already, let's look at a practical example of scripting in more detail.

15.3.4 Scripting the gaps of traceability

In more complex models, such as those from the Fast R-CNN family for detection or recurrent networks used in natural language processing, the bits with control flow like `for` loops need to be scripted. Similarly, if we needed the flexibility, we would find the code bit the tracer warned about.

Listing 15.8 From utils/unet.py

```
class UNetUpBlock(nn.Module):
    ...
    def center_crop(self, layer, target_size):
        _, _, layer_height, layer_width = layer.size()
        diff_y = (layer_height - target_size[0]) // 2
        diff_x = (layer_width - target_size[1]) // 2
        return layer[:, :, diff_y:(diff_y + target_size[0]),
➥ diff_x:(diff_x + target_size[1])]            <-- The tracer warns here.

    def forward(self, x, bridge):
        ...
        crop1 = self.center_crop(bridge, up.shape[2:])
 ...
```

What happens is that the JIT magically replaces the shape tuple `up.shape` with a 1D integer tensor with the same information. Now the slicing `[2:]` and the calculation of `diff_x` and `diff_y` are all traceable tensor operations. However, that does not save us, because the slicing then wants Python `ints`; and there, the reach of the JIT ends, giving us the warning.

But we can solve this issue in a straightforward way: we script `center_crop`. We slightly change the cut between caller and callee by passing `up` to the scripted `center_crop` and extracting the sizes there. Other than that, all we need is to add the `@torch.jit.script` decorator. The result is the following code, which makes the U-Net model traceable without warnings.

Listing 15.9 Rewritten excerpt from utils/unet.py

```
@torch.jit.script
def center_crop(layer, target):
    _, _, layer_height, layer_width = layer.size()
    _, _, target_height, target_width = target.size()
    diff_y = (layer_height - target_height) // 2
    diff_x = (layer_width - target_width]) // 2
    return layer[:, :, diff_y:(diff_y + target_height),
➥   diff_x:(diff_x + target_width)]

class UNetUpBlock(nn.Module):
    ...

    def forward(self, x, bridge):
        ...
        crop1 = center_crop(bridge, up)
  ...
```

Changes the signature, taking target instead of target_size

Gets the sizes within the scripted part

The indexing uses the size values we got.

We adapt our call to pass up rather than the size.

Another option we could choose—but that we will not use here—would be to move unscriptable things into custom operators implemented in C++. The TorchVision library does that for some specialty operations in Mask R-CNN models.

15.4 LibTorch: PyTorch in C++

We have seen various way to export our models, but so far, we have used Python. We'll now look at how we can forgo Python and work with C++ directly.

Let's go back to the horse-to-zebra CycleGAN example. We will now take the JITed model from section 15.2.3 and run it from a C++ program.

15.4.1 Running JITed models from C++

The hardest part about deploying PyTorch vision models in C++ is choosing an image library to choose the data.[8] Here, we go with the very lightweight library CImg (http://cimg.eu). If you are very familiar with OpenCV, you can adapt the code to use that instead; we just felt that CImg is easiest for our exposition.

Running a JITed model is very simple. We'll first show the image handling; it is not really what we are after, so we will do this very quickly.[9]

Listing 15.10 cyclegan_jit.cpp

```
#include "torch/script.h"
#define cimg_use_jpeg
#include "CImg.h"
using namespace cimg_library;
int main(int argc, char **argv) {
  CImg<float> image(argv[2]);
```

Includes the PyTorch script header and CImg with native JPEG support

Loads and decodes the image into a float array

[8] But TorchVision may develop a convenience function for loading images.

[9] The code works with PyTorch 1.4 and, hopefully, above. In PyTorch versions before 1.3 you needed `data` in place of `data_ptr`.

```
  image = image.resize(227, 227);                          <-- Resizes to a smaller size
  // ...here we need to produce an output tensor from input
  CImg<float> out_img(output.data_ptr<float>(), output.size(2),
                      output.size(3), 1, output.size(1));
  out_img.save(argv[3]);   <-- Saves the image
  return 0;
}
```

The method data_ptr<float>() gives us a pointer to the tensor storage. With it and the shape information, we can construct the output image.

For the PyTorch side, we include a C++ header torch/script.h. Then we need to set up and include the `CImg` library. In the `main` function, we load an image from a file given on the command line and resize it (in CImg). So we now have a 227 × 227 image in the `CImg<float>` variable `image`. At the end of the program, we'll create an `out_img` of the same type from our `(1, 3, 277, 277)`-shaped tensor and save it.

Don't worry about these bits. They are not the PyTorch C++ we want to learn, so we can just take them as is.

The actual computation is straightforward, too. We need to make an input tensor from the image, load our model, and run the input tensor through it.

Listing 15.11 cyclegan_jit.cpp

Puts the image data into a tensor

```
auto input_ = torch::tensor(
    torch::ArrayRef<float>(image.data(), image.size()));
auto input = input_.reshape({1, 3, image.height(),
                  image.width()}).div_(255);

auto module = torch::jit::load(argv[1]);

std::vector<torch::jit::IValue> inputs;
inputs.push_back(input);
auto output_ = module.forward(inputs).toTensor();

auto output = output_.contiguous().mul_(255);
```

Reshapes and rescales to move from CImg conventions to PyTorch's

Loads the JITed model or function from a file

Packs the input into a (one-element) vector of IValues

Calls the module and extracts the result tensor. For efficiency, the ownership is moved, so if we held on to the IValue, it would be empty afterward.

Makes sure our result is contiguous

Recall from chapter 3 that PyTorch keeps the values of a tensor in a large chunk of memory in a particular order. So does CImg, and we can get a pointer to this memory chunk (as a `float` array) using `image.data()` and the number of elements using `image.size()`. With these two, we can create a somewhat smarter reference: a `torch::ArrayRef` (which is just shorthand for pointer plus size; PyTorch uses those at the C++ level for data but also for returning sizes without copying). Then we can just parse that into the `torch::tensor` constructor, just as we would with a list.

> **TIP** Sometimes you might want to use the similar-working `torch::from_blob` instead of `torch::tensor`. The difference is that `tensor` will copy the data. If you do not want copying, you can use `from_blob`, but then you need to take care that the underpinning memory is available during the lifetime of the tensor.

Our tensor is only 1D, so we need to reshape it. Conveniently, CImg uses the same ordering as PyTorch (channel, rows, columns). If not, we would need to adapt the reshaping and permute the axes as we did in chapter 4. As CImg uses a range of 0…255 and we made our model to use 0…1, we divide here and multiply later. This could, of course, be absorbed into the model, but we wanted to reuse our traced model.

A common pitfall to avoid: pre- and postprocessing

When switching from one library to another, it is easy to forget to check that the conversion steps are compatible. They are non-obvious unless we look up the memory layout and scaling convention of PyTorch and the image processing library we use. If we forget, we will be disappointed by not getting the results we anticipate.

Here, the model would go wild because it gets extremely large inputs. However, in the end, the output convention of our model is to give RGB values in the 0..1 range. If we used this directly with CImg, the result would look all black.

Other frameworks have other conventions: for example OpenCV likes to store images as BGR instead of RGB, requiring us to flip the channel dimension. We always want to make sure the input we feed to the model in the deployment is the same as what we fed into it in Python.

Loading the traced model is very straightforward using `torch::jit::load`. Next, we have to deal with an abstraction PyTorch introduces to bridge between Python and C++: we need to wrap our input in an `IValue` (or several `IValues`), the *generic* data type for any value. A function in the JIT is passed a vector of `IValues`, so we declare that and then `push_back` our input tensor. This will automatically wrap our tensor into an `IValue`. We feed this vector of `IValues` to the forward and get a single one back. We can then unpack the tensor in the resulting `IValue` with `.toTensor`.

Here we see a bit about `IValues`: they have a type (here, `Tensor`), but they could also be holding `int64_ts` or `doubles` or a list of tensors. For example, if we had multiple outputs, we would get an `IValue` holding a list of tensors, which ultimately stems from the Python calling conventions. When we unpack a tensor from an `IValue` using `.toTensor`, the `IValue` transfers ownership (becomes invalid). But let's not worry about it; we got a tensor back. Because sometimes the model may return non-contiguous data (with gaps in the storage from chapter 3), but `CImg` reasonably requires us to provide it with a contiguous block, we call `contiguous`. It is important that we assign this contiguous tensor to a variable that is in scope until we are done working with the underlying memory. Just like in Python, PyTorch will free memory if it sees that no tensors are using it anymore.

So let's compile this! On Debian or Ubuntu, you need to install `cimg-dev`, `libjpeg-dev`, and `libx11-dev` to use `CImg`.

You can download a C++ library of PyTorch from the PyTorch page. But given that we already have PyTorch installed,[10] we might as well use that; it comes with all we need for C++. We need to know where our PyTorch installation lives, so open Python and check `torch.__file__`, which may say /usr/local/lib/python3.7/dist-packages/torch/__init__.py. This means the CMake files we need are in /usr/local/lib/python3.7/dist-packages/torch/share/cmake/.

While using CMake seems like overkill for a single source file project, linking to PyTorch is a bit complex; so we just use the following as a boilerplate CMake file.[11]

Listing 15.12 CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0 FATAL_ERROR)
project(cyclegan-jit)

find_package(Torch REQUIRED)
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${TORCH_CXX_FLAGS}")

add_executable(cyclegan-jit cyclegan_jit.cpp)
target_link_libraries(cyclegan-jit pthread jpeg X11)
target_link_libraries(cyclegan-jit "${TORCH_LIBRARIES}")
set_property(TARGET cyclegan-jit PROPERTY CXX_STANDARD 14)
```

Project name. Replace it with your own here and on the other lines.

We need Torch.

We want to compile an executable named cyclegan-jit from the cyclegan_jit.cpp source file.

Links to the bits required for CImg. CImg itself is all-include, so it does not appear here.

It is best to make a build directory as a subdirectory of where the source code resides and then in it run CMake as[12] `CMAKE_PREFIX_PATH=/usr/local/lib/python3.7/dist-packages/torch/share/cmake/ cmake ..` and finally `make`. This will build the `cyclegan-jit` program, which we can then run as follows:

```
./cyclegan-jit ../traced_zebra_model.pt  ../../data/p1ch2/horse.jpg /tmp/z.jpg
```

We just ran our PyTorch model without Python. Awesome! If you want to ship your application, you likely want to copy the libraries from /usr/local/lib/python3.7/dist-packages/torch/lib into where your executable is, so that they will always be found.

15.4.2 C++ from the start: The C++ API

The C++ modular API is intended to feel a lot like the Python one. To get a taste, we will translate the CycleGAN generator into a model natively defined in C++, but without the JIT. We do, however, need the pretrained weights, so we'll save a traced version of the model (and here it is important to trace not a function but the model).

[10] We hope you have not been slacking off about trying out things you read.

[11] The code directory has a bit longer version to work around Windows issues.

[12] You might have to replace the path with where your PyTorch or LibTorch installation is located. Note that the C++ library can be more picky than the Python one in terms of compatibility: If you are using a CUDA-enabled library, you need to have the matching CUDA headers installed. If you get cryptic error messages about "Caffe2 using CUDA," you need to install a CPU-only version of the library, but CMake found a CUDA-enabled one.

We'll start with some administrative details: includes and namespaces.

Listing 15.13 cyclegan_cpp_api.cpp

```
#include <torch/torch.h>
#define cimg_use_jpeg
#include <CImg.h>
using torch::Tensor;
```

Imports the one-stop torch/torch.h header and CImg

Spelling out torch::Tensor can be tedious, so we import the name into the main namespace.

When we look at the source code in the file, we find that `ConvTransposed2d` is ad hoc defined, when ideally it should be taken from the standard library. The issue here is that the C++ modular API is still under development; and with PyTorch 1.4, the premade `ConvTranspose2d` module cannot be used in `Sequential` because it takes an optional second argument.[13] Usually we could just leave `Sequential`—as we did for Python—but we want our model to have the same structure as the Python CycleGAN generator from chapter 2.

Next, let's look at the residual block.

Listing 15.14 Residual block in cyclegan_cpp_api.cpp

```
struct ResNetBlock : torch::nn::Module {
  torch::nn::Sequential conv_block;
  ResNetBlock(int64_t dim)
      : conv_block(
          torch::nn::ReflectionPad2d(1),
          torch::nn::Conv2d(torch::nn::Conv2dOptions(dim, dim, 3)),
          torch::nn::InstanceNorm2d(
          torch::nn::InstanceNorm2dOptions(dim)),
          torch::nn::ReLU(/*inplace=*/true),
        torch::nn::ReflectionPad2d(1),
          torch::nn::Conv2d(torch::nn::Conv2dOptions(dim, dim, 3)),
          torch::nn::InstanceNorm2d(
          torch::nn::InstanceNorm2dOptions(dim))) {
    register_module("conv_block", conv_block);
  }

  Tensor forward(const Tensor &inp) {
    return inp + conv_block->forward(inp);
  }
};
```

Initializes Sequential, including its submodules

Always remember to register the modules you assign, or bad things will happen!

As might be expected, our forward function is pretty simple.

Just as we would in Python, we register a subclass of `torch::nn::Module`. Our residual block has a sequential `conv_block` submodule.

And just as we did in Python, we need to initialize our submodules, notably `Sequential`. We do so using the C++ initialization statement. This is similar to how we

[13] This is a great improvement over PyTorch 1.3, where we needed to implement custom modules for ReLU, `ÌnstanceNorm2d`, and others.

construct submodules in Python in the `__init__` constructor. Unlike Python, C++ does not have the introspection and hooking capabilities that enable redirection of `__setattr__` to combine assignment to a member and registration.

Since the lack of keyword arguments makes the parameter specification awkward with default arguments, modules (like tensor factory functions) typically take an `options` argument. Optional keyword arguments in Python correspond to methods of the options object that we can chain. For example, the Python module `nn.Conv2d(in_channels, out_channels, kernel_size, stride=2, padding=1)` that we need to convert translates to `torch::nn::Conv2d(torch::nn::Conv2dOptions (in_channels, out_channels, kernel_size).stride(2).padding(1))`. This is a bit more tedious, but you're reading this because you love C++ and aren't deterred by the hoops it makes you jump through.

We should always take care that registration and assignment to members is in sync, or things will not work as expected: for example, loading and updating parameters during training will happen to the registered module, but the actual module being called is a member. This synchronization was done behind the scenes by the Python `nn.Module` class, but it is not automatic in C++. Failing to do so will cause us many headaches.

In contrast to what we did (and should!) in Python, we need to call `m->forward(…)` for our modules. Some modules can also be called directly, but for `Sequential`, this is not currently the case.

A final comment on calling conventions is in order: depending on whether you modify tensors provided to functions,[14] tensor arguments should always be passed as `const Tensor&` for tensors that are left unchanged or `Tensor` if they are changed. Tensors should be returned as `Tensor`. Wrong argument types like non-const references (`Tensor&`) will lead to unparsable compiler errors.

In the main generator class, we'll follow a typical pattern in the C++ API more closely by naming our class `ResNetGeneratorImpl` and promoting it to a torch module `ResNetGenerator` using the `TORCH_MODULE` macro. The background is that we want to mostly handle modules as references or shared pointers. The wrapped class accomplishes this.

Listing 15.15 `ResNetGenerator` in cyclegan_cpp_api.cpp

```
struct ResNetGeneratorImpl : torch::nn::Module {
  torch::nn::Sequential model;
  ResNetGeneratorImpl(int64_t input_nc = 3, int64_t output_nc = 3,
                      int64_t ngf = 64, int64_t n_blocks = 9) {
    TORCH_CHECK(n_blocks >= 0);
    model->push_back(torch::nn::ReflectionPad2d(3));
```

Adds modules to the Sequential container in the constructor. This allows us to add a variable number of modules in a for loop.

[14] This is a bit blurry because you can create a new tensor sharing memory with an input and modify it in place, but it's best to avoid that if possible.

```
    ...
      model->push_back(torch::nn::Conv2d(
          torch::nn::Conv2dOptions(ngf * mult, ngf * mult * 2, 3)
              .stride(2)
              .padding(1)));
    ...
    register_module("model", model);
  }
  Tensor forward(const Tensor &inp) { return model->forward(inp); }
};

TORCH_MODULE(ResNetGenerator);
```

Spares us from reproducing some tedious things

An example of Options in action

Creates a wrapper ResNetGenerator around our ResNetGeneratorImpl class. As archaic as it seems, the matching names are important here.

That's it—we've defined the perfect C++ analogue of the Python `ResNetGenerator` model. Now we only need a `main` function to load parameters and run our model. Loading the image with CImg and converting from image to tensor and tensor back to image are the same as in the previous section. To include some variation, we'll display the image instead of writing it to disk.

Listing 15.16 cyclegan_cpp_api.cpp `main`

```
ResNetGenerator model;
  ...
  torch::load(model, argv[1]);
  ...
  cimg_library::CImg<float> image(argv[2]);
  image.resize(400, 400);
  auto input_ =
      torch::tensor(torch::ArrayRef<float>(image.data(), image.size()));
  auto input = input_.reshape({1, 3, image.height(), image.width()});
  torch::NoGradGuard no_grad;

  model->eval();

  auto output = model->forward(input);
  ...
  cimg_library::CImg<float> out_img(output.data_ptr<float>(),
                    output.size(3), output.size(2),
                    1, output.size(1));
  cimg_library::CImgDisplay disp(out_img, "See a C++ API zebra!");
  while (!disp.is_closed()) {
    disp.wait();
  }
```

Instantiates our model

Loads the parameters

Declaring a guard variable is the equivalent of the torch.no_grad() context. You can put it in a { ... } block if you need to limit how long you turn off gradients.

As in Python, eval mode is turned on (for our model, it would not be strictly relevant).

Again, we call forward rather than the model.

Displaying the image, we need to wait for a key rather than immediately exiting our program.

The interesting changes are in how we create and run the model. Just as expected, we instantiate the model by declaring a variable of the model type. We load the model using `torch::load` (here it is important that we wrapped the model). While this looks very familiar to PyTorch practitioners, note that it will work on JIT-saved files rather than Python-serialized state dictionaries.

When running the model, we need the equivalent of `with torch.no_grad():`. This is provided by instantiating a variable of type `NoGradGuard` and keeping it in scope for

as long as we do not want gradients. Just like in Python, we set the model into evaluation mode calling `model->eval()`. This time around, we call `model->forward` with our input tensor and get a tensor as a result—no JIT is involved, so we do not need `IValue` packing and unpacking.

Phew. Writing this in C++ was a lot of work for the Python fans that we are. We are glad that we only promised to do inference here, but of course LibTorch also offers optimizers, data loaders, and much more. The main reason to use the API is, of course, when you want to create models and neither the JIT nor Python is a good fit.

For your convenience, CMakeLists.txt contains also the instructions for building `cyclegan-cpp-api`, so building is just like in the previous section.

We can run the program as

```
./cyclegan_cpp_api ../traced_zebra_model.pt ../../data/p1ch2/horse.jpg
```

But we knew what the model would be doing, didn't we?

15.5 Going mobile

As the last variant of deploying a model, we will consider deployment to mobile devices. When we want to bring our models to mobile, we are typically looking at Android and/or iOS. Here, we'll focus on Android.

The C++ parts of PyTorch—LibTorch—can be compiled for Android, and we could access that from an app written in Java using the Android Java Native Interface (JNI). But we really only need a handful of functions from PyTorch—loading a JITed model, making inputs into tensors and `IValue`s, running them through the model, and getting results back. To save us the trouble of using the JNI, the PyTorch developers wrapped these functions into a small library called PyTorch Mobile.

The stock way of developing apps in Android is to use the Android Studio IDE, and we will be using it, too. But this means there are a few dozen files of administrativa—which also happen to change from one Android version to the next. As such, we focus on the bits that turn one of the Android Studio templates (Java App with Empty Activity) into an app that takes a picture, runs it through our zebra-CycleGAN, and displays the result. Sticking with the theme of the book, we will be efficient with the Android bits (and they can be painful compared with writing PyTorch code) in the example app.

To infuse life into the template, we need to do three things. First, we need to define a UI. To keep things as simple as we can, we have two elements: a `TextView` named `headline` that we can click to take and transform a picture; and an `ImageView` to show our picture, which we call `image_view`. We will leave the picture-taking to the camera app (which you would likely avoid doing in an app for a smoother user experience), because dealing with the camera directly would blur our focus on deploying PyTorch models.[15]

[15] We are very proud of the topical metaphor.

Then, we need to include PyTorch as a dependency. This is done by editing our app's build.gradle file and adding `pytorch_android` and `pytorch_android_torchvision`.

Listing 15.17 Additions to build.gradle

```
dependencies {
  ...
  implementation 'org.pytorch:pytorch_android:1.4.0'

  implementation 'org.pytorch:pytorch_android_torchvision:1.4.0'
}
```

The dependencies section is very likely already there. If not, add it at the bottom.

The pytorch_android library gets the core things mentioned in the text.

The helper library pytorch_android_torchvision—perhaps a bit immodestly named when compared to its larger TorchVision sibling—contains a few utilities to convert bitmap objects to tensors, but at the time of writing not much more.

We need to add our traced model as an asset.

Finally, we can get to the meat of our shiny app: the Java class derived from activity that contains our main code. We'll just discuss an excerpt here. It starts with imports and model setup.

Listing 15.18 MainActivity.java part 1

```
...
import org.pytorch.IValue;
import org.pytorch.Module;
import org.pytorch.Tensor;
import org.pytorch.torchvision.TensorImageUtils;
...
public class MainActivity extends AppCompatActivity {
  private org.pytorch.Module model;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    ...
    try {
      model = Module.load(assetFilePath(this, "traced_zebra_model.pt"));
    } catch (IOException e) {
      Log.e("Zebraify", "Error reading assets", e);
      finish();
    }
    ...
  }
  ...
}
```

Don't you love imports?

Holds our JITed model

In Java we have to catch the exceptions.

Loads the module from a file

We need some imports from the `org.pytorch` namespace. In the typical style that is a hallmark of Java, we import `IValue`, `Module`, and `Tensor`, which do what we might expect; and the class `org.pytorch.torchvision.TensorImageUtils`, which holds utility functions to convert between tensors and images.

First, of course, we need to declare a variable holding our model. Then, when our app is started—in `onCreate` of our activity—we'll load the module using the `Model.load`

method from the location given as an argument. There is a slight complication though: apps' data is provided by the supplier as *assets* that are not easily accessible from the filesystem. For this reason, a utility method called `assetFilePath` (taken from the PyTorch Android examples) copies the asset to a location in the filesystem. Finally, in Java, we need to catch exceptions that our code throws, unless we want to (and are able to) declare the method we are coding as throwing them in turn.

When we get an image from the camera app using Android's `Intent` mechanism, we need to run it through our model and display it. This happens in the `onActivityResult` event handler.

Listing 15.19 MainActivity.java, part 2

```
@Override
protected void onActivityResult(int requestCode, int resultCode,
                                Intent data) {
  if (requestCode == REQUEST_IMAGE_CAPTURE &&
      resultCode == RESULT_OK) {
    Bitmap bitmap = (Bitmap) data.getExtras().get("data");

    final float[] means = {0.0f, 0.0f, 0.0f};
    final float[] stds = {1.0f, 1.0f, 1.0f};

    final Tensor inputTensor = TensorImageUtils.bitmapToFloat32Tensor(
        bitmap, means, stds);

    final Tensor outputTensor = model.forward(
        IValue.from(inputTensor)).toTensor();
    Bitmap output_bitmap = tensorToBitmap(outputTensor, means, stds,
        Bitmap.Config.RGB_565);
    image_view.setImageBitmap(output_bitmap);
  }
}
```

Performs normalization, but the default is images in the range of 0…1 so we do not need to transform: that is, have 0 shift and a scaling divisor of 1.

Gets a tensor from a bitmap, combining steps like TorchVision's ToTensor (converting to a float tensor with entries between 0 and 1) and Normalize

This is executed when the camera app takes a picture.

This looks almost like what we did in C++.

tensorToBitmap is our own invention.

Converting the bitmap we get from Android to a tensor is handled by the `TensorImageUtils.bitmapToFloat32Tensor` function (static method), which takes two float arrays, `means` and `stds`, in addition to `bitmap`. Here we specify the mean and standard deviation of our input data(set), which will then be mapped to have zero mean and unit standard deviation just like TorchVision's `Normalize` transform. Android already gives us the images in the 0..1 range that we need to feed into our model, so we specify mean 0 and standard deviation 1 to prevent the normalization from changing our image.

Around the actual call to `model.forward`, we then do the same `IValue` wrapping and unwrapping dance that we did when using the JIT in C++, except that our `forward` takes a single `IValue` rather than a vector of them. Finally, we need to get back to a bitmap. Here PyTorch will not help us, so we need to define our own `tensorToBitmap` (and submit the pull request to PyTorch). We spare you the details here, as they are tedious

and full of copying (from the tensor to a `float[]` array to a `int[]` array containing ARGB values to the bitmap), but it is as it is. It is designed to be the inverse of `bitmapToFloat32Tensor`.

And that's all we need to do to get PyTorch into Android. Using the minimal additions to the code we left out here to request a picture, we have a `Zebraify` Android app that looks like in what we see in figure 15.5. Well done![16]

We should note that we end up with a full version of PyTorch with all ops on Android. This will, in general, also include operations you will not need for a given task, leading to the question of whether we could save some space by leaving them out. It turns out that starting with PyTorch 1.4, you can build a customized version of the PyTorch library that includes only the operations you need (see https://pytorch.org/mobile/android/#custom-build).

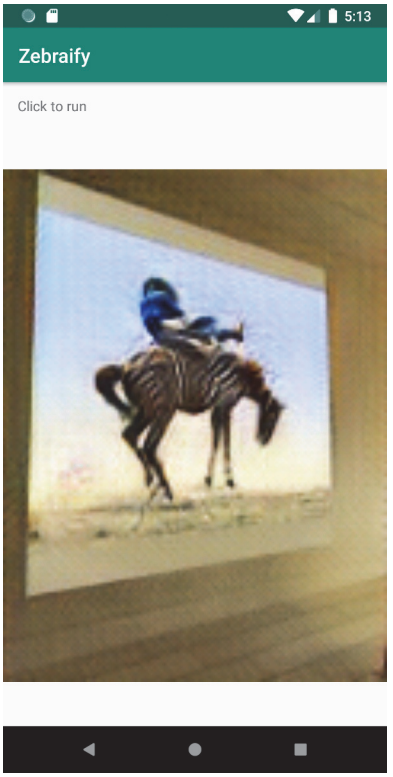


Figure 15.5 Our CycleGAN zebra app

15.5.1 Improving efficiency: Model design and quantization

If we want to explore mobile in more detail, our next step is to try to make our models faster. When we wish to reduce the memory and compute footprint of our models, the first thing to look at is streamlining the model itself: that is, computing the same or very similar mappings from inputs to outputs with fewer parameters and operations. This is often called *distillation*. The details of distillation vary—sometimes we try to shrink each weight by eliminating small or irrelevant weights;[17] in other examples, we combine several layers of a net into one (DistilBERT) or even train a fully different, simpler model to reproduce the larger model's outputs (OpenNMT's original CTranslate). We mention this because these modifications are likely to be the first step in getting models to run faster.

Another approach to is to reduce the footprint of each parameter and operation: instead of expending the usual 32-bit per parameter in the form of a float, we convert our model to work with integers (a typical choice is 8-bit). This is *quantization*.[18]

[16] At the time of writing, PyTorch Mobile is still relatively young, and you may hit rough edges. On Pytorch 1.3, the colors were off on an actual 32-bit ARM phone while working in the emulator. The reason is likely a bug in one of the computational backend functions that are only used on ARM. With PyTorch 1.4 and a newer phone (64-bit ARM), it seemed to work better.

[17] Examples include the Lottery Ticket Hypothesis and WaveRNN.

[18] In contrast to quantization, (partially) moving to 16-bit floating-point for training is usually called *reduced* or (if some bits stay 32-bit) *mixed-precision* training.

PyTorch does offer quantized tensors for this purpose. They are exposed as a set of scalar types similar to `torch.float`, `torch.double`, and `torch.long` (compare section 3.5). The most common quantized tensor scalar types are `torch.quint8` and `torch.qint8`, representing numbers as unsigned and signed 8-bit integers, respectively. PyTorch uses a separate scalar type here in order to use the dispatch mechanism we briefly looked at in section 3.11.

It might seem surprising that using 8-bit integers instead of 32-bit floating-points works at all; and typically there is a slight degradation in results, but not much. Two things seem to contribute: if we consider rounding errors as essentially random, and convolutions and linear layers as weighted averages, we may expect rounding errors to typically cancel.[19] This allows reducing the relative precision from more than 20 bits in 32-bit floating-points to the 7 bits that signed integers offer. The other thing quantization does (in contrast to training with 16-bit floating-points) is move from floating-point to fixed precision (per tensor or channel). This means the largest values are resolved to 7-bit precision, and values that are one-eighth of the largest values to only $7 - 3 = 4$ bits. But if things like L1 regularization (briefly mentioned in chapter 8) work, we might hope similar effects allow us to afford less precision to the smaller values in our weights when quantizing. In many cases, they do.

Quantization debuted with PyTorch 1.3 and is still a bit rough in terms of supported operations in PyTorch 1.4. It is rapidly maturing, though, and we recommend checking it out if you are serious about computationally efficient deployment.

15.6 Emerging technology: Enterprise serving of PyTorch models

We may ask ourselves whether all the deployment aspects discussed so far should involve as much coding as they do. Sure, it is common enough for someone to code all that. As of early 2020, while we are busy with the finishing touches to the book, we have great expectations for the near future; but at the same time, we feel that the deployment landscape will significantly change by the summer.

Currently, RedisAI (https://github.com/RedisAI/redisai-py), which one of the authors is involved with, is waiting to apply Redis goodness to our models. PyTorch has just experimentally released TorchServe (after this book is finalized, see https://pytorch.org/blog/pytorch-library-updates-new-model-serving-library/#torchserve-experimental).

Similarly, MLflow (https://mlflow.org) is building out more and more support, and Cortex (https://cortex.dev) wants us to use it to deploy models. For the more specific task of information retrieval, there also is EuclidesDB (https://euclidesdb.readthedocs.io/en/latest) to do AI-based feature databases.

Exciting times, but unfortunately, they do not sync with our writing schedule. We hope to have more to tell in the second edition (or a second book)!

---

[19] Fancy people would refer to the Central Limit Theorem here. And indeed, we must take care that the independence (in the statistical sense) of rounding errors is preserved. For example, we usually want zero (a prominent output of ReLU) to be exactly representable. Otherwise, all the zeros would be changed by the exact same quantity in rounding, leading to errors adding up rather than canceling.

15.7 Conclusion

This concludes our short tour of how to get our models out to where we want to apply them. While the ready-made Torch serving is not quite there yet as we write this, when it arrives you will likely want to export your models through the JIT—so you'll be glad we went through it here. In the meantime, you now know how to deploy your model to a network service, in a C++ application, or on mobile. We look forward to seeing what you will build!

Hopefully we've also delivered on the promise of this book: a working knowledge of deep learning basics, and a level of comfort with the PyTorch library. We hope you've enjoyed reading as much as we've enjoyed writing.[20]

15.8 Exercises

As we close out *Deep Learning with PyTorch*, we have one final exercise for you:

1. Pick a project that sounds exciting to you. Kaggle is a great place to start looking. Dive in.

You have acquired the skills and learned the tools you need to succeed. We can't wait to hear what you do next; drop us a line on the book's forum and let us know!

15.9 Summary

- We can serve PyTorch models by wrapping them in a Python web server framework such as Flask.
- By using JITed models, we can avoid the GIL even when calling them from Python, which is a good idea for serving.
- Request batching and asynchronous processing helps use resources efficiently, in particular when inference is on the GPU.
- To export models beyond PyTorch, ONNX is a great format. ONNX Runtime provides a backend for many purposes, including the Raspberry Pi.
- The JIT allows you to export and run arbitrary PyTorch code in C++ or on mobile with little effort.
- Tracing is the easiest way to get JITed models; you might need to use scripting for some particularly dynamic parts.
- There also is good support for C++ (and an increasing number of other languages) for running models both JITed and natively.
- PyTorch Mobile lets us easily integrate JITed models into Android or iOS apps.
- For mobile deployments, we want to streamline the model architecture and quantize models if possible.
- A few deployment frameworks are emerging, but a standard isn't quite visible yet.

[20] More, actually; writing books is hard!

index

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

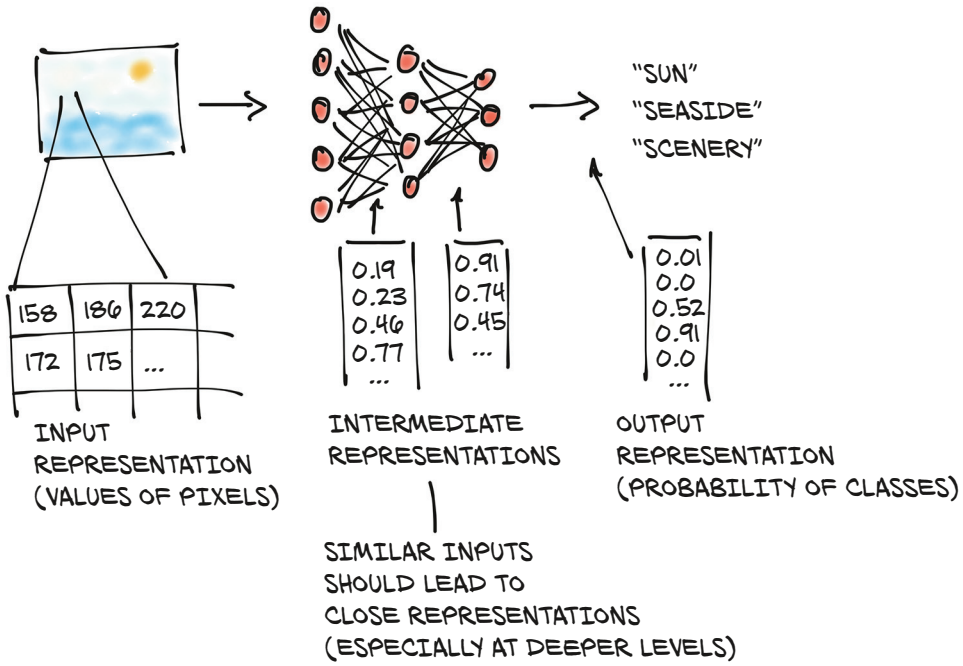
V

W

X

Y

Z


"SUN"
"SEASIDE"
"SCENERY"
158 186 220
172 175 ...
0.19
0.23
0.46
0.77
...
0.91
0.74
0.45
...
0.01
0.0
0.52
0.91
0.0
...
INPUT REPRESENTATION (VALUES OF PIXELS)
INTERMEDIATE REPRESENTATIONS
OUTPUT REPRESENTATION (PROBABILITY OF CLASSES)
SIMILAR INPUTS SHOULD LEAD TO CLOSE REPRESENTATIONS (ESPECIALLY AT DEEPER LEVELS)

PYTHON/DATA SCIENCE

Deep Learning with PyTorch

Stevens • Antiga • Viehmann

Although many deep learning tools use Python, the PyTorch library is truly Pythonic. Instantly familiar to anyone who knows PyData tools like NumPy and scikit-learn, PyTorch simplifies deep learning without sacrificing advanced features. It's excellent for building quick models, and it scales smoothly from laptop to enterprise. Because companies like Apple, Facebook, and JPMorgan Chase rely on PyTorch, it's a great skill to have as you expand your career options.

Deep Learning with PyTorch teaches you to create neural networks and deep learning systems with PyTorch. This practical book quickly gets you to work building a real-world example from scratch: a tumor image classifier. Along the way, it covers best practices for the entire DL pipeline, including the PyTorch Tensor API, loading data in Python, monitoring training, and visualizing results.

What's Inside

- Training deep neural networks
- Implementing modules and loss functions
- Utilizing pretrained models from PyTorch Hub
- Exploring code samples in Jupyter Notebooks

For Python programmers with an interest in machine learning.

Eli Stevens had roles from software engineer to CTO, and is currently working on machine learning in the self-driving-car industry. **Luca Antiga** is cofounder of an AI engineering company and an AI tech startup, as well as a former PyTorch contributor. **Thomas Viehmann** is a PyTorch core developer and machine learning trainer and consultant.

"With this publication, we finally have a definitive treatise on PyTorch. It covers the basics and abstractions in great detail."

—From the Foreword by Soumith Chintala, Cocreator of PyTorch

"Deep learning divided into digestible chunks with code samples that build up logically."

—Mathieu Zhang, NVIDIA

"Timely, practical, and thorough. Don't put it on your bookshelf, but next to your laptop."

—Philippe Van Bergen
P² Consulting

"*Deep Learning with PyTorch* offers a very pragmatic overview of deep learning . . . It is a didactical resource."

—Orlando Alejo Méndez Morales
Experian

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
www.manning.com/books/deep-learning-with-pytorch

MANNING $49.99 / Can $65.99 [INCLUDING eBOOK]

ISBN: 978-1-61729-526-3