Early detection of lung cancer

Name:	**Samia Kiran**
Subject:	**Deep Learning**
School:	**SEECS, NUST**

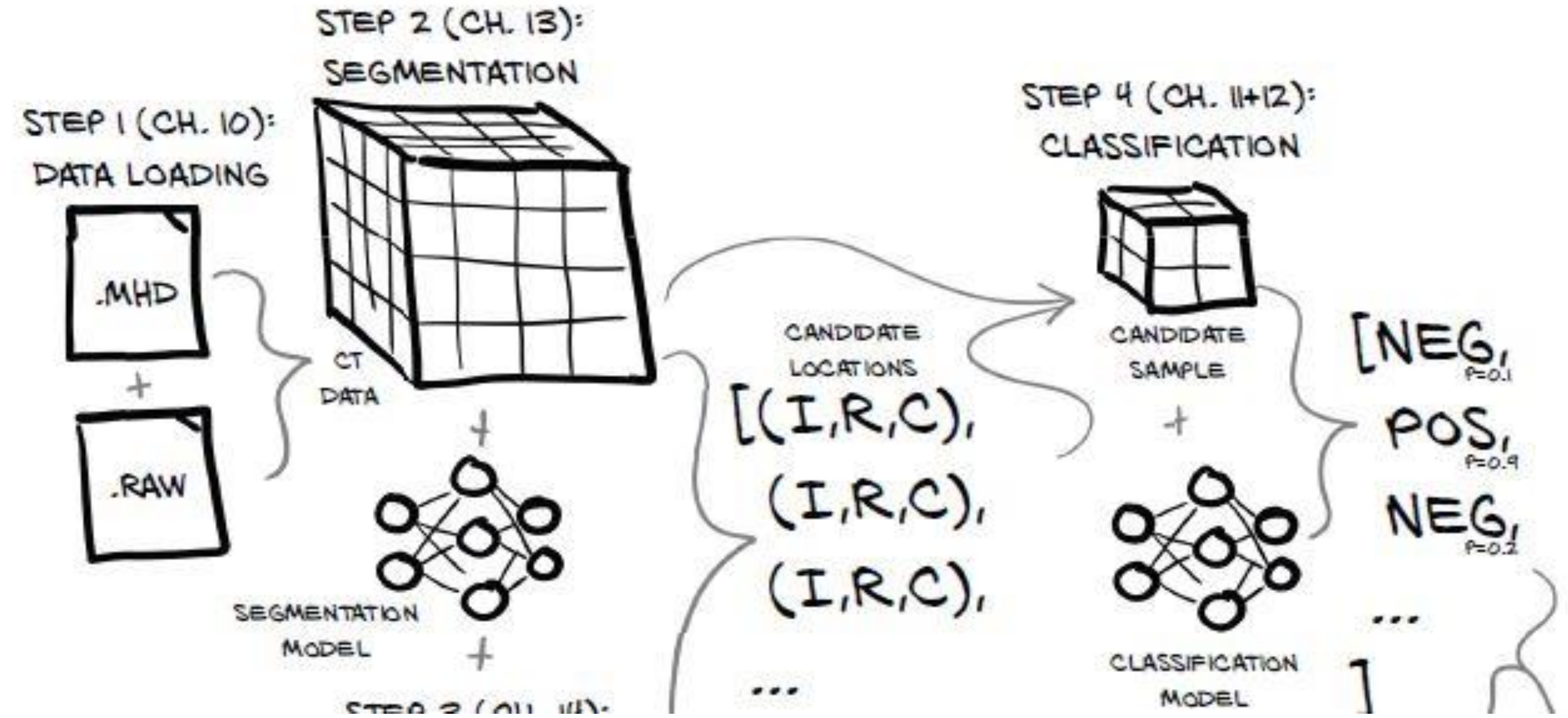
Problem Statement:

- Finding the cancerous tissues in lungs given a 3D CT Scan.
- The CT Scan contains both healthy and unhealthy tissues.

Some Terminologies:

- **CT Scans** are essentially 3D X-rays, represented as a 3D array single-channel data.
- **Voxel** represents a value on a regular grid in three-dimensional space. As with pixels in 2 dimensions.
- **Nodule** is a small mass in the lung, can turn out to be benign or a malignant tumor (also referred to as cancer).

End-to-End Process:



Loading - Data (CH. 10)

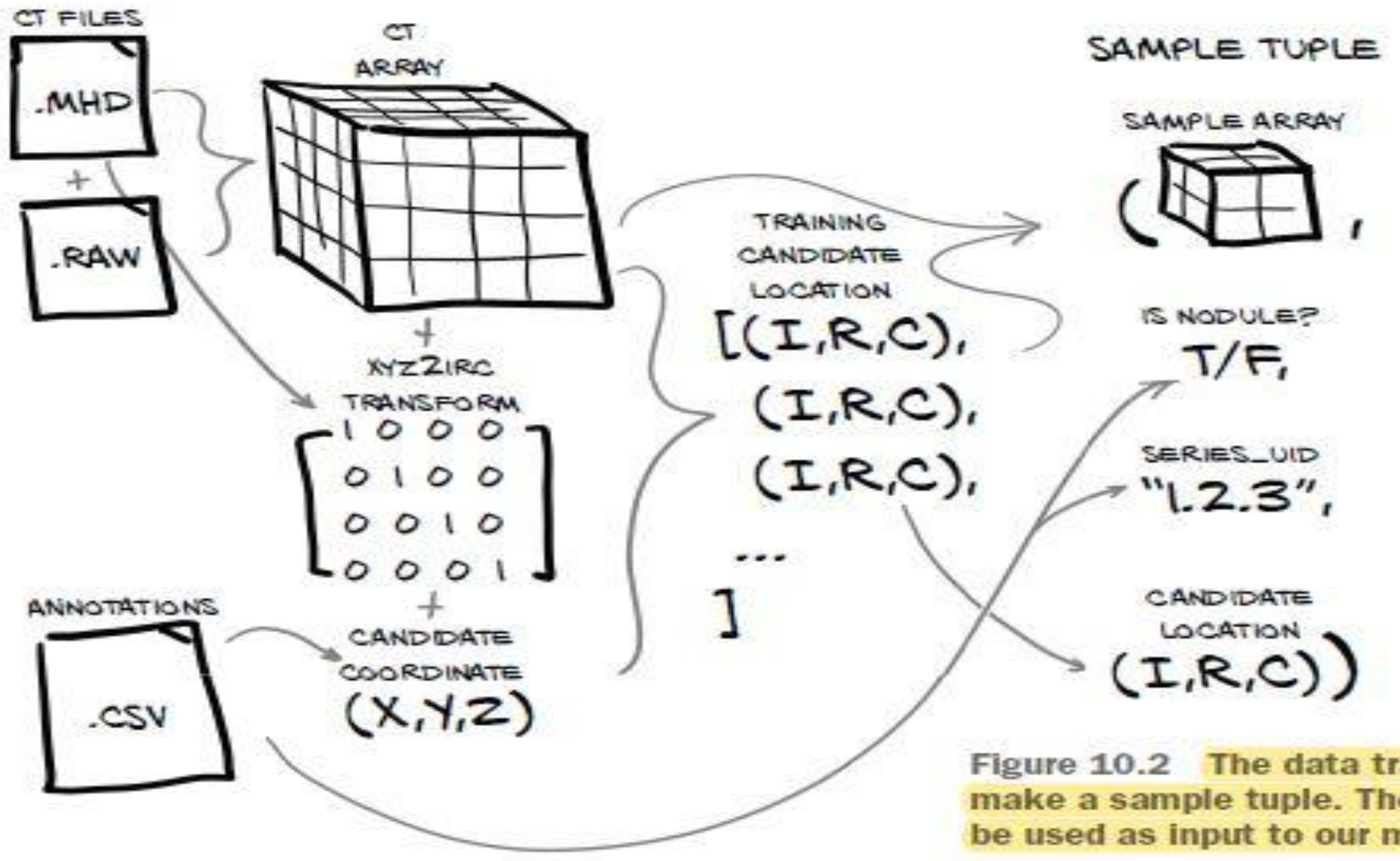


Figure 10.2 The data transforms required to make a sample tuple. These sample tuples will be used as input to our model training routine.

Terminologies (CH. 10)

- **.mhd** file contains metadata header information.
- **.raw** file contains the raw bytes that make up the 3D array.
- **series UID** uniquely identify each CT Scan Image.
- **candidates.csv** file contains information about all lumps that potentially look like nodules.
- **annotations.csv** file contains information about some of the candidates that have been flagged as nodules.
- **(X,Y,Z)** is a millimeter-based coordinate system.
- **(I,R,C)** voxel-address-based coordinate system.

Candidate Info Tuple:

- **Nodule status** (what we’re going to be training the model to classify).
- **Diameter** (useful for getting the size of nodule).
- **Series** (to locate the correct CT scan).
- **Candidate** center (to find the candidate in the larger CT).

Why xyz-irc?

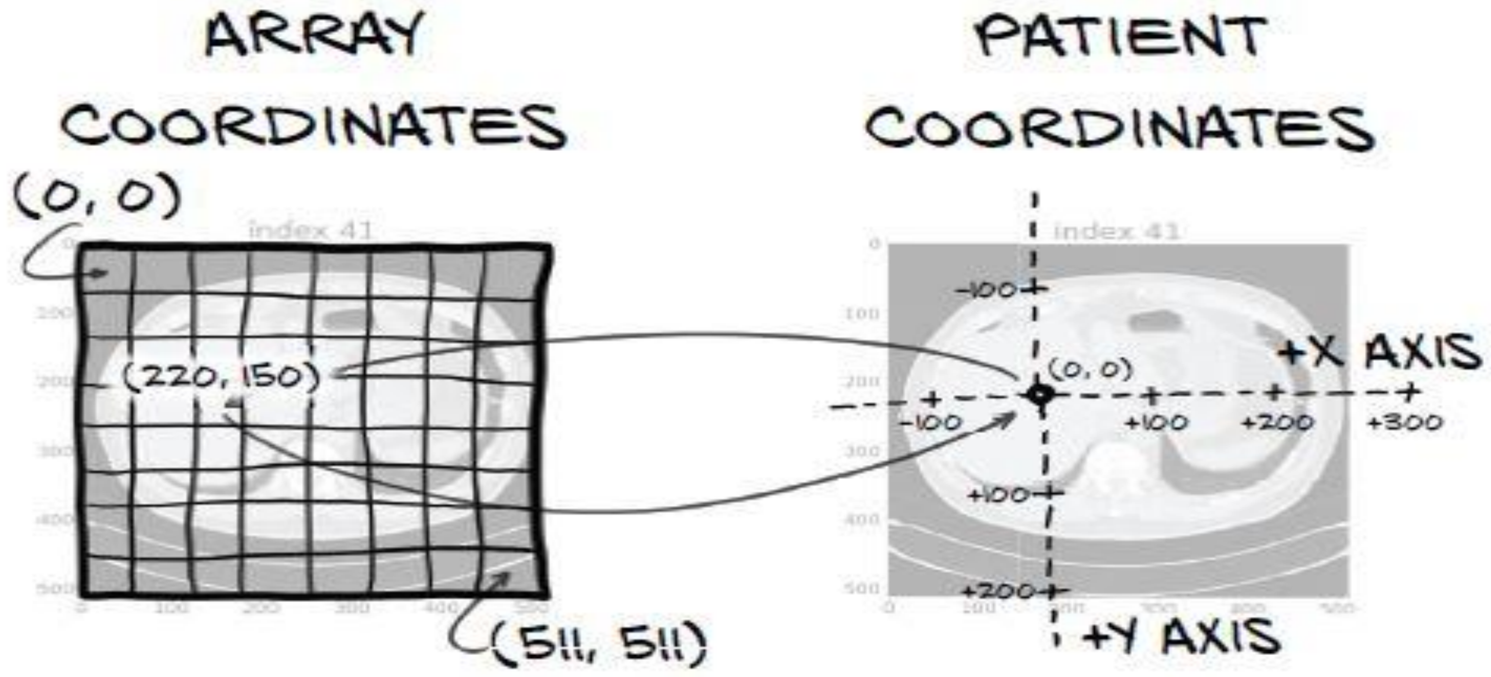
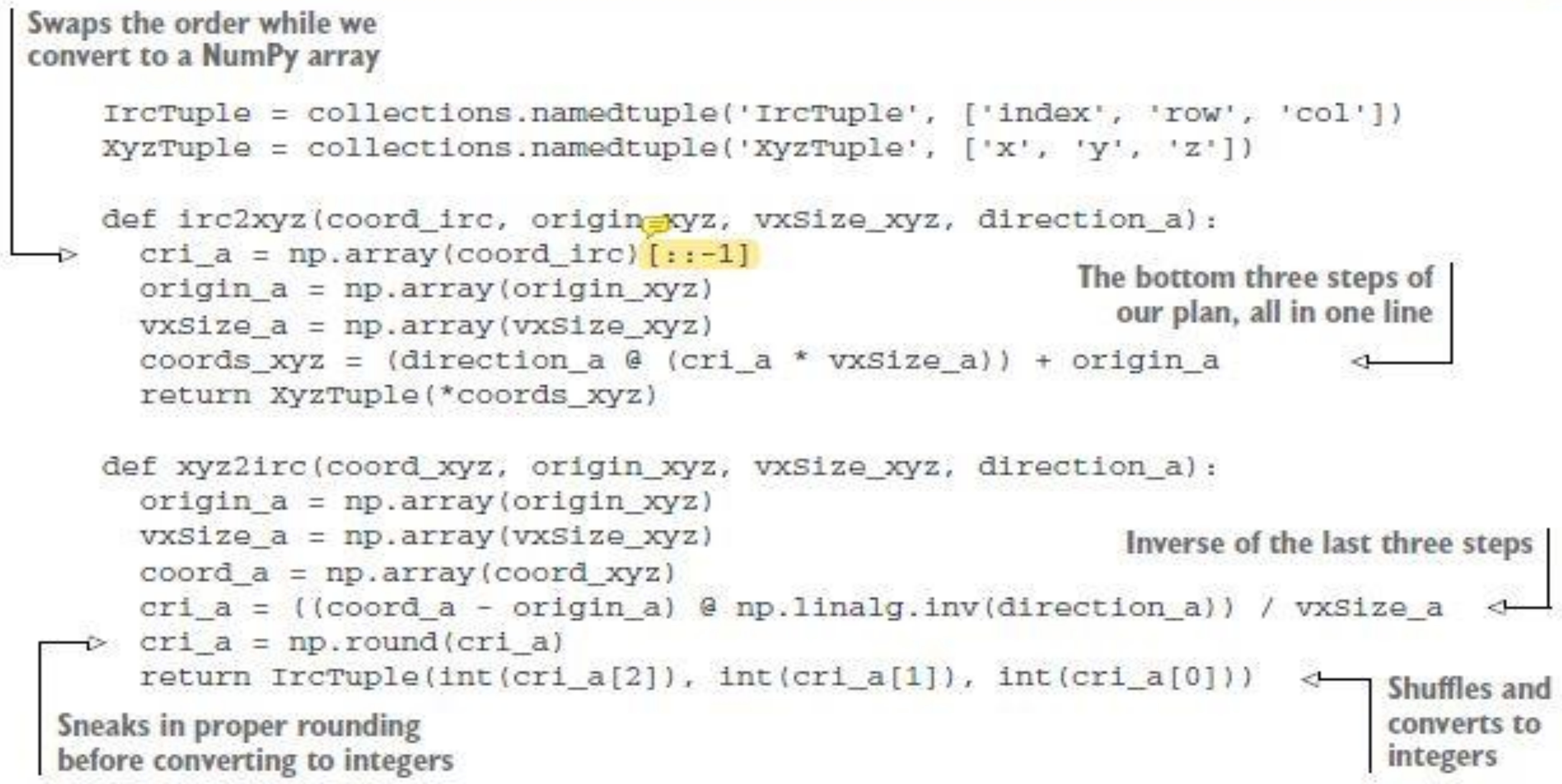


Figure 10.7 Array coordinates and patient coordinates have different origins and scaling.

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

xyz-irc Conversion:



Swaps the order while we convert to a NumPy array

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

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

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

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

Inverse of the last three steps

Sneaks in proper rounding before converting to integers

Shuffles and converts to integers

Crop a nodule from CT scan:

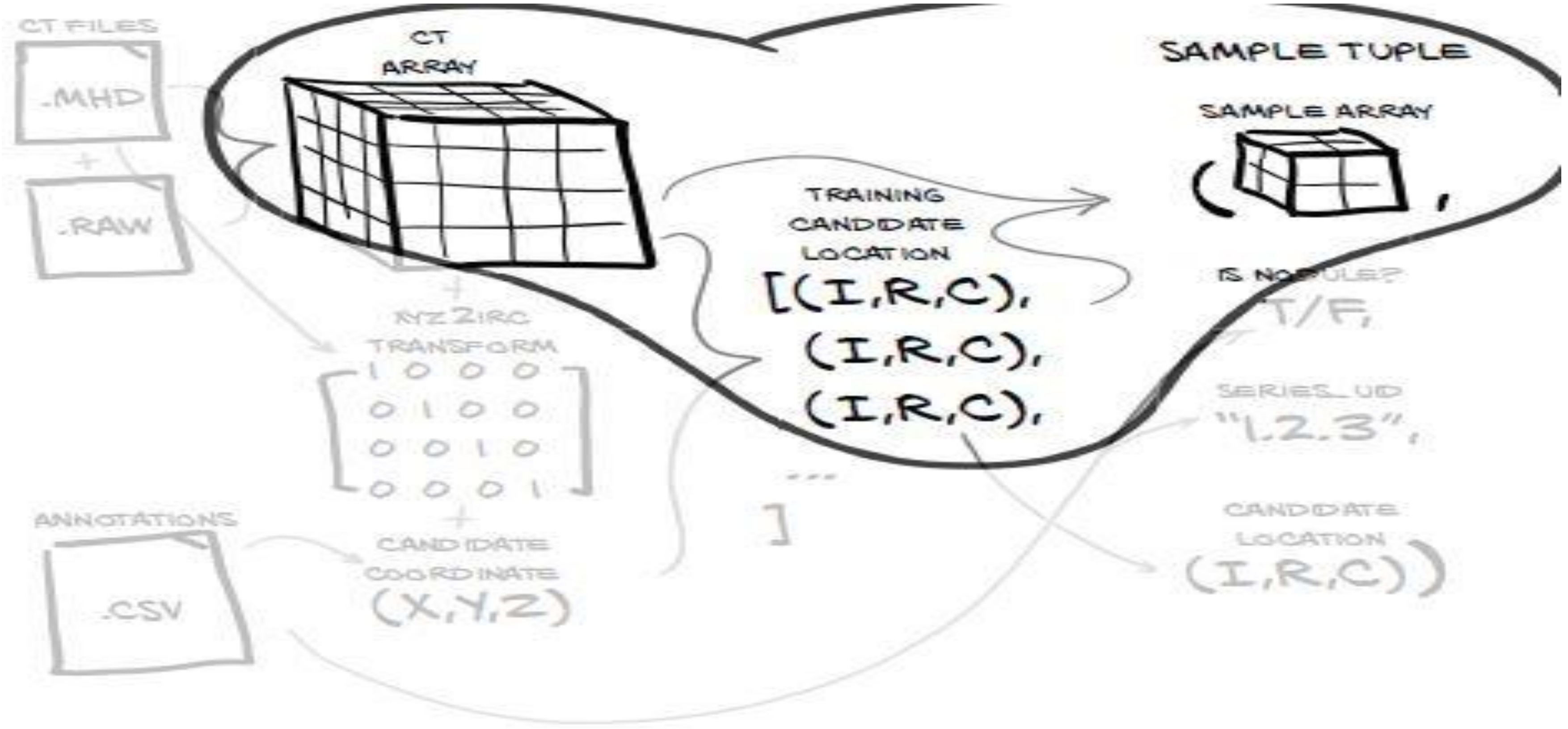


Figure 10.9 Cropping a candidate sample out of the larger CT voxel array using the candidate center's array coordinate information (Index,Row,Column)

Dataset Implementation:

- An implementation of **__len__** that must return a single, constant value after initialization.
- The **__getitem__** method, which takes an index and returns a tuple with sample.

Model Architecture (CH. 11)

- Using PyTorch DataLoaders to load data.
- Implementing a model that performs classification on our CT data.
- Setting up the basic skeleton for our application.

Model Architecture (CH. 11)

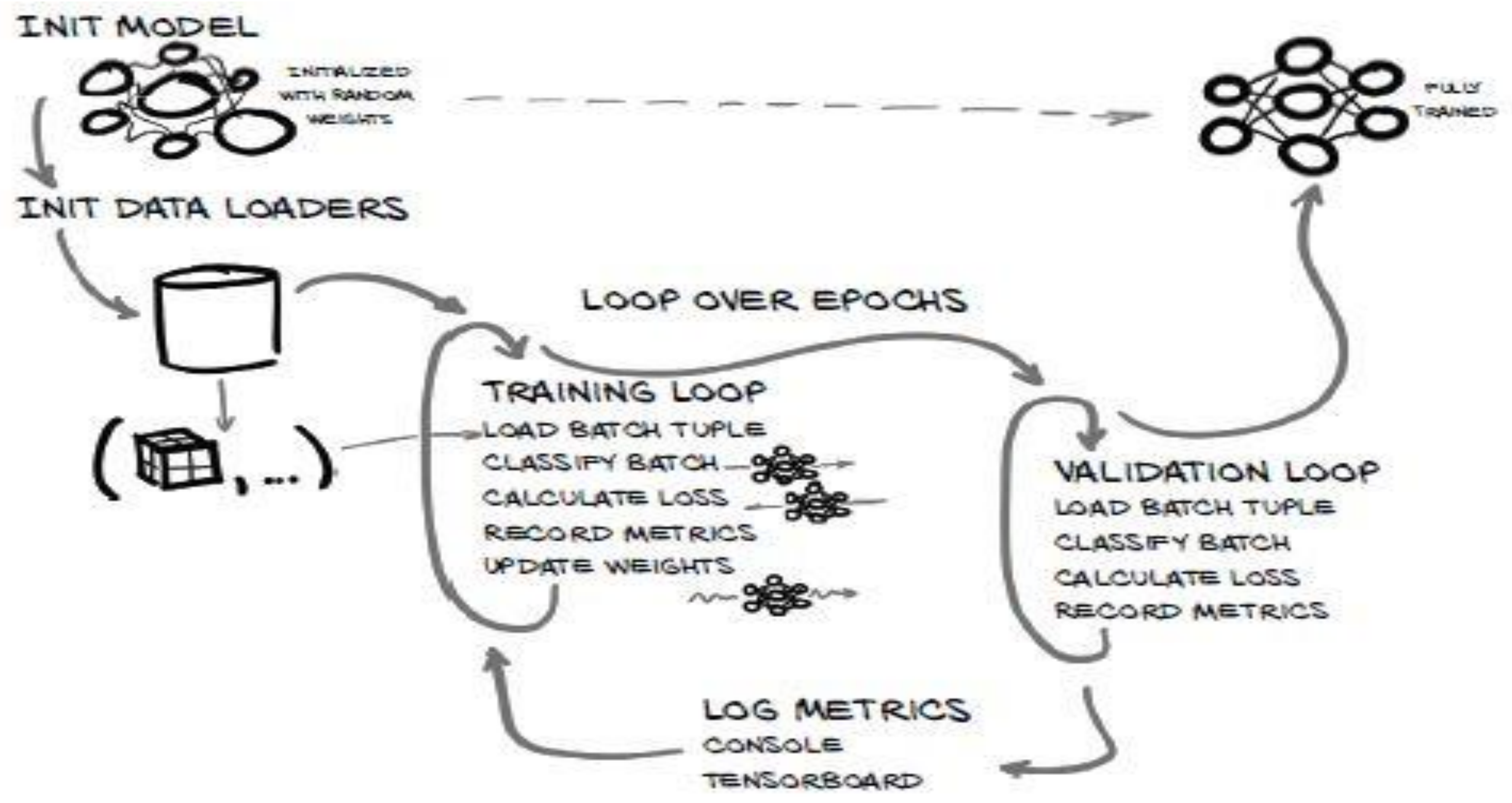


Figure 11.2 The training and validation script we will implement in this chapter

Input from Data Loaders:

- Group sample tuples together into a batch tuple.
- Allowing multiple samples to be processed at the same time.

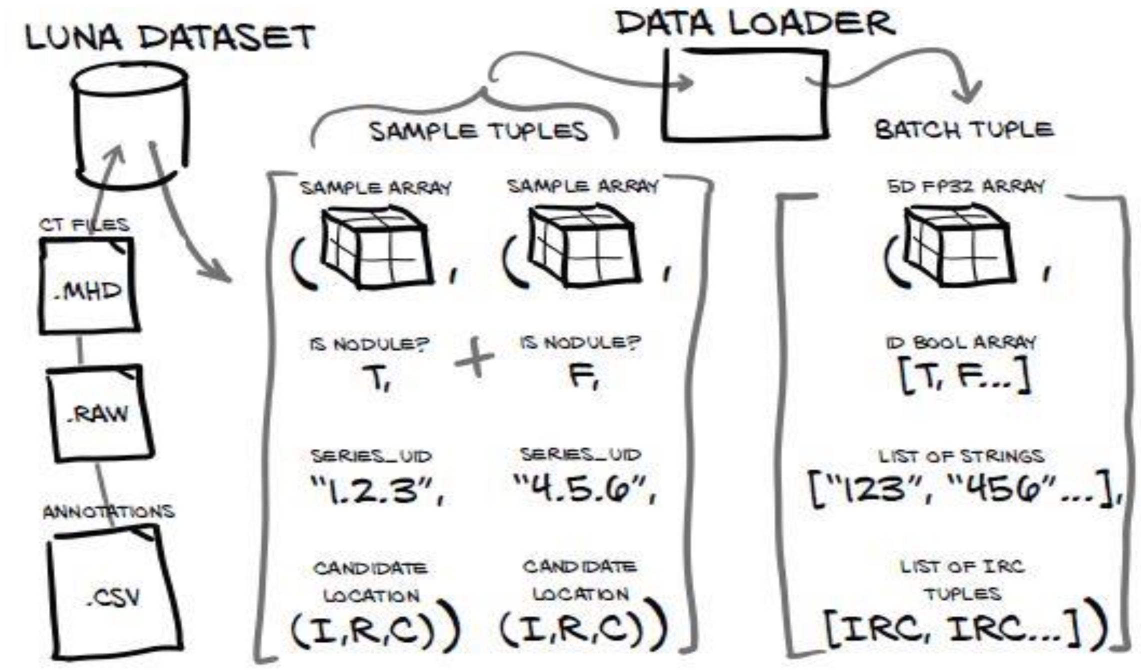
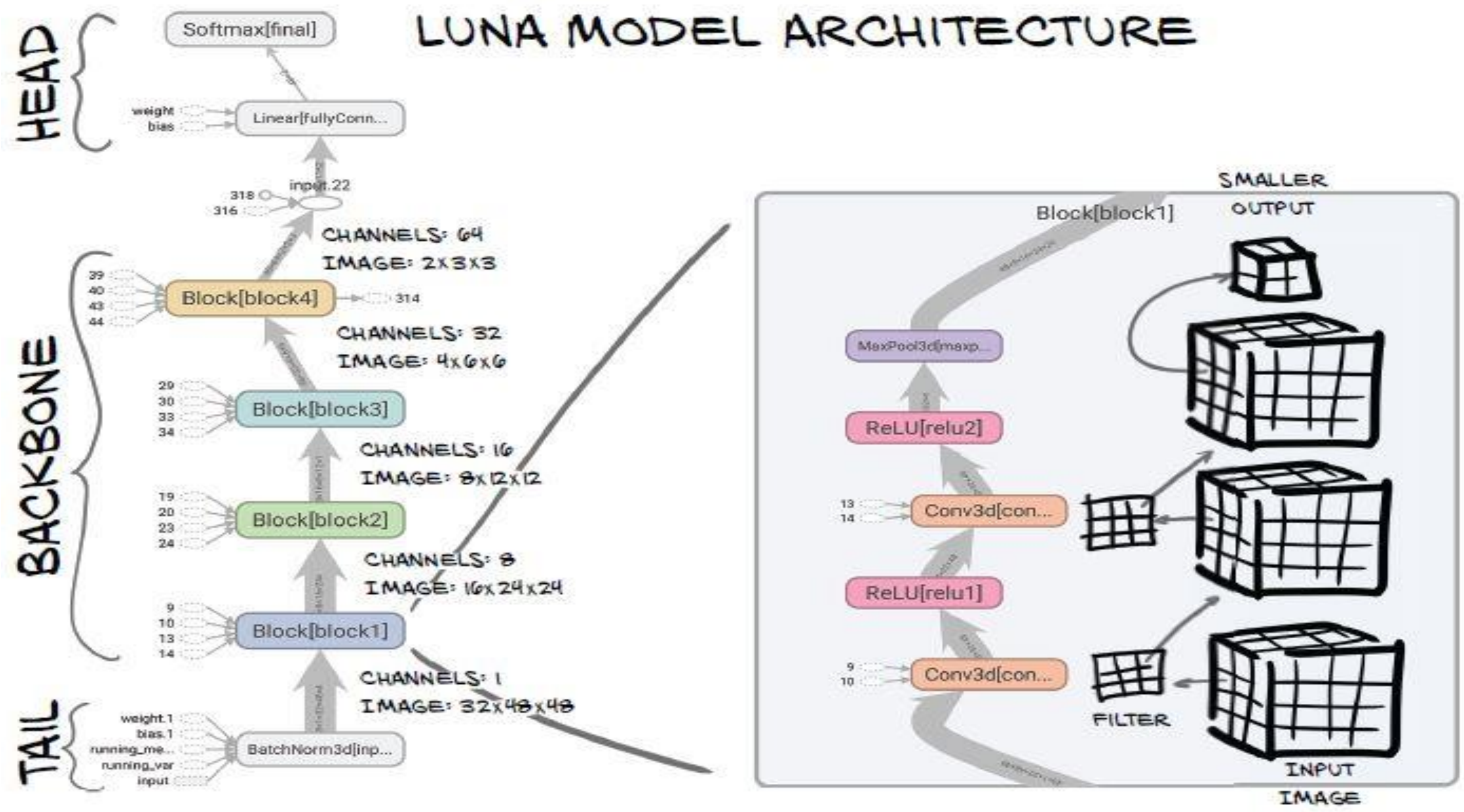


Figure 11.4 Sample tuples being collated into a single batch tuple inside a data loader

Model:



LUNA MODEL ARCHITECTURE
HEAD
BACKBONE
TAIL
Softmax[final]
weight
bias
Linear[fullyConn...
318
316
input.22
CHANNELS: 64
IMAGE: 2x3x3
39
40
43
44
Block[block4]
314
CHANNELS: 32
IMAGE: 4x6x6
29
30
33
34
Block[block3]
CHANNELS: 16
IMAGE: 8x12x12
19
20
23
24
Block[block2]
CHANNELS: 8
IMAGE: 16x24x24
9
10
13
14
Block[block1]
CHANNELS: 1
IMAGE: 32x48x48
weight.1
bias.1
running_me...
running_var
input
BatchNorm3d[inp...
Block[block1]
SMALLER OUTPUT
MaxPool3d[maxp...
ReLU[relu2]
13
14
Conv3d[con...
ReLU[relu1]
9
10
Conv3d[con...
FILTER
INPUT IMAGE

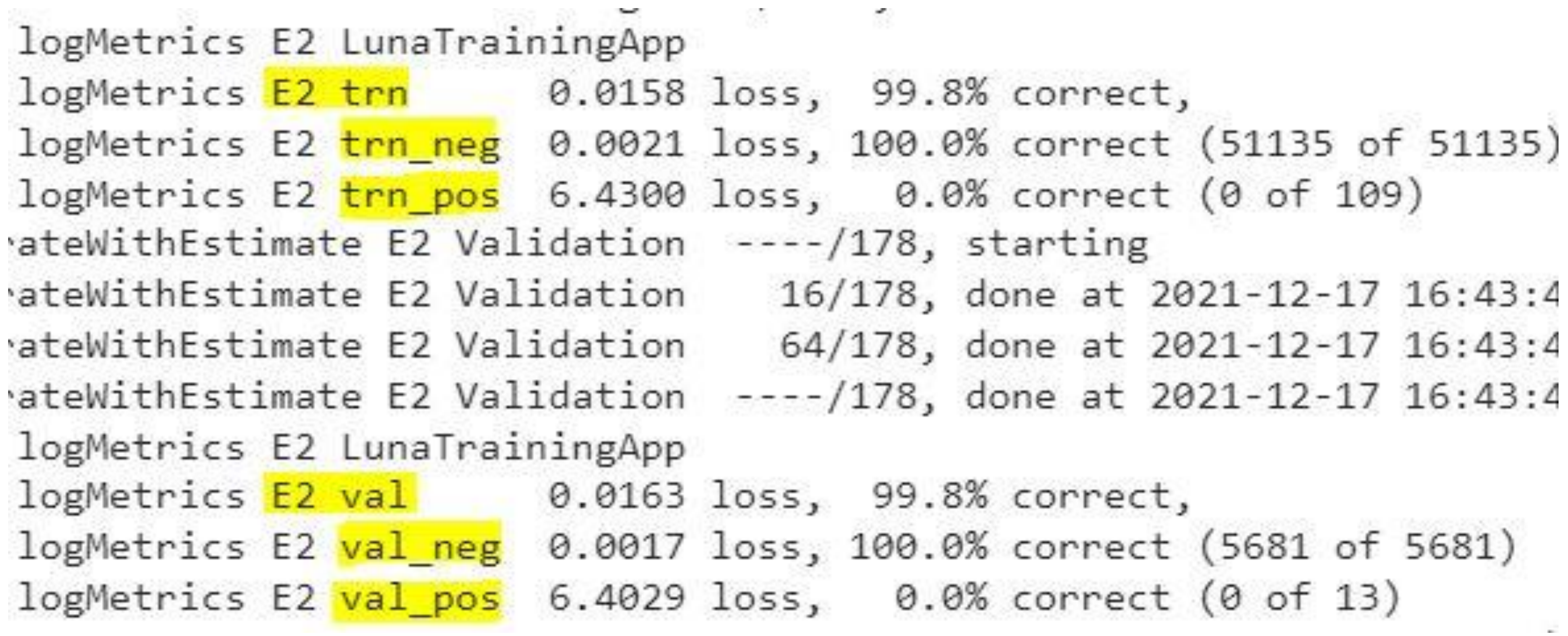
computeBatchLoss function:

- nn.CrossEntropyLoss()
- We get a tensor of loss values, one per sample.
- Lets us track the individual losses, which means we can aggregate them as we wish (per class, for example).
- We'll return the mean of those per-sample losses, which is equivalent to the batch loss.

Log metrics:

- We compute the average loss over the entire epoch.
- We limit the loss averaging to only those samples with a negative label.
- We do the same with the positive loss.
- We determine the fraction of samples we classified correctly.

Log metrics (Results):



```
 logMetrics E2 LunaTrainingApp
 logMetrics E2 trn       0.0158 loss,  99.8% correct,
 logMetrics E2 trn_neg   0.0021 loss, 100.0% correct (51135 of 51135)
 logMetrics E2 trn_pos   6.4300 loss,   0.0% correct (0 of 109)
ateWithEstimate E2 Validation  ----/178, starting
ateWithEstimate E2 Validation    16/178, done at 2021-12-17 16:43:4
ateWithEstimate E2 Validation    64/178, done at 2021-12-17 16:43:4
ateWithEstimate E2 Validation  ----/178, done at 2021-12-17 16:43:4
 logMetrics E2 LunaTrainingApp
 logMetrics E2 val       0.0163 loss,  99.8% correct,
 logMetrics E2 val_neg   0.0017 loss, 100.0% correct (5681 of 5681)
 logMetrics E2 val_pos   6.4029 loss,   0.0% correct (0 of 13)
```

Evaluating the model:

- On the validation set, we’re getting non-nodules 100% correct, but the actual nodules are 100% wrong.
- The network is just classifying everything as not-a-nodule!
- After 10 epochs, It’s interesting that we’re starting to see some decrease in the val_pos loss, however, while not seeing a corresponding increase in the val_neg loss.
- This implies that the network is learning something. Unfortunately, it’s learning very, very slowly.

How do we tackle this?

- Improve the metrics we’re using to track our progress.
- Balancing Dataset.

Augmentation (CH. 12)

- 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.

Precision/Recall and F1 Score:

- **False Positive** is when an actually uninteresting candidate is flagged as a nodule.
- **True Positive** are nodules of interest that are classified correctly.
- **False Negative** is when a nodule (that is, a potential cancer) goes undetected.
- **True Negative** are uninteresting nodules that are correctly identified as such.

Trade off btw Precision/Recall:

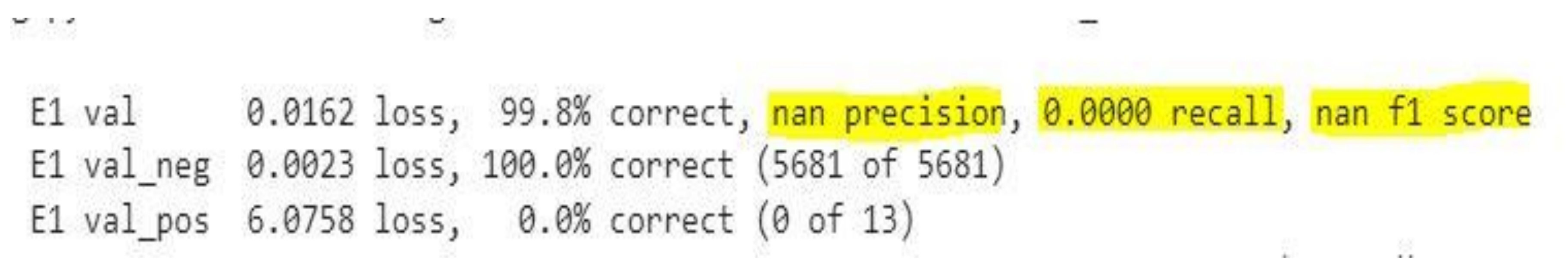
- **Precision** is the ratio of the true positives to the union of true positives and false positives.
- **Recall** is the ratio of the true positives to the union of true positives and false negatives.
- If either of them drops to zero, it’s likely that our model has started to behave in a degenerate manner.

F1 Score:

- Neither precision nor recall entirely captures what we need in order to be able to evaluate a model.
- We need something that combines both of those values.
- F1 score ranges between 0 (a classifier with no real-world predictive power) and 1 (a classifier that has perfect predictions)

Results with F1 Score:

- 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 F1 score NaN



```
E1 val      0.0162 loss,  99.8% correct, nan precision, 0.0000 recall, nan f1 score
E1 val_neg  0.0023 loss, 100.0% correct (5681 of 5681)
E1 val_pos  6.0758 loss,   0.0% correct (0 of 13)
```

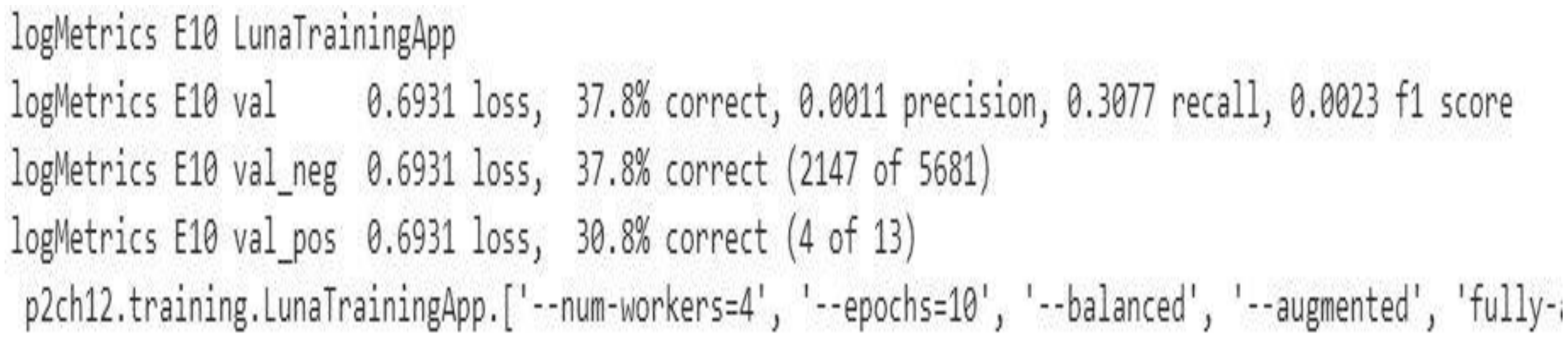
Still no Improvement?

- The Dataset Is Crushingly Imbalanced!!!
- There’s a 400:1 ratio of positive samples to negative ones.

Balancing dataset (Augmentation):

- 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.
- **getCtAugmentedCandidate()** is responsible for taking our standard chunk-of-CT-with-candidate-inside and modifying it.

Results After Augmentation:



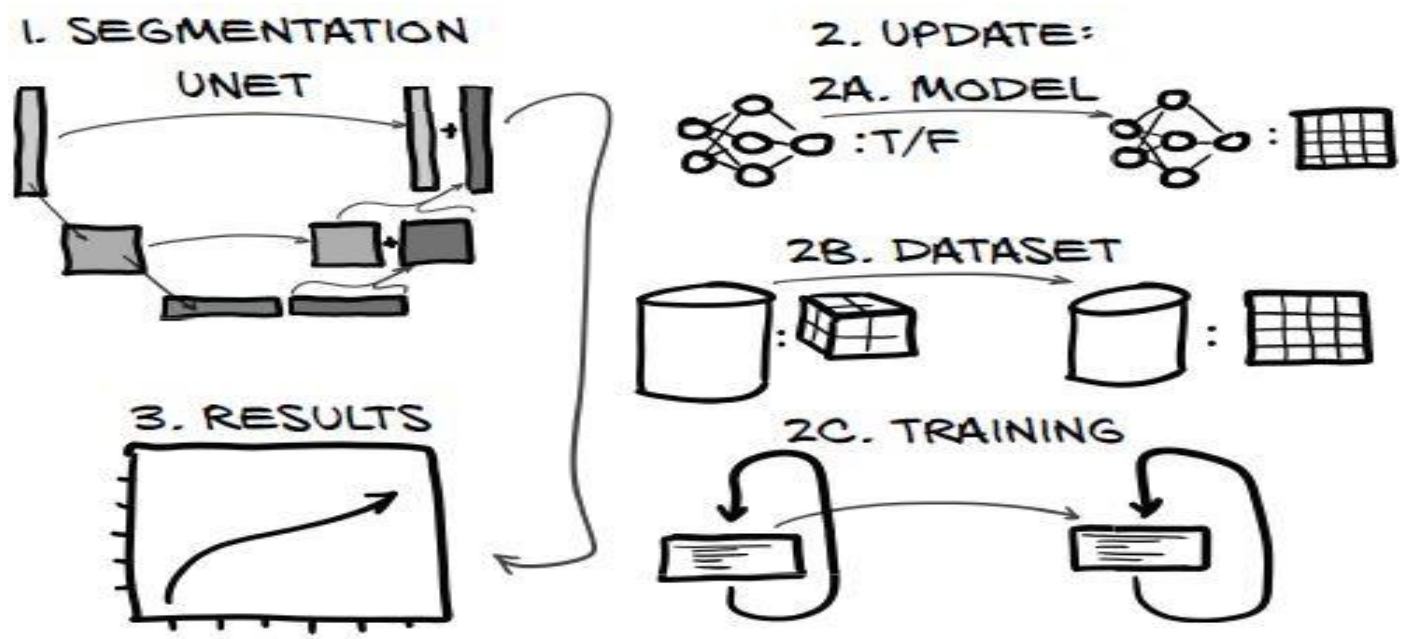
```
logMetrics E10 LunaTrainingApp
logMetrics E10 val      0.6931 loss,  37.8% correct, 0.0011 precision, 0.3077 recall, 0.0023 f1 score
logMetrics E10 val_neg  0.6931 loss,  37.8% correct (2147 of 5681)
logMetrics E10 val_pos  0.6931 loss,  30.8% correct (4 of 13)
 p2ch12.training.LunaTrainingApp.['--num-workers=4', '--epochs=10', '--balanced', '--augmented', 'fully-
```

Segmentation (CH. 13)

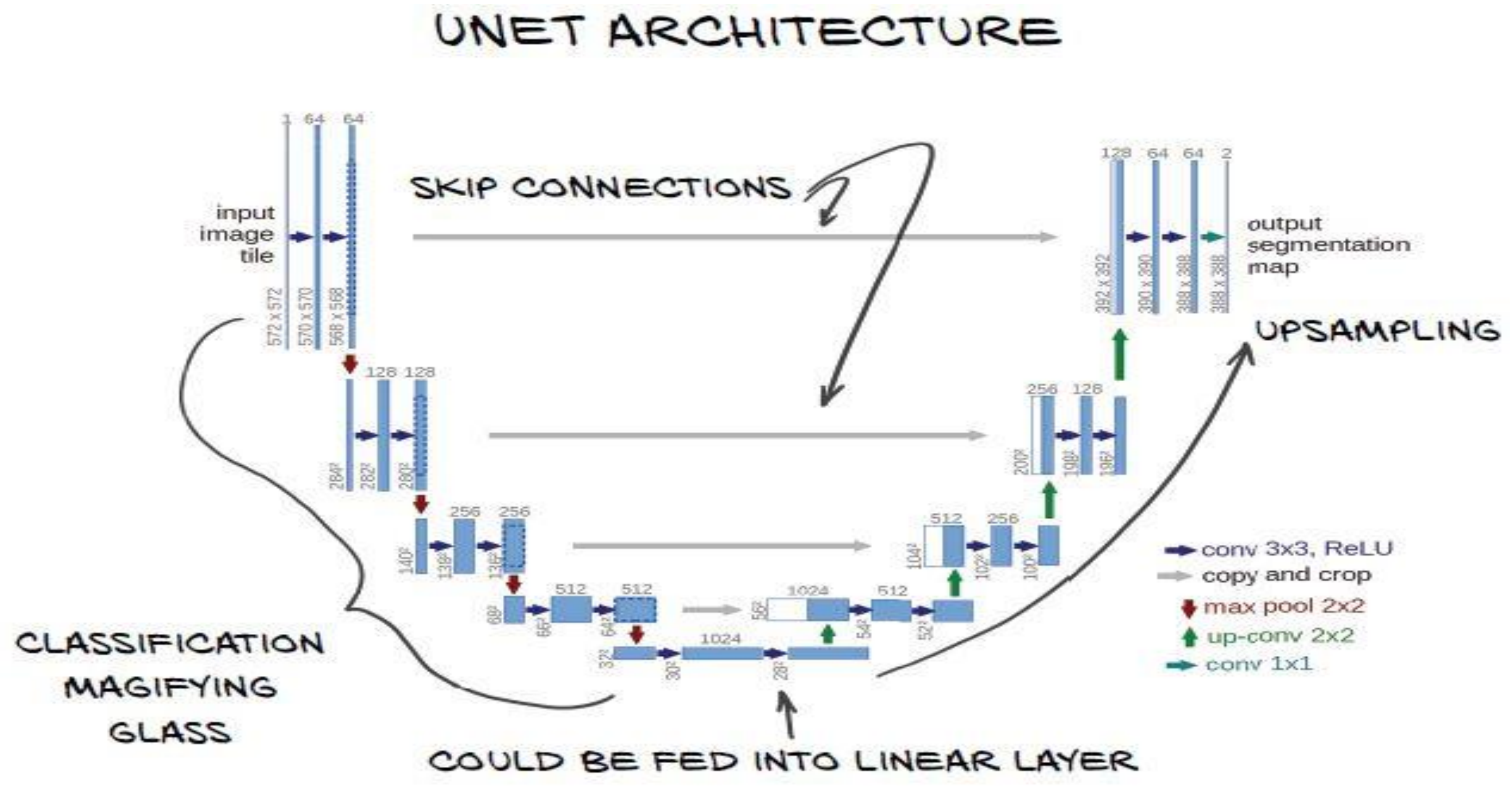
- Segmenting data with a pixel-to-pixel model.
- Performing segmentation with U-Net.
- Understanding mask prediction using Dice loss.
- Evaluating a segmentation model’s performance.

Segmentation (CH. 13)

- Segmentation flags individual pixels or voxels for membership in a class.
- This takes the form of a label mask or heatmap that identifies nodule candidates.



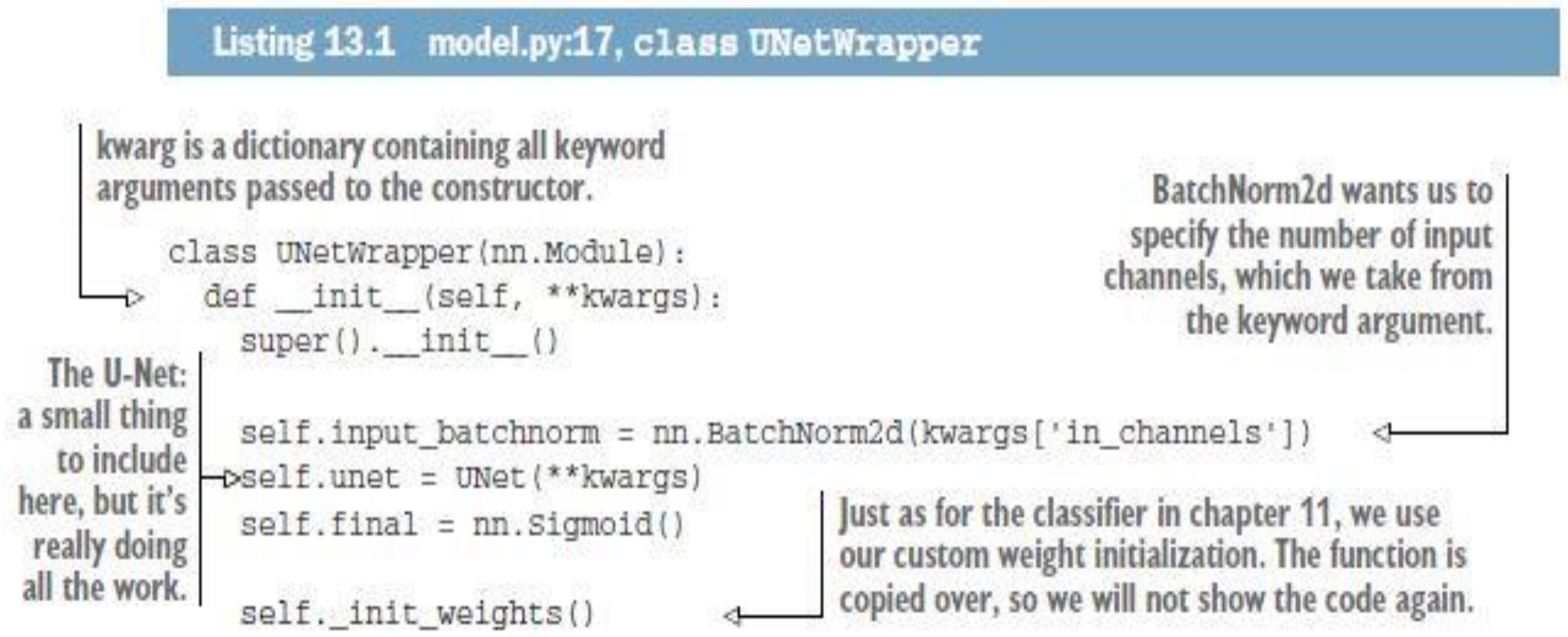
U-Net (For Segmentation):



Update Model:

- First, we’re going to pass the input through batch normalization.
- We are going to pass the output through an nn.Sigmoid layer to restrict the output to the range [0, 1].
- 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.

Update Model:



Listing 13.1 model.py:17, class UNetWrapper

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

```
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()
```

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.

Update Dataset (Bounding Box):

- Trace outward from center point in all three dimensions until we hit low-density voxels, indicating that we've reached normal lung tissue.

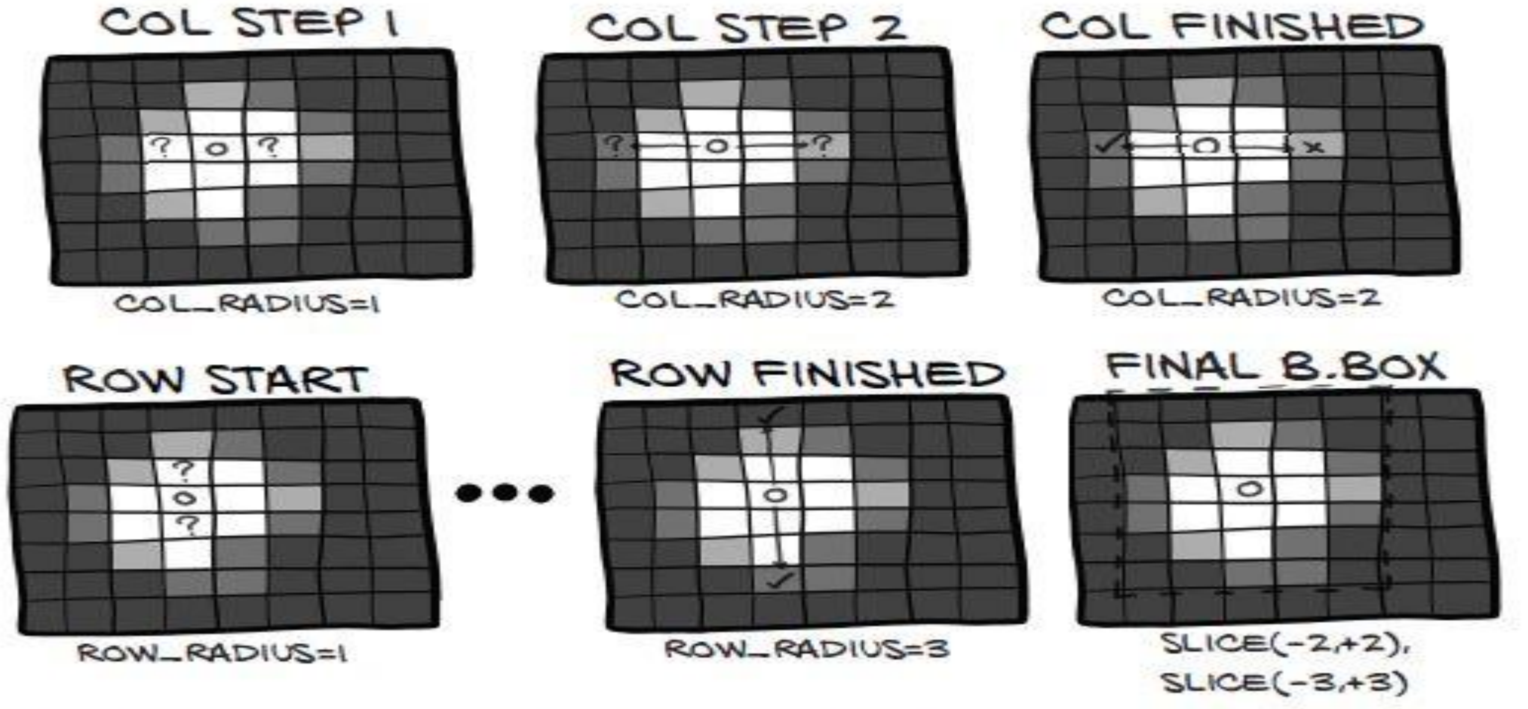
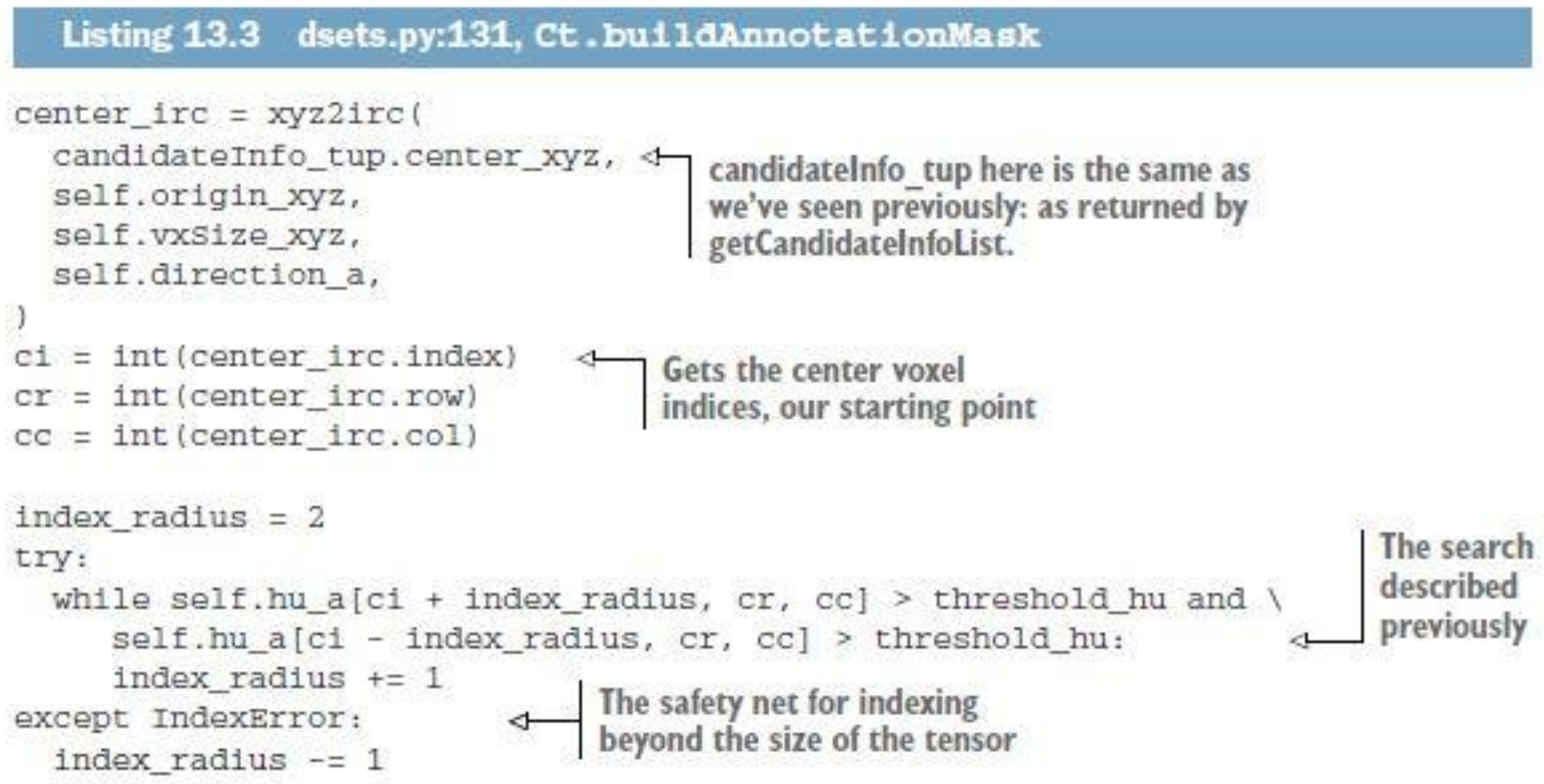


Figure 13.10 An algorithm for finding a bounding box around a lung nodule

Update Dataset (Bounding Box):



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

Update Dataset :

- The data that we produce will be two-dimensional CT slices with multiple channels.
- The extra channels will hold adjacent slices of CT.
- Each slice of CT scan can be thought of as a 2D grayscale image.

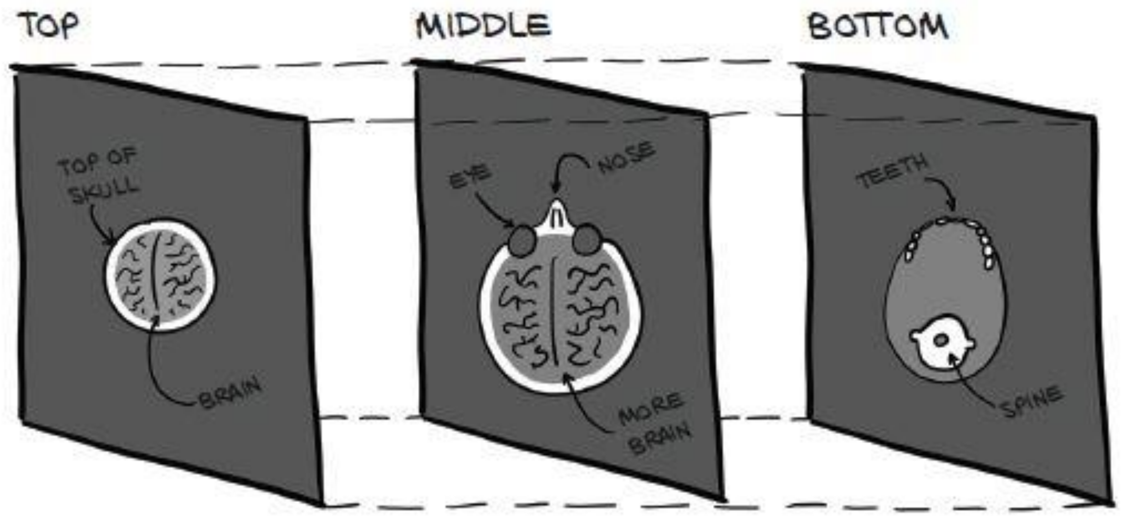
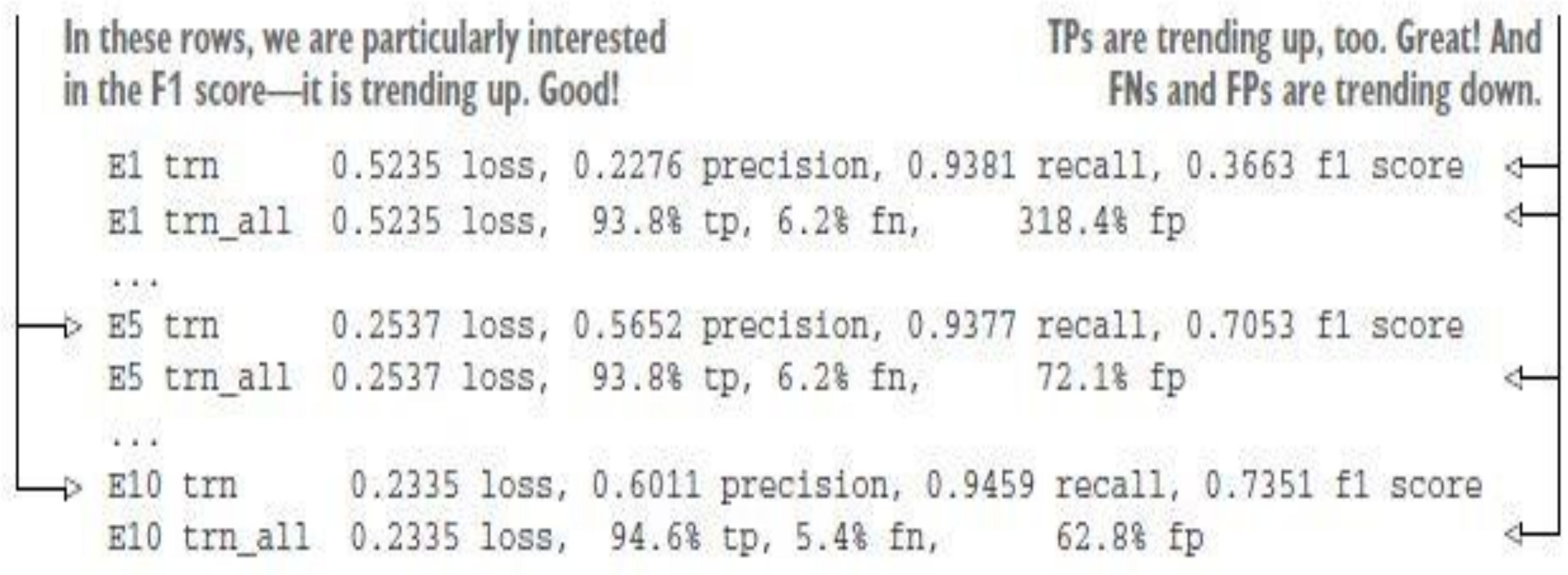


Figure 13.12 Each slice of a CT scan represents a different position in space.

Update Training Script:

- We are using the **UNetWrapper** class and giving it our configuration parameters.
- Now we have a second model for augmentation followed by segmentation.
- Using the Adam optimizer.
- 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.
- We’ll keep track of the best score we’ve seen so far in this training run before saving the model.

Results After Segmentation:



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
```

Takeaways:

- Caching can be useful if the parsing and loading routines are expensive.
- Splitting the data into a sensible training set and a validation set requires that we make sure no sample is in both sets.
- Data visualization is important; being able to investigate data visually can provide important clues about errors.
- The choice of metrics that we monitor during training is important. It is easy to accidentally pick metrics that are misleading about how the model is performing.

Takeaways:

- Balancing the training set to have an equal number of positive and negative samples during training can result in the model performing better.
- Data Augmentation helps cure overfitting when you have small data.
- It is possible to train a segmentation model on image crops while validating on whole-image slices.
- Model parameters can be saved to disk and loaded back to reconstitute a model that was saved earlier.

Questions?