CHAPTER 24

From Classifying Images to Detecting Objects

An image classifier accepts an image and produces a class label. The taxonomy of labels chosen might be quite restricted. For example, an application that chooses pictures to decorate web pages might classify images into "appealing image", "unappealing image" or "do not show under any circumstances". Very rich taxonomies are also possible. For example, a standard image classification task is ImageNet, where one must classify images into 1000 classes. The classes are categories associated with the most prominent object in the image. As another example, there are a number of applications that will classify images of individual birds (Merlin, `https://merlin.allaboutbirds.org`, is a well known example). These applications are mostly capable of classifying birds to a species level.

Yet another standard image classification task is to classify images into the *scene* depicted. Examples of scenes include "bakery shop", "kitchen" and "chaparral". Scenes are important, because different sets of objects and activities are associated with different scenes (you would not expect to see "playing hockey" in a "bakery shop"; a "commode" in a "kitchen"; or a "polar bear" in "chaparral").

The taxonomy needs to be appropriate for the images you are working with. To classify an image into an object category, it needs to be reasonably clear what the image is "about". It would usually not make sense to try and map an image into an object category if there are many prominent objects in the image.

24.1 THE MECHANICS OF IMAGE CLASSIFICATION

Image classification is hard for a number of important reasons. Objects and scenes can look different when you look at them from different directions. For example, a car seen from above can look very different from a car seen from the side. Objects can appear in images at a wide range of scales and locations. For example, a single image can contain large faces (from people standing close to the camera) and small faces (from people in the background). Many objects and scenes can deform without changing their identity. For example, dishwashers can appear in many different places (or not at all) in a kitchen, without changing the identity of the scene. There are often nasty hierarchical structures to worry about. For example, chairs have legs, backs, bolts, washers, nuts, cushions, stitches (on the cushions), and so on. Similarly, a "bakery shop" is a "food shop" which is a "shop". It can be difficult to tell whether an image belongs to only one of a set of categories. For example, many but not all images containing "fish" also contain some bits of "angler". Finally, some distinctions are just very difficult. For example, birdwatchers regard it as very difficult to tell which species a juvenile gull belongs to.

Generally image classification networks are an encoder followed by a linear

classifier as in Section 20.3.4. Mostly, the area varies by what you want from the encoder. Encoders can go from being quite simple to very deep (up to thousands of convolutional layers; transformers with billions of parameters). Training can range from the very simple version of Section 20.3.4 to elaborate constructions using auxiliary data as well as the original labelled data.

There are three natural clusters in image classification procedures, mostly determined by what you want from the encoder. Small specialized image classifiers are intended to deal with particular quite constrained application problems quickly and efficiently; as long as the encoder is enough to deal with the problem at hand, you are happy. General purpose image classifiers involve elaborately trained image encoders that are slightly fine-tuned (or even left unchanged) before being applied to your problem. Large specialized image classifiers typically deal with specialized problems – for example, identifying wildlife – that might be very hard and might require careful and specialized encoder training or construction and also quite elaborate data curation regimes.

24.1.1 Small Specialized Classifiers

This is a simple and common case, which applies if there is some reason to expect the problem is relatively straightforward, or if you just can't put much effort into obtaining a solution. The recipe here is straightforward. Obtain (or build) an image encoder. Usually, you don't want to use a great deal of computation, so it is likely a good idea to use a small convolutional encoder. Building your own is informative, but tiresome, so you could use one of the many example models available (for example, a ResNet-18 from `https://docs.pytorch.org/vision/0.20/models/resnet.html`). Attach a linear classifier to this model, and train it on your dataset, following the outline of Section 20.3.4. You may find an advantage to using a pre-trained encoder. Many pre-trained encoders are available; some are listed in the resources below.

24.1.2 General Purpose Classifiers

For problems where more effort is justified, it is now usual to use pre-trained encoders. These encoders may be quite elaborate, and may be trained using quite elaborate strategies that do not rely on image labels. You then attach a linear classifier to the encoder, and train that classifier. Usually, but not always, you finetune the encoder at the same time.

The encoder that you use will likely have been at least pre-trained on the ImageNet dataset. Other possibilities, sketched here and described in somewhat greater detail in Section 26.3, include:

24.1.3 Large Specialized Classifiers: Fine Grained Category Recognition

24.2 DATASETS AND RESOURCES

- things go obsolete fairly quickly, and disappear

24.2.1 Fine-grained Datasets

The Oxford Flowers dataset consists of images of 102 types of flower occurring in the UK. You can download the dataset from `https://www.robots.ox.ac.uk/~vgg/data/flowers/102/`. The number of images per category is variable, but no category has fewer than 40 images. The Oxford-IIT pet dataset consists of about 200 images per class of 37 different breeds of dog and cat (so "beagle"; "boxer";"abyssinian"; "bengal"). These datasets are commonly used as examples of a small, hard dataset.

24.2.2 Smaller Image Categorization Datasets

MNIST is an early dataset of handwritten digits, originally constructed by Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. You can find this dataset in several places. The original dataset is at `http://yann.lecun.com/exdb/mnist/`. The version I used was prepared for a Kaggle competition (so I didn't have to decompress Lecun's original format). I found it at `http://www.kaggle.com/c/digit-recognizer`. MNIST has had tremendous influence, but is now obsolete.

Fashion-MNIST is a collection of 28x28 grayscale images, organized into 10 categories (to do with garment identity). There are 5, 000 training and 1, 000 test examples per category. You can find this dataset at `https://github.com/zalandoresearch/fashion-mnist`. Fashion-MNIST is still occasionally used.

CIFAR-10 and CIFAR-100 are datasets of 32×32 color images in 10 categories, collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. There are 5000 training images and 1000 test images per class in CIFAR-10. There are 500 training images and 100 test images per class in CIFAR-10. You can download these datasets from `https://www.cs.toronto.edu/kriz/cifar.html`. Rodrigo Benenson maintains a website giving best performance to date on these datasets at `https://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html`. The best error rate recorded there for MNIST is 0.21% (i.e. a total of 21 test examples wrong in the 10, 000 example test set). For CIFAR-10, the best error rate is 3.47% (i.e. a total of 347 test examples wrong); for CIFAR-100, 24.28%. The last records in that collection are from 2015, but there is no reason to expect major improvements have occurred. CIFAR-10 and CIFAR-100 are still occasionally used to evaluate small specialized architectures.

24.2.3 Bigger Image Categorization Datasets

SUN

The Places 2 dataset consists of some 10 million images of scenes in more than 400 categories, originally collected by Bolei Zhou, Agata Lapedriza, Aditya Khosla, Antonio Torralba and Aude Oliva of MIT. There were variants available, but a number of links have now expired. A version of the dataset is available at `https://www.kaggle.com/datasets/nickj26/places2-mit-dataset/data`.

FIGURE 24.1: *A simple penguin detector sweeps boxes across an image, passing each box to a classifier which computes a penguin score. On the* **right***, quite rough localization (the* **red** *box is not exactly on the penguin) requires scoring a very large number of boxes (the* **blue** *boxes give some sense of the number). More accurate localization might require more boxes, but too many boxes create problems with efficiency and with the classifier. Because the classifier must be able to deal with boxes that don't closely hug the penguin, it must respond to the* **green** *and* **white** *boxes, too. You do not want to count three penguins, so use non-maximum suppression to produce the* **red** *box. You can then use bounding box regression to predict an update to the original box (***center***). If you want to find larger penguins, search a smaller version of the image (***right***).* **Image credit:** *Robert Forsyth's photograph of a Cape Penguin at Boulders Beach.*

24.3 EXAMPLE APPLICATIONS

24.3.1 Predicting Scene Labels

24.3.2 Identifying NSFW Images

24.3.3 Predicting Phrases from Images

24.4 BOXOLOGY: STRAIGHTFORWARD OBJECT DETECTORS

An object detector accepts an image and produces a set of image domains with a label for each domain. These domains have often been axis aligned boxes. More recent detectors produce segmentation masks as well. To build a detector, you need to choose an object taxonomy in advance and obtain examples of the categories. Good performance currently requires many and varied examples of each category.

As in image classification, one detects object categories rather than object instances, so "faces" rather than "Inkosi Albert Luthuli's face". As in image classification, there can be some confusion about where category boundaries lie – so, for example, a "cat" detector probably should respect the difference between "cat"s

and "lion"s, but might well fudge the distinction between "cats" and "sand cats" or other small felids that look very like domestic cats. There are reasons to build detectors that observe very fine grained distinctions between categories – for example, applications that tell what species of bird or plant appear in a picture (Figure 21.3) – but one seldom needs to detect instances.

Here is a simple recipe to build an object detector. Choose a set of C categories you wish to detect, and for which you have training data. This set of categories should include "nothing" or "unknown", for convenience. Build an image classifier that can tell which category is present in an image box of fixed size. Apply the classifier to each of an appropriate set of boxes in the image. Each box now has a score for each of C categories. Postprocess the boxes to remove non-maxima (ignore the details for the moment) and to remove boxes where the largest object score is too small. Adjust the remaining boxes to improve localization, then report them and their scores. You now have a set of domains with a label for each.

Part of the difficulties in object detection stem from the same effects that make image classification hard. Objects look different when you look at them from different directions. Shading effects can significantly change what objects look like. Different instances of the same category look different. An object detector takes an image, and, for each object class it knows about, produces a list of domains each of which has a score.

There are some difficulties that are specific to detection. Your classifier will need to cope with a large number of boxes, most of which don't have an object, without loss of accuracy. You will need to tell the difference between two overlapping objects and one object which produces high scores in two boxes near each other. Sorting out where objects are in detail presents problems. If you try to localize objects very accurately by using a very large number of boxes, the detector is likely to be inefficient. If you use fewer boxes, most objects will not have a box on top of them – you will need to adjust the position of each box that is determined to contain an object. Finally, it is inefficient to chop an image into a large number of overlapping boxes and then pass each through a feature encoder then a classifier. Instead, you should pass the whole image into an encoder, then use some mechanism to extract a set of features describing each box from the encoder and steer these to a fixed linear classifier.

24.4.1 Key Principles

The recipe sketches the main issues in object detection. You must find both what is in the image and where it is. To do so, you must decide how to sample object *configuration* (at least locations and scales in the image); you must decide whether there is an object at each of the configurations you test; and you need to resolve interactions between hypotheses and clean up configuration estimates. The very simplest procedure for sampling configurations is to use all boxes of fixed size on some grid, where the fineness of the grid is determined by how accurately you want to find the object's configuration. Doing this turns out to be grossly inefficient, for three reasons that are important in shaping how detectors are built. First, classifier scores for windows with heavy overlap are correlated. Second, you can predict the location of the "best" window from a good window. And third, it is easy to check

whether windows are worth looking at.

Correlated scores: Present the classifier with an image window, record its score, then move the window over by one column of pixels and compute the score there. These two scores must be very strongly related, because the classifier sees mostly the same pixels in mostly similar locations. Furthermore, the classifier should be *trained* to produce strongly related scores, because you can never be sure that the image window will be dead on top of an object of exactly the right size. In turn, it follows that you can interpolate classifier scores quite effectively, so you don't need to compute the score at every window.

Extrapolating configuration: Assume you have a window that has a strong score from the classifier. This window should mostly overlap an object but may not be placed exactly on top of the object. It turns out that you can predict fairly accurately where the best placed window is. You should expect this, because the scores are correlated. The procedure for extrapolating is known as *bounding box regression* – details below.

Worthwhile windows: It is a remarkable fact that it is quite easy to tell whether a window might have an object in it, even without specifying what the object is. Boxes that span a boundary probably contain only part of an object. Boxes that have no boundaries nearby likely don't contain anything interesting. It is still quite difficult to actually find the boundaries of objects. Nonetheless, it has been known for some time that one can use boundaries to score boxes for their "objectness". Efficient detectors look only at boxes that have a high enough objectness score.

24.4.2 Evaluating Detectors

Evaluating object detectors takes care. The detector reports domains with labels and scores. Evaluating the detector involves comparing these domains with ground truth domains that have been marked on the image by people. The evaluation should favor detectors that get the right number of the right object in the right place. It should discourage detectors that just produce an awful lot of domains, or detectors that only produce absolutely obvious domains.

To start, assume the detector responds to only one kind of object. You now have two lists: one ($\mathcal{G}$) is the list of ground truth domains, the other ($\mathcal{D}$) is the list of domains the detector produces, which has already been subject to anything the team that created the detector can think of. You should think of the detector as a search process. The detector has searched a huge collection of domains, and produced some domains that it asserts are relevant, in order of relevance (this is the list $\mathcal{D}$). This list needs to be scored. The evaluation must mark domains in $\mathcal{D}$ with `relevant` if they match ground truth domains and `irrelevant` otherwise, and then summarize the lists.

The domains that the detector predicts are unlikely to match ground truth exactly, so you need some way of telling whether the domains are good enough. The standard method for doing this is to test the IoU of Section 21.2.1.(Intersection over Union). Write D_g for the ground truth domain and D_p for the predicted domain. The IoU is

$$\text{IoU}(D_p, D_g) = \frac{\text{Area}(D_g \cap D_p)}{\text{Area}(D_g \cup D_p)}.$$

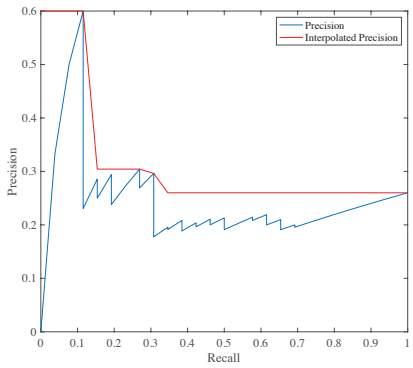

FIGURE 24.2: *Two plots for an imaginary search process. The precision plotted against recall shows a characteristic sawtooth shape. Interpolated precision measures the best precision you can get by increasing the recall, and so smoothes the plot. Interpolated precision is also a more natural representation of what one wants from search results – most people would be willing to add items to get higher precision. Interpolated precision is used to evaluate detectors.*

Choose some threshold t. If $\text{IoU}(D_p, D_g) > t$, then D_p could match the ground truth domain D_g.

The detector should be credited for producing a domain that has a high score and matches a ground truth domain. But the detetector should not be able to improve its score by predicting many domains on top of a ground truth domain. The standard way to handle the problem is to mark the overlapping domain with highest score `relevant`. The procedure is:

- Choose a threshold t.
- Order $\mathcal{D}$ by the score of each domain, and mark every element of $\mathcal{D}$ with `irrelevant`. Choose a threshold t.
- For each element of $\mathcal{D}$ in order of score, compare that domain against all ground truth domains. If any ground truth domain has $\text{IoU} > t$, mark the detector domain `relevant` and remove that ground truth domain from $\mathcal{G}$. Proceed until there are no more ground truth domains or until all elements of $\mathcal{D}$ have been compared.

Now every domain in $\mathcal{D}$ is tagged either `relevant` or `irrelevant`.

There are standard evaluations for search results like those produced by our detector. The first step is to merge the lists for each evaluation image into a single list of results, sorted by score. The *precision* of a set of search results $\mathcal{S}$ is given by

$$\mathbf{P}(\mathcal{S}) = \frac{\text{number of relevant search results}}{\text{total number of search results}}.$$

The *recall* is given by

$$\mathbf{R}(\mathcal{S}) = \frac{\text{number of relevant search results}}{\text{total number of relevant items in collection}}.$$

As you move down the list $\mathcal{D}$ in order of score, you get a new set of search results. The recall never decreases as the set gets larger, and so you could plot the precision as a function of recall (write $\mathbf{P}(\mathbf{R})$). These plots have a characteristic saw-tooth structure (Figure 24.2). If you add a single irrelevant item to the set of results, the precision will fall; if you then add a relevant item, it jumps up. The sawtooth doesn't really reflect how useful the set of results is — people are usually willing to add several items to a set of search results to improve the precision — and so it is better to use *interpolated precision*. The interpolated precision at some recall value R_0 is given by

$$\hat{\mathcal{P}}(R_0) = \max_{R \geq R_0} \mathbf{P}(R)$$

(Figure 24.2). By convention, the *average precision* is computed as

$$\frac{1}{11}\sum_{i=0}^{10} \hat{\mathcal{P}}(\frac{i}{10}).$$

This value summarizes the recall-precision curve. Notice this averages in interpolated precision at high recall. Doing so means a detector cannot get a high score by producing only very few, very accurate domains — to do well, a detector should have high precision even when it is forced to predict every domain.

Average precision evaluates detection for one category of object. The *mean average precision* (mAP) is the mean of the average precision for each category. The value depends on the IoU threshold chosen. One convention is to report mAP at $IoU = 0.5$. Another is to compute mAP at a set of 10 IoU values ($0.45 + i \times 0.05$ for $i \in 1 \ldots 10$), then average the mAP's. These evaluations produce numbers that tend to be bigger for better detectors, but it takes some practice to have a clear sense of what an improvement in mAP actually means.

********** rare category MAP

Remember this: *Evaluating object detectors should favor detectors that get the right number of the right objects in the right places, and should discourage detectors that just produce a lot of domains. Evaluation scores domains produced by the detector for relevance (is this the right domain in the right place?) using IoU scores to evaluate how well domains overlap with ground truth. The average precision is computed from an interpolated precision curve for each type of object. This is then averaged over object types to yield mAP.*

24.4.3 Training Considerations

The most natural training data for detectors is a large collection of images where each object has been labelled and had a box placed over it by a human annotator. Call any such box a positive, and any other box you happen to select in any of the images a negative. The vast majority of boxes are negatives, meaning there is a nasty imbalance problem in detection. This is usually dealt with by building training batches that have a fixed ratio of positive to negative boxes (typically ranging from 1:1 to 3:1; I will use 1:1 as an example, because it makes the sums easier).

A batch will contain a fixed number of boxes, say $2B$. Assemble a batch by selecting I images at random, where I is typically much smaller than $2B$. From these, select B positive boxes. If these images don't happen to have B positive boxes, copy some boxes at random. Select B negative boxes, which should be chosen randomly, but having size and aspect ratio like that of positive boxes. For example, you might build a simple probability distribution model of size and aspect ratio, draw from that model, and use that to obtain a negative box. The resulting set of $2B$ boxes is your batch. This strategy is sometimes known as a *image centric training*.

**** unsure about this

Early detectors were built on pretrained image encoders that were relatively straightforward, and tended to use fairly elaborate training procedures to avoid disrupting the weights of the image encoders too significantly. More recent options are: training the encoder and detector simultaneously, which is uncommon because you can usually get better performance from a pre-trained encoder; freezing the encoder and training only layers specific to the detector; and using a much smaller learning rate for the encoder parameters than for the detector parameters, so essentially fine-tuning the detector.