CHAPTER 18

Classifying Images and Detecting Objects

Neural networks have gone from being one curiousity in lists of classification methods to being the prime engine of a huge and very successful industry. This has happened in a very short time, less than a decade. The main reason is that, with enough training data and enough training ingenuity, neural networks produce very successful classification systems, much better than anyone has been able to produce with other methods. They are particularly good at classifying images. As Figure 18.1 shows, the top-5 error rate on one (very large and very hard) image classification dataset has collapsed in quite a short period. The primary reason seems to be that the features that are being used by the classifier are themselves learned from data. The learning process seems to ensure that the features are useful for classification. It's easy to see that it might do so; the news here is that it does.

There are two important trends that have advanced this area. One is the development of large, challenging (but not unreasonably hard) datasets, that are publicly available and where accuracy is evaluated using conventions that are fair and open. The second is the widespread dissemination of successful models. If someone produces a really good image classifier, you can usually find an implementation on the internet fairly soon afterwards. This means that it's easy to fiddle with successful architectures and try to make them better. Very often, these implementations come with pre-trained models.

This chapter will describe the main recent successes in image classification and object detection using neural networks. It's unlikely you would be able to build anything I describe here from the text alone, but you can likely find a trained version elsewhere. You should get a good enough grasp of what people do, what seems to work and why to apply and use models that have been shared.

18.1 IMAGE CLASSIFICATION

I will describe several important network architectures in the following subsections, but building any of these from scratch based only on this description would be a heroic (and likely unsuccessful) venture. What you should do is download a version for the environment you prefer, and play with that. You can find pretrained models at:

- https://pjreddie.com/darknet/imagenet/ (for darknet);
- http://www.vlfeat.org/matconvnet/pretrained/ (for matconvnet);
- https://mxnet.apache.org/api/python/gluon/model_zoo.html (for mxnet);
- https://github.com/PaddlePaddle/models (for PaddlePaddle; it helps to be able to read Chinese);

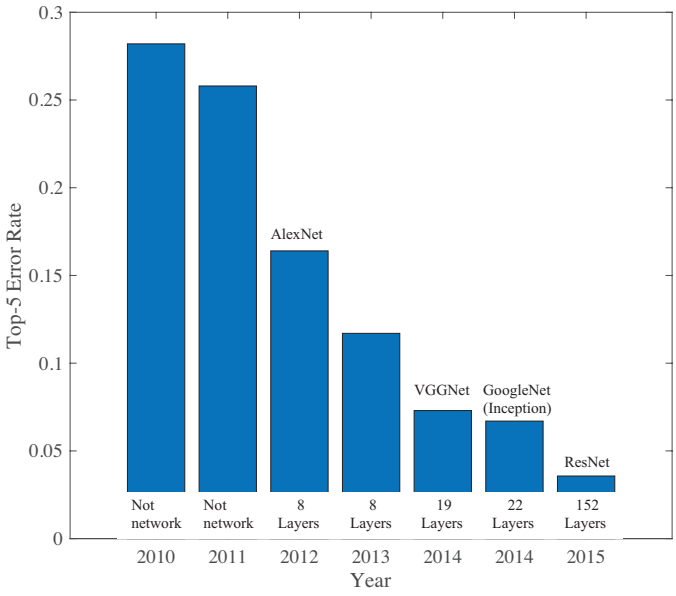

FIGURE 18.1: *The top-5 error rate for image classification using the ImageNet dataset has collapsed from* 28% *to* 3.6% *from 2010 to 2015. There are two 2014 entries here, which makes the fall in error rate look slower. This is because each of these methods is significant, and discussed in the sections below. Notice how increasing network depth seems to have produced reduced error rates. This figure uses ideas from an earlier figure by Kaiming He. Each of the named networks is described briefly in a section below.*

- https://pytorch.org/docs/stable/torchvision/models.html (for pytorch);
- https://github.com/tensorflow/models (for tensorflow);
- https://keras.io (for Keras; look for "examples" in the sidebar).

18.1.1 Datasets for Classifying Images of Objects

MNIST and CIFAR-10 are no longer cutting edge image classification datasets. The networks I described are quite simple, but work rather well on these problems. The very best methods are now extremely good. Rodrigo Benenson maintains a website giving best performance to date on these datasets at http://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; much better than our 2,000 odd). Mostly, methods work so well that improvements must be very small, and so it is difficult to see what is an important change and what is a lucky accident. These datasets are now mostly used for warming-up purposes – to check that an idea isn't awful, or that a method can work on an "easy" dataset.

Remember this: *MNIST and CIFAR-10 are warmup datasets. You can find MNIST at http://yann.lecun.com/exdb/mnist/ or at http://www.kaggle.com/c/digit-recognizer. You can find CIFAR-10 at https://www.cs.toronto.edu/kriz/cifar.html.*

It is difficult to say precisely what makes a dataset hard. It is very likely that having more categories makes a dataset harder than having few categories. It is very likely that having a lot of training data per category makes a dataset easier. It is certain that labelling errors and differences between test images and training images will cause problems. Modern datasets tend to be built carefully using protocols that try to ensure that the label for each data item is right. For example, one can have images labelled independently, then check the labels agree. There isn't any way of checking to see that the training set is like the test set, but one can collect first, then split later.

MNIST and CIFAR-10 contain pictures of largely isolated objects. A harder dataset is CIFAR-100. This is very like CIFAR-10, but now with 100 categories. Images are 32×32 color images in 100 categories, collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. There are 50, 000 training images (so now 500 per category, rather than 5,000) and 10, 000 test images, and the test-train split is standard. Images are evenly split between the classes. The categories are grouped rather roughly into superclasses, so that there are several different insect categories, several different reptile categories, and so on.

Remember this: *CIFAR-100 is a small hard image classification dataset. You can download this dataset from https://www.cs.toronto.edu/kriz/cifar.html. CIFAR-100 accuracy is also recorded at http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html. The best error rate (24.28%) is a crude indicator that this dataset is harder than CIFAR-10 or MNIST.*

There are several important big image classification datasets. Datasets tend to develop over time, and should be followed by looking at a series of workshops. The **Pascal** visual object classes challenges are a set of workshops held from 2005-2012 to respond to challenges in image classification. The workshops, which were a community wide effort led by the late Mark Everingham, resulted in a number of tasks and datasets which are still used. There is more information, including leaderboards, best practice, organizers, etc. at http://host.robots.ox.ac.uk/pascal/VOC/.

Remember this: *PASCAL VOC 2007 remains a standard image classification dataset. You can find this at (http://host.robots.ox.ac.uk/pascal/VOC/voc2007/index.html). The dataset uses a collection of 20 object classes that became a form of standard.*

There is very little point in classifying images of objects into classes that aren't useful, but it isn't obvious what classes should be used. One strategy is to organize classes in the same way that nouns for objects are organized. **WordNet** is a lexical database of the English language, organized hierarchically in a way that tries to represent the distinctions that people draw between objects. So, for example, a `cat` is a `felid` which is a `carnivore` which is a `placental mammal` which is a `vertebrate` which is a `chordate` which is an `animal` (and so on...). You can explore WordNet at https://wordnet.princeton.edu. **ImageNet** is a collection organized according to a semantic hierarchy taken from WordNet. ImageNet Large Scale Visual Recognition Challenge (ILSVRC) workshops were held from 2010-2017, organized around a variety of different challenges.

Remember this: *ImageNet is an extremely important large scale image classification dataset. A very commonly used standard is the ILSVRC2012 dataset, with 1000 classes and 1.28 million training images. There's a standard validation set of 50, 000 images (50 per category). You can find this at http://www.image-net.org/challenges/LSVRC/2012/nonpub-downloads. The dataset uses a collection of 1000 object classes that became a form of standard.*

18.1.2 Datasets for Classifying Images of Scenes

Objects tend to appear together in quite structured ways, so if you see a giraffe you might also expect to see an acacia or a lion, but you wouldn't expect to see a submarine or a couch. Different contexts tend to result in different groups of objects. So in grassland you might see a giraffe or a lion, and in the living room you might see a couch, but you don't expect a giraffe in a living room. This suggests that environments are broken up into clusters that look different and tend to contain different objects. Such clusters are widely called **scene**s in the vision community. An important image classification challenge is to take an image of a scene and predict what the scene is.

One important scene classification dataset is the **SUN** dataset. This is widely used for training, and for various classification challenges. There is a benchmark dataset with 397 categories. The full dataset contains over 900 categories and many

million images. Workshop challenges, including particular datasets used and leaderboards, appear at http://lsun.cs.princeton.edu/2016/ (LSUN 2016); and http://lsun.cs.princeton.edu/2017/ (LSUN 2017). The challenges use a selected subset of the scene categories.

Remember this: *SUN is a large-scale scene classification dataset that has been the core of several challenge workshops. The dataset appears at https://groups.csail.mit.edu/vision/SUN/.*

Another important dataset is the **Places-2** dataset. There are 10 million images in over 400 categories, with annotations of scene attributes and a variety of other materials.

Remember this: *Places-2 is a large-scale scene classification dataset. You can find this at http://places2.csail.mit.edu.*

18.1.3 Augmentation and Ensembles

Three important practical issues that need to be addressed to build very strong image classifiers.

- **Data sparsity:** Datasets of images are never big enough to show all effects accurately. This is because an image of a horse is still an image of a horse even if it has been through a small rotation, or has been resized to be a bit bigger or smaller, or has been cropped differently, and so on. There is no way to take account of these effects in the architecture of the network.
- **Data compliance:** We want each image fed into the network to be the same size.
- **Network variance:** The network we have is never the best network; training started at a random set of parameters, and has a strong component of randomness in it. For example, most minibatch selection algorithms select random minibatches. Training the same architecture on the same dataset twice will not yield the same network.

All three can be addressed by some care with training and test data.

Generally, the way to address data sparsity is **data augmentation**, by expanding the training dataset to include different rotations, scalings, and crops of images. Doing so is relatively straightforward. You take each training image, and generate a collection of extra training images from it. You can obtain this collection by: resizing and then cropping the training image; using different crops of the same

training image (assuming that training images are a little bigger than the size of image you will work with); rotating the training image by a small amount, resizing and cropping; and so on.

There are some cautions. When you rotate then crop, you need to be sure that no "unknown" pixels find their way into the final crop. You can't crop too much, because you need to ensure that the modified images are still of the relevant class, and too aggressive a crop might cut out the horse (or whatever) entirely. This somewhat depends on the dataset. If each image consists of a tiny object on a large background, and the objects are widely scattered, crops need to be cautious; but if the object covers a large fraction of the image, the cropping can be quite aggressive.

Cropping is usually the right way to ensure that each image has the same size. Resizing images might cause some to stretch or squash, if they have the wrong aspect ratio. This likely isn't a great idea, because it will cause objects to stretch or squash, making them harder to recognize. It is usual to resize images to a convenient size without changing the aspect ratio, then crop to a fixed size.

There are two ways to think about network variance (at least!). If the network you train isn't the best network (because it can't be), then it's very likely that training multiple networks and combining the results in some way will improve classification. You could combine results by, for example, voting. Small improvements can be obtained reliably like this, but the strategy is often deprecated because it isn't particularly elegant or efficient. A more usual approach is to realize that the network might very well handle one crop of a test image rather better than others (because it isn't the best network, etc.). Small improvements in performance can be obtained very reliably by presenting multiple crops of a test image to a given network, and combining the results for those crops.

18.1.4 Alexnet

The first really successful neural network image classifier was **Alexnet**, described in "ImageNet Classification with Deep Convolutional Neural Networks", a NIPS 2012 paper by Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton. Alexnet is quite like the simple networks we have seen – a sequence of convolutional layers that reduce the spatial dimensions of the data block, followed by some fully connected layers – but has a few special features. GPU memories in 2012 were much smaller than they are now, and the network architecture is constructed so that the data blocks can be split across two GPUs. There are new normalization layers, and there is a fully-connected layer that reduces a data block in size in a new way.

The impact of splitting the data blocks is quite significant. As Figure 18.2 shows, the image passes into a convolutional layer with 96 kernels followed by a ReLU, response normalization (which modifies values in a block, but doesn't change its size) and maxpooling. This would normally result in a data block of dimension $55 \times 55 \times 96$, but here each GPU gets a block consisting of the output of a different half of the kernels (so there are two $55 \times 55 \times 48$ blocks). Each goes through another convolutional layer of 128 kernels (size $5 \times 5 \times 48$), with a total of 256 kernels. The blocks on GPU 1 and GPU 2 may contain quite different features; the block on GPU 1 at B in the figure does *not* see the block on GPU 2 at A. The block at C

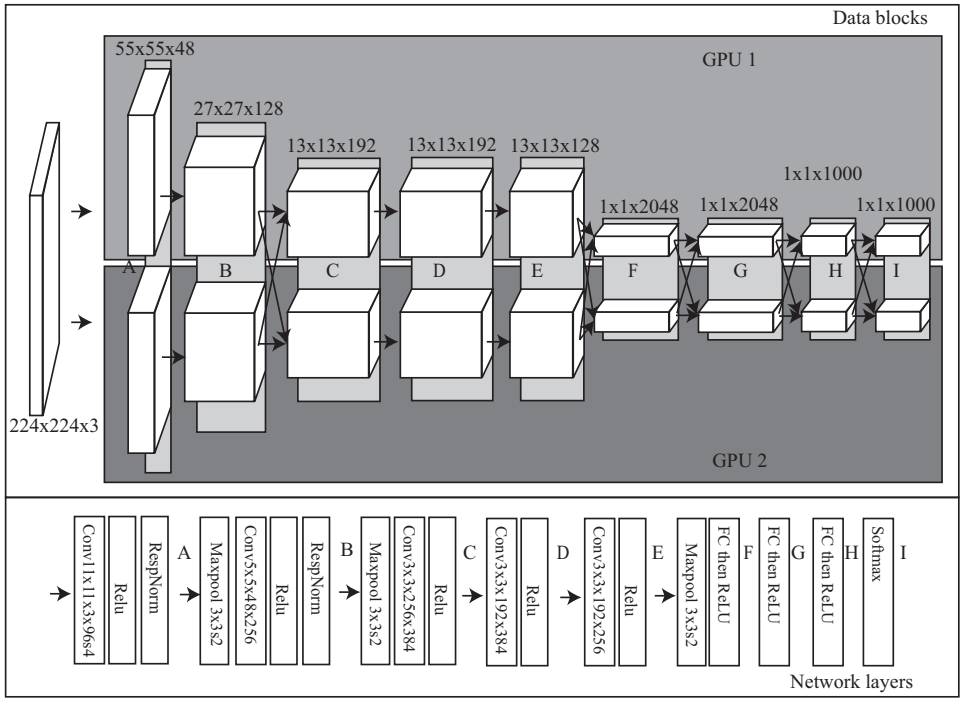

FIGURE 18.2: *Two views of the architecture of Alexnet, the first convolutional neural network architecture to beat earlier feature constructions at image classification. There are five convolutional layers with ReLU, response normalization and pooling layers interspersed.* **Top** *shows the data blocks at various stages through the network and* **bottom** *shows all the layers (capital letters key stages in the network to blocks of data). Horizontal and diagonal arrows in the top box indicate how data is split between GPUs, details in the main text. The response normalization layer is described in the text. I have compacted the final fully connected layers to fit the figure in.*

for each GPU is constructed using the block at B for both GPUs, but then blocks move through the network without interacting until the dense layer (which turns E into F). This means that features on one GPU could encode rather different properties, and this actually happens in practice.

For each location in a block, response normalization layers then scale the value at that location using a summary of nearby values. Response normalization like this is no longer widely used, so I will omit details. This network was trained using substantial data-augmentation, as above. Units in the first two layers are dropped out with a probability of 0.5. Training uses the usual stochastic gradient descent, but with momentum. Alexnet was a spectacular success, achieving top-1 and top-5 error rates of 37.5% and 17.0% respectively on the ImageNet ILSVRC-2010 challenge. These scores are significantly better than any other method had produced in the past, and stimulated widespread investigation into network architectures that

might do better.

Remember this: *Alexnet was a spectacular success at classifying ImageNet images.*

18.1.5 VGGNet

Alexnet has some odd features. It has relatively few layers. It splits data blocks across GPU's. The kernels in the first layer are large, and have a large stride. And it has response normalization layers. VGGnet is a family of networks built to investigate these and other issues. Using the best member of the family, the best practices in cropping, evaluation, data augmentation, and so on, VGGnet obtained top-1 and top-5 error rates of 23.7% and 6.8% respectively on the ImageNet ILSVRC-2014 challenge. This was a substantial improvement. Table 18.1 describes the five most important VGGnets (the sixth was used to establish that response normalization wasn't helpful for everything; this doesn't matter to us).

Table 18.1 is a more compact presentation of much of the information in Figure 18.2, but for the five VGGnets. The table shows the flow of information downwards. The naming conventions work like this. The term "convX-Y" means a convolutional layer of Y $X \times X$ kernels followed by a ReLU layer. The term "FC-X" means a fully connected layer that produces an X dimensional vector. For example, in VGGnet-A, a $224 \times 224 \times 3$ image passes into a layer, labelled "conv3-64". This consists of a convolutional layer of 64 $3 \times 3 \times 3$ kernels, followed by a ReLU layer. The block then passes into a maxpool layer, pooling over 2×2 windows with stride 2. The result goes to a convolutional layer of 128 $3 \times 3 \times 3$ kernels, followed by a ReLU layer. Eventually, the block of data goes to a fully connected layer that produces a 4096 dimensional vector ("FC-4096"), passes through another of these to an FC-1000 layer, and then to a softmax layer.

Reading across the table gives the different versions of the network. Notice that there are significantly more layers with trainable weights than for Alexnet. The E version (widely known as **VGG-19**) is the most widely used; others were mainly used in training, and to establish that more layers gives better performance. The networks have more layers as the version goes up. Terms in bold identify layers introduced when the network changes (reading right). So, for example, the B version has a conv3-64 term that the A version doesn't have, and the C, D and E versions keep; the C version has a conv1-512 term that the A and B versions don't have, and the D and E versions replace with a conv3-512 term.

You should expect that training a network this deep is hard (recall section 16.4.3). VGGnet training followed a more elaborate version of the procedure I used in section 16.4.3. Notice that the B version is the A version together with two new terms, etc. Training proceeded by training the A version. Once the A version was trained, the new layers were inserted to make a B version (keeping the parameter values of the A version's layers), and the new network was trained from

Network Architecture				
A	B	C	D	E
Number of layers with learnable weights				
11	13	16	16	19
Input (224x224x3 image)				
conv3-64	conv3-64	conv3-64	conv3-64	conv3-64
	conv3-64	conv3-64	conv3-64	conv3-64
maxpool2x2s2				
conv3-64	conv3-64	conv3-64	conv3-64	conv3-64
	conv3-64	conv3-64	conv3-64	conv3-64
maxpool2x2s2				
conv3-128	conv3-128	conv3-128	conv3-128	conv3-128
	conv3-128	conv3-128	conv3-128	conv3-128
maxpool2x2s2				
conv3-256	conv3-256	conv3-256	conv3-256	conv3-256
conv3-256	conv3-256	conv3-256	conv3-256	conv3-256
		conv1-256	**conv3-256**	conv3-256
				conv3-256
maxpool2x2s2				
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
		conv1-512	**conv3-512**	conv3-512
				conv3-512
maxpool2x2s2				
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
		conv1-512	**conv3-512**	conv3-512
				conv3-512
maxpool2x2s2				
FC-4096				
FC-4096				
FC-1000				
softmax				

TABLE 18.1: *This table summarizes the architecture of five VGGnets. Details in the text.*

that initialization. All parameter values in the new network were updated. The C version was then trained from B, and so on. All training is by minibatch stochastic gradient descent with momentum. The first two layers were subject to dropout (probability of dropout 0.5). Data was aggressively augmented.

Experiment suggests that the features constructed by VGG-19 and networks like it are canonical in some way. If you have a task that involves computing something from an image, using VGG-19 features for that task very often works. Alternatively, you could use VGG-19 as an initialization for training a network for your task. VGG-19 is still widely used as a **feature stack** – a network that was trained for classification, but whose features are being used for something else.

Remember this: *VGGNet outperformed Alexnet at classifying ImageNet images. There are several versions. VGG-19 is still used to produce image features for other tasks.*

18.1.6 Batch Normalization

There is good experimental evidence that large values of inputs to any layer within a neural network lead to problems. One source of the problem could be this. Imagine some input to some unit has a large absolute value. If the corresponding weight is relatively small, then one gradient step could cause the weight to change sign. In turn, the output of the unit will swing from one side of the ReLU's non-linearity to the other. If this happens for too many units, there will be training problems because the gradient is then a poor prediction of what will actually happen to the output. So we should like to ensure that relatively few values at the input of any layer have large absolute values. We will build a new layer, sometimes called a **batch normalization layer**, which can be inserted between two existing layers.

Write $\mathbf{x}^b$ for the input of this layer, and $\mathbf{o}^b$ for its output. The output has the same dimension as the input, and I shall write this dimension d. The layer has two vectors of parameters, γ and β, each of dimension d. Write $\text{diag}(\mathbf{v})$ for the matrix whose diagonal is $\mathbf{v}$, and with all other entries zero. Assume we know the mean ($\mathbf{m}$) and standard deviation ($\mathbf{s}$) of each component of $\mathbf{x}^b$, where the expectation is taken over all relevant data. The layer forms

$$\begin{aligned}\mathbf{x}^n &= [\text{diag}(\mathbf{s}+\epsilon)]^{-1}\left(\mathbf{x}^b - \mathbf{m}\right)\\ \mathbf{o}^b &= [\text{diag}(\gamma)]\,\mathbf{x}^n + \beta.\end{aligned}$$

Notice that the output of the layer is a differentiable function of γ and β. Notice also that this layer *could* implement the identity transform, if $\gamma = \text{diag}(\mathbf{s}+\epsilon)$ and $\beta = \mathbf{m}$. We adjust the parameters in training to achieve the best performance. It can be helpful to think about this layer as follows. The layer rescales its input to have zero mean and unit standard deviation, then allows training to readjust the mean and standard deviation as required. In essence, we expect that large values encountered between layers are likely an accident of the difficulty training a network, rather than required for good performance.

The difficulty here is we don't know either $\mathbf{m}$ or $\mathbf{s}$, because we don't know the parameters used for previous layers. Current practice is as follows. First, start with $\mathbf{m} = \mathbf{0}$ and $\mathbf{s} = \mathbf{1}$ for each layer. Now choose a minibatch, and train the network using that minibatch. Once you have taken enough gradient steps and are ready to work on another minibatch, reestimate $\mathbf{m}$ as the mean of values of the inputs to the layer, and $\mathbf{s}$ as the corresponding standard deviations. Now obtain another minibatch, and proceed. Remember, γ and β are parameters that are trained, just like the others (using gradient descent, momentum, adagrad, or whatever). Once

the network has been trained, one then takes the mean (resp. standard deviation) of the layer inputs over the training data for $\mathbf{m}$ (resp. $\mathbf{s}$). Most neural network implementation environments will do all the work for you. It is quite usual to place a batch normalization layer between each layer within the network.

For some problems, minibatches are small, usually because one is using a large model or a large data item and its hard to cram many items into the GPU. If you have many GPUs, you can consider synchronizing the minibatches and then averaging over all the minibatches being presented to the GPU — this isn't for everybody. If the minibatch is small, then the estimate of $\mathbf{m}$ and $\mathbf{s}$ obtained using a minibatch will be noisy, and batch normalization typically performs poorly. For many problems involving images, you can reasonably expect that groups of features should share the same scale. This justifies using **group normalization**, where the feature channels are normalized in groups across a minibatch. The advantage of doing so is that you will have more values to use when estimating the parameters; the disadvantage is that you need to choose which channels form groups.

There is a general agreement that normalization improves training, but some disagreement about the details. Experiments comparing two networks, one with normalization the other without, suggest that the same number of steps tends to produce a lower error rate when batch normalized. Some authors suggest that convergence is faster (which isn't quite the same thing). Others suggest that larger learning rates can be used.

> **Remember this:** *Batch normalization improves training by discouraging large numbers in datablocks that aren't required for accuracy. When minibatches are small, it can be better to use group normalization, where one normalizes over groups of features.*

18.1.7 Computation Graphs

In section 113, I wrote a simple network in the following form

$$\begin{aligned}
&\mathbf{o}^{(D)} \\
&\text{where} \\
\mathbf{o}^{(D)} &= \mathbf{o}^{(D)}(\mathbf{u}^{(D)}, \theta^{(D)}) \\
\mathbf{u}^{(D)} &= \mathbf{o}^{(D-1)}(\mathbf{u}^{(D-1)}, \theta^{(D-1)}) \\
\ldots &= \ldots \\
\mathbf{u}^{(2)} &= \mathbf{o}^{(1)}(\mathbf{u}^{(1)}, \theta^{1}) \\
\mathbf{u}^{(1)} &= \mathbf{x}.
\end{aligned}$$

These equations really were a map for a computation. You feed in $\mathbf{x}$; this gives $\mathbf{u}^{(1)}$; which gives $\mathbf{u}^{(2)}$; and so on, up to $\mathbf{o}^{(D)}$. The gradient follows from passing information back down this map. These procedures don't require that any layer

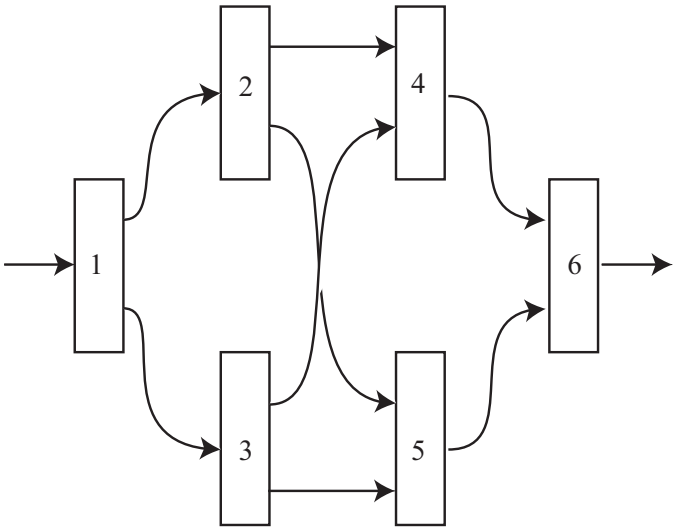

FIGURE 18.3: *A simple computation graph. You should reassure yourself that a straightforward adjustment to backpropagation will yield all gradients of interest for this network.*

has only one input or that any layer has only one output. All we need is to connect the inputs and the outputs in a directed acyclic graph, so that at any node we know what it means for information to go forward (resp. backward). This graph is known as a **computation graph**. Figure 18.3 shows an example that you should use to check that you understand how gradients would be computed. A key feature of good software environments is that they support building complex computation graphs.

18.1.8 Inception Networks

Up to here, we have seen image classification networks as a sequence of layers, where each layer has one input and one output, and information passes from layer to layer in order, and in blocks. This isn't necessary for backpropagation to work. It's enough to have a set of blocks (equivalent to our layers), each with possibly more than one input and possibly more than one outputs. As long as you know how to differentiate each output with respect to each input, and as long as outputs are connected to inputs in a directed acyclic graph, backpropagation works.

This means that we can build structures that are far richer than a sequence of layers. A natural way to do this is to build layers of modules. Figure 18.4 shows two **inception modules** (of a fairly large vocabulary that you can find in the literature; there are some pointers at the end of the chapter). The base block passes its input to each output. A block labelled "AxB" is a convolution layer of $A \times B$ kernels followed by a layer of ReLUs; a stack block stacks each of the data blocks from its input to form its output.

Modules consist of a set of streams that operate independently on a data

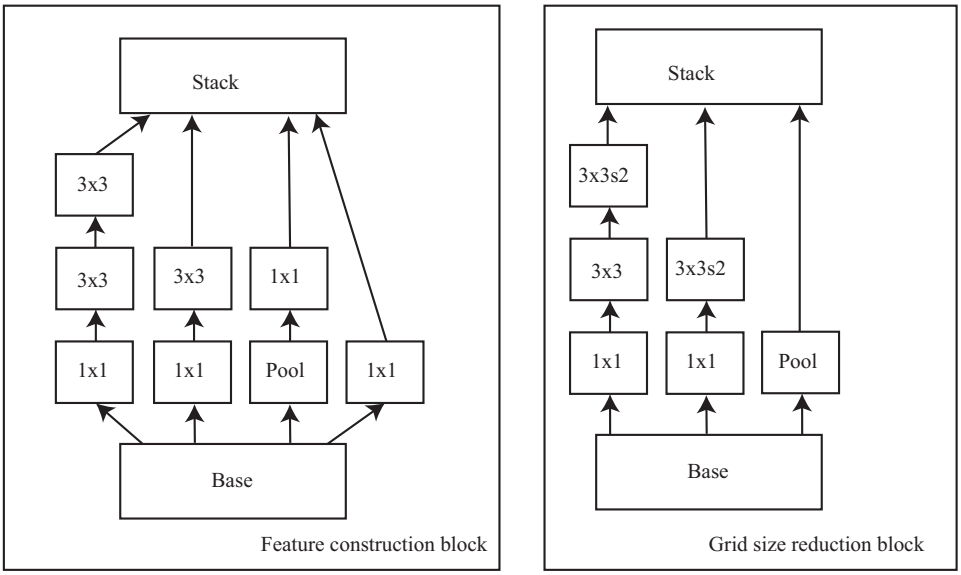

FIGURE 18.4: *On the* **left** *an inception module for computing features. On the* **right**, *a module that reduces the size of the grid. The feature module features with:* 5×5 *support (far left stream);* 3×3 *support (left stream);* 1×1 *support after pooling (right stream); and* 1×1 *support without pooling. These are then stacked into a block. The grid size reduction module takes a block of features on a grid, and reduces the size of the grid. The stream on the left constructs a reduced size grid of features that have quite broad support (*5×5 *in the input stream); the one in the center constructs a reduced size grid of features that have medium support (*3×3 *in the input stream); and the one on the right just pools. The outputs of these streams are then stacked.*

block; the resulting blocks are then stacked. Stacking means each stream must produce a block of the same spatial size, so all the streams must have consistent stride. Each of the streams has a 1×1 convolution in it, which is used for dimension reduction. This is means that if you stack two modules, each stream in the top module can select from features which look at the incoming data over different spatial scales. This selection occurs because the network learns the linear map that achieves dimension reduction. In the network, some units can specialize in big (or small, or mixed size) patterns, and later units can choose to make their patterns out of big (or small, or mixed size) components.

There are many different inception modules, and a rich collection of possible networks built out of them. Networks built out of these modules are usually called inception networks. Inception networks tend to be somewhat smaller and faster than VGG-19. An inception network (with appropriate practices in cropping, evaluation, data augmentation, and so on) obtained top-1 and top-5 error rates of 21.2% and 5.6% respectively on the ImageNet ILSVRC-2012 classification challenge dataset. This was a substantial improvement. As you would expect,

training can be tricky. It's usual to use RMSprop.

> **Remember this:** *Inception networks outperformed VGG-19 on ImageNet. Inception networks are built of modules. Feature modules select from incoming data using a 1×1 convolution, then construct features at different spatial scales, thenstack them. Other modules reduce the size of the spatial grid. Training can be tricky.*

18.1.9 Residual Networks

A randomly initialized deep network can so severely mangle its inputs that only a wholly impractical amount of training will cause the latest layers to do anything useful. As a result, there have been practical limits on the number of layers that can be stacked. One recent strategy for avoiding this difficulty is to use **residual connections**.

Our usual process takes a data block $\mathcal{X}^{(l)}$, forms a function of that block $\mathcal{W}(\mathcal{X}^{(l)})$, then applies a ReLU to the result. To date, the function involves applying either a fully connected layer or a convolution, then adding bias terms. Writing $F(\cdot)$ for a ReLU, we have

$$\mathcal{X}^{(l+1)} = F(\mathcal{W}(\mathcal{X}^{(l)}).$$

Now assume the linear function does not change the size of the block. We replace this process with

$$\mathcal{X}^{(l+1)} = F(\mathcal{W}(\mathcal{X}^{(l)}) + \mathcal{X}^{(l)}$$

(where F, $\mathcal{W}$ etc. are as before). The usual way to think about this is that a layer now passes on its input, but adds a residual term to it. The point of all this is that, at least in principle, this residual layer can represent its output as a small offset on its input. If it is presented with large inputs, it can produce large outputs by passing on the input. Its output is also significantly less mangled by stacking layers, because its output is largely given by its input plus a non-linear function. These residual connections can be used to bypass multiple blocks. Networks that use residual connections are often known as **ResNets**.

There is good evidence that residual connections allow layers to be stacked very deeply indeed (for example, 1001 layers to get under 5% error on CIFAR-10; beat that if you can!). One reason is that there are useful components to the gradient for each layer that do not get mangled by previous layers. You can see this by considering the Jacobian of such a layer with respect to its inputs. You will see that this Jacobian will have the form

$$\mathcal{J}_{\mathbf{o}^{(l)};\mathbf{u}^l} = (\mathcal{I} + \mathcal{M}_l)$$

where $\mathcal{I}$ is the identity matrix and $\mathcal{M}_l$ is a set of terms that depend on the map $\mathcal{W}$. Now remember that, when we construct the gradient at the k'th layer, we evaluate

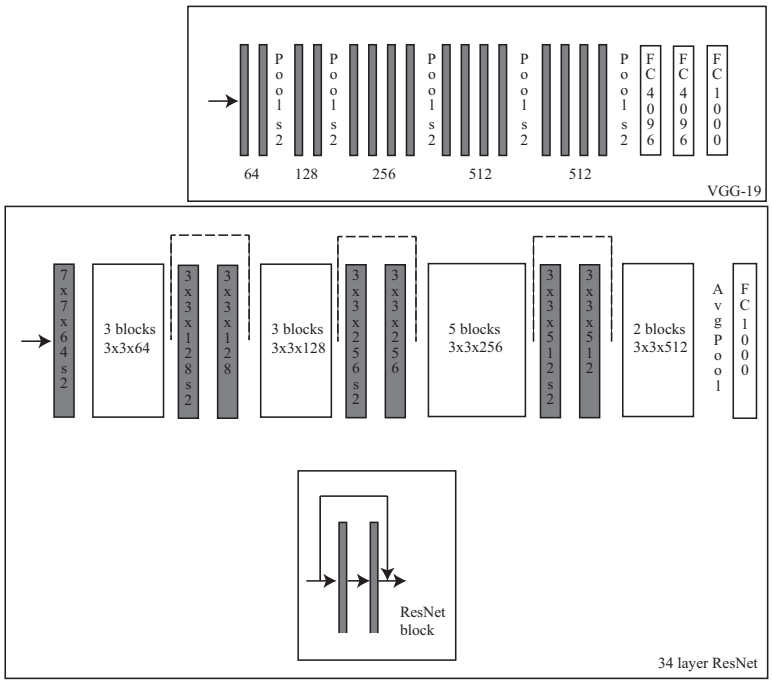

FIGURE 18.5: *Comparing VGG-19 to a 34 layer ResNet requires an even more compact graphical representation. Each shaded box is a convolutional layer of $3 \times 3 \times D$ kernels followed by a ReLU. The number of kernels is given the notation below the box, and D follows by matching sizes. Every layer with learnable parameters is represented by a box, so VGG-19 has 19 such layers, together with pooling layers. The 34 layer ResNet has 34 such layers. There are a few specialized layers (text in the box), but most appear in the blocks (inset) which have two $3 \times 3 \times D$ layers with a residual connection that skips both. These blocks are stacked, as indicated in the figure. The dashed lines around grayed blocks represent a residual connection that causes the size of the data block to change.*

by multiplying a set of Jacobians corresponding to the layers above. This product in turn must look like

$$(\nabla_{\mathbf{o}^{(D)}} L)\, J_{\mathbf{o}^{(D)};\mathbf{u}^{(D)}} \times J_{\mathbf{o}^{(D-1)};\mathbf{u}^{(D-1)}} \times \ldots \times J_{\mathbf{o}^{k};\theta^{k}}$$

which is

$$(\nabla_{\mathbf{o}^{(D)}} L)\, (\mathcal{I} + \mathcal{M}_D)(\mathcal{I} + \mathcal{M}_{D-1}) \ldots (\mathcal{I} + \mathcal{M}_{l+1}) \mathcal{J}_{\mathbf{x}^{k+1};\theta^{k}}$$

which is

$$(\nabla_{\mathbf{o}^{(D)}} L)\, (\mathcal{I} + \mathcal{M}_D + \mathcal{M}_{D-1}) \ldots + \mathcal{M}_{l+1} + \ldots) \mathcal{J}_{\mathbf{x}^{k+1};\theta^{k}}.$$

which means that some components of the gradient at that layer do not get mangled by being passed through a sequence of poorly estimated Jacobians.

For some choices of function, the size of the block changes. In this case, we cannot use the form $\mathcal{X}^{(l+1)} = F(\mathcal{W}(\mathcal{X}^{(l)}) + \mathcal{X}^{(l)}$, but instead use

$$\mathcal{X}^{(l+1)} = F(\mathcal{W}(\mathcal{X}^{(l)}) + \mathcal{G}(\mathcal{X}^{(l)})$$

where $\mathcal{G}$ represents a learned linear projection of $\mathcal{X}^{(l)}$ to the right size block.

It is possible to train very deep networks with this structure very successfully. Figure 18.5 compares a 34 layer residual network with a VGG-19 network. A network with this structure (with appropriate practices in cropping, evaluation, data augmentation, and so on) obtained top-1 and top-5 error rates of 24.2% and 7.4% respectively on the ImageNet ILSVRC-2012 classification challenge validation dataset. This is somewhat worse than the inception network performance, but accuracy can be significantly improved by building deeper networks (hard to draw) and using ensembles, voting over different crops, and so on. A model using 152 layers (ResNet-152) obtained a top-5 error of 3.57% ImageNet ILSVRC-2015 challenge. ResNet-152 is widely used as a feature stack, and is usually more accurate than VGGNet.

Remember this: *ResNets are the go-to for image classification. ResNets use a network block that adds a processed version of the input to the input. This means that helpful gradient values are available even for very deep networks. ResNet models can be built with extremely deep networks, and are widely used to make features for tasks other than image classification.*

18.2 OBJECT DETECTION

An object detection program must mark the locations of each object from a known set of classes in test images. Object detection is hard for many reasons. First, objects 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. Second, 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). Third, many objects (like people) deform without changing their identity. Fourth, 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. Finally, most scenes contain an awful lot of objects (think about the number of bolts in a picture of a lecture hall – each chair has many) and most are not worth mentioning.

18.2.1 How Object Detectors Work

Object detectors are built out of image classifiers. Here is the simplest way to build (say) a face detector. Build an image classifier that can tell whether a face is present

in an image window of fixed size or not. This classifier produces a high score for faces, and a low score for non-faces. Take this face classifier, and search through a set of windows selected from the image. Use the resulting scores to decide which windows contain faces. This very simple model exposes the big questions to be addressed. We must:

- **Decide on a window shape:** this is easy. There are two possibilities: a box, or something else. Boxes are easy to represent, and are used for almost all practical detectors. The alternative — some form of mask that cuts the object out of the image — is hardly ever used, because it is hard to represent.
- **Build a classifier for windows:** this is easy – we've seen multiple constructions for image classifiers.
- **Decide which windows to look at:** this turns out to be an interesting problem. Searching all windows isn't efficient.
- **Choose which windows with high classifier scores to report:** this is interesting, too, because windows will overlap, and we don't want to report the same object multiple times in slightly different windows.
- **Report the precise locations of all faces using these windows:** this is also interesting. It turns out our window is likely not the best available, and we can improve it after deciding it contains a face.

Which window to look at is hard, and most innovation has occurred here. Each window is a hypothesis about the **configuration** (position and size) of the object. The very simplest procedure for choosing windows is to use all windows on some grid (if you want to find larger faces, use the same grid on a smaller version of the image). No modern detector looks at a grid because it is inefficient. A detector that looks at closely spaced windows may be able to **localize** (estimate position and size of) the object more accurately. But more windows means the classifier's false positive rate must be extremely small to produce a useful detector. Tiling the image tends to produce far too many windows, many of which are fairly obviously bad (for example, a box might cut an object in half).

Deciding which windows to report presents minor but important problems. Assume you look at 32×32 windows with a stride of 1. Then there will be many windows that overlap the object fairly tightly, and these should have quite similar scores. Just thresholding the value of the score will mean that we report many instances of the same object in about the same place, which is unhelpful. If the stride is large, no window may properly overlap the object and it might be missed. Instead, most methods adopt variants of a greedy algorithm usually called **non-maximum suppression**. First, build a sorted list of all windows whose score is over threshold. Now repeat until the list is empty: choose the window with highest score, and accept it as containing an object; now remove all windows with large enough overlap on the object window.

Deciding precisely where the object is also presents minor but important problems. Assume we have a window that has a high score, and has passed through non-maximum suppression. The procedure that generated the window does not do a detailed assessment of all pixels in the window (otherwise we wouldn't have needed the classifier), so this window likely does not represent the best localization of the object. A better estimate can be obtained by predicting a new bounding

box using a feature representation for the pixels in the current box. It's natural to use the feature representation computed by the classifier for this **bounding box regression** step.

> **Remember this:** *Object detectors work by passing image boxes that are likely to contain objects into a classifier. The classifier gives scores for each possible object in the box. Multiple detections of the same object by overlapping boxes can be dealt with by non-maximum suppression, where higher-scoring boxes eliminate lower-scoring but overlapping boxes. Boxes are then adjusted with a bounding box regression step.*

18.2.2 Selective Search

The simplest procedure for building boxes is to slide a window over the image. This is simple, but works rather badly. It produces a large number of boxes, and the boxes themselves ignore important image evidence. Objects tend to have quite clear boundaries in images. For example, if you are looking at a picture of a horse in a field, there's usually no uncertainty about where the horse ends and where the field begins. At these boundaries, a variety of image properties change quite sharply. At the boundary of the horse, color changes (say, brown to green); texture changes (say, smooth skin to rough grass); intensity changes (say, dark brown horse to brighter green grass); and so on.

Making boxes by sliding windows ignores this information. 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, because not every boundary has a color change (think of a brown horse in a brown field), *and* some color changes occur away from boundaries (think about the stripes on a zebra). Nonetheless, it has been known for some time that one can use boundaries to score boxes for their "objectness". The best detectors are built by looking only at boxes that have a high enough objectness score.

The standard mechanism for computing such boxes is known as **selective search**. A quick description is straightforward, but the details matter (and you'll need to look them up). First, one breaks up the image into **regions** — groups of pixels that have coherent appearance — using an agglomerative clusterer. The agglomerative clusterer is quite important, because the representation it produces allows big regions to be made of smaller regions (so, for example, a horse might be made of a head, body and legs). Second, one scores the regions produced by the clusterer for "objectness". This score is computed from computing a variety of region features, encoding color, texture and so on. Finally, the regions are ranked by the score. It isn't safe to assume that regions with a score over some threshold are objects and the others aren't, but the process is very good at reducing the

number of boxes to look at. One does not need to go very deep into the ranked list of regions to find all objects of interest in a picture (2000 is a standard).

Remember this: *Image boxes that are likely to contain objects are closely related to regions. Selective search finds these boxes by building a region hierarchy, then scoring regions for objectness; regions with good objectness score produce bounding boxes. This gives an effective way of finding the boxes that are likely to contain objects.*

18.2.3 R-CNN, Fast R-CNN and Faster R-CNN

There is a natural way to build a detector using selective search and an image classifier. Use selective search to build a ranked list of regions. For each region in the ranked list, build a bounding box. Now warp this box to a standard size, and pass the resulting image to an image classifier. Rank the resulting boxes by the predicted score for the best object, and keep boxes whose score is over a threshold. Now apply non-maximum suppression and bounding box regression to that list. Figure 134 shows this architecture, known as **R-CNN**; it produces a very successful detector, but a speedup is available.

The problem with R-CNN is that one must pass each box independently through an image classifier. There tends to be a high degree of overlap between the boxes. This means the image classifier has to compute the neural network features at a given pixel for every box that overlaps the pixel, so doing unnecessary redundant work. The cure produces a detector known as **Fast R-CNN**. Pass the whole image through a convolutional neural network classifier (but ignore the fully connected layers). Now take the boxes that come from selective search, and use them to identify regions of interest (ROI's) in the feature maps. Compute class probabilities from these regions of interest using image classification machinery.

The ROI's will have different sizes, depending on the scale of the object. These need to be reduced to a standard size, otherwise we cannot pass them into the usual machinery. The trick is a **ROI pooling layer**, which produces a standard size summary of each ROI that is effective for classification. Decide on a standard size to which the ROI's will be reduced (say $r_x \times r_y$). Make a stack of grids this size, one per ROI. For each ROI, break the ROI into an $r_x \times r_y$ grid of evenly sized blocks. Now compute the maximum value in each block, and place that value in the corresponding location in the grid representing the ROI. This stack of grids can then be passed to a classifier.

The culmination of this line of reasoning (so far!) is **Faster R-CNN**. It turns out that selective search slows down Fast R-CNN. At least part of this slow down is computing features, etc. for selective search. But selective search is a process that predicts boxes from image data. There is no particular reason to use special features for this purpose, and it is natural to try and use the same set of features to

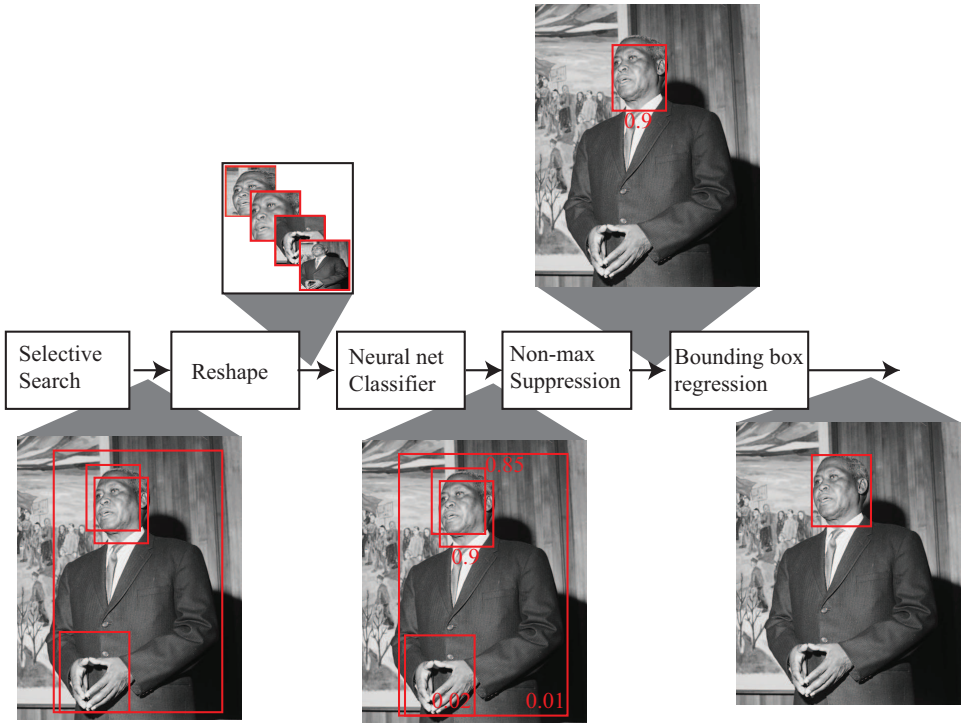

FIGURE 18.6: *A schematic picture of how R-CNN works. A picture of Inkosi Albert Luthuli is fed in to selective search, which proposes possible boxes; these are cut out of the image, and reshaped to fixed size; the boxes are classified (scores next to each box); non-maximum suppression finds high scoring boxes and suppresses nearby high scoring boxes (so his face isn't found twice); and finally bounding box regression adjusts the corners of the box to get the best fit using the features inside the box.*

predict boxes and to classify them. Faster R-CNN uses image features to identify important boxes.

Convolutional neural networks aren't particularly good at making lists, but are very good at making spatial maps. The trick is to encode a large collection of image boxes in a representation of fixed size that can be thought of as a map. The set of boxes can be represented like this. Construct a 3D block where each spatial location in the block represents a point on a grid in the image (a stride of 16 between the gridpoints in the original). The third coordinate in the block represents an **anchor box**. These are boxes of different size and aspect ratio, centered at the grid location (Figure 18.8; 9 in the original). You might be concerned that looking at a relatively small number of sizes, locations and aspect ratios creates problems; but bounding box regression is capable of dealing with any issues that arise. We want the entries in this map to be large when a box is likely to contain an object (you can think of this as an "objectness" score) and small otherwise. Thresholding the boxes and using non-maximum suppression yields a list of possible boxes, which

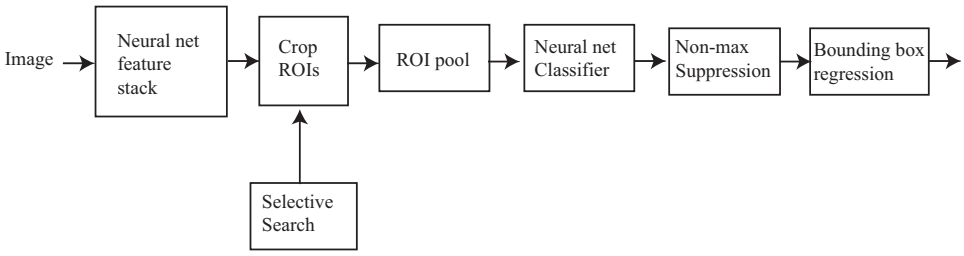

FIGURE 18.7: *Fast R-CNN is much more efficient than R-CNN, because it computes a single feature map from the image, then uses the boxes proposed by selective search to cut regions of interest (ROI's) from it. These are mapped to a standard size by a ROI pooling layer, then presented to a classifier. The rest should be familiar.*

can be handled as above.

A significant attraction of this approach is that the process that makes boxes can be trained at the same time as the classifier – box proposals can take classifier eccentricities in mind, and vice versa. At training time, one needs two losses. One loss measures the effectiveness of the box proposal process and the other measures the accuracy of the detector. The main difference is that the box proposal process needs to give a high score to any box with a good IoU against any ground truth bounding box (whatever the object in the box). The detector needs to name the object in the box.

Remember this: *R-CNN, Fast R-CNN and Faster R-CNN are strongly performing object detection systems that differ by how boxes are proposed. R-CNN and Fast R-CNN use selective search; Faster R-CNN scores anchor boxes. As of writing, Faster R-CNN is the reference object detector.*

18.2.4 YOLO

All the detectors we have seen so far come up with a list of boxes that are likely to be useful. **YOLO** (You Only Look Once) is a family of detectors (variants pay off accuracy against speed) that uses an entirely different approach to boxes. The image is divided into an $S \times S$ grid of tiles. Each tile is responsible for predicting the box of any object whose center lies inside the tile. Each tile is required to report B boxes, where each box is represented by the location of its center in the tile together with its width and its height. For each of these boxes (write b), each tile must also report a box confidence score $c(b(\text{tile}))$. The method is trained to produce a confidence score of zero if no object has its center in the tile, and the IoU for the box with ground truth if there is such an object (of course, at run time it might not report this score correctly).

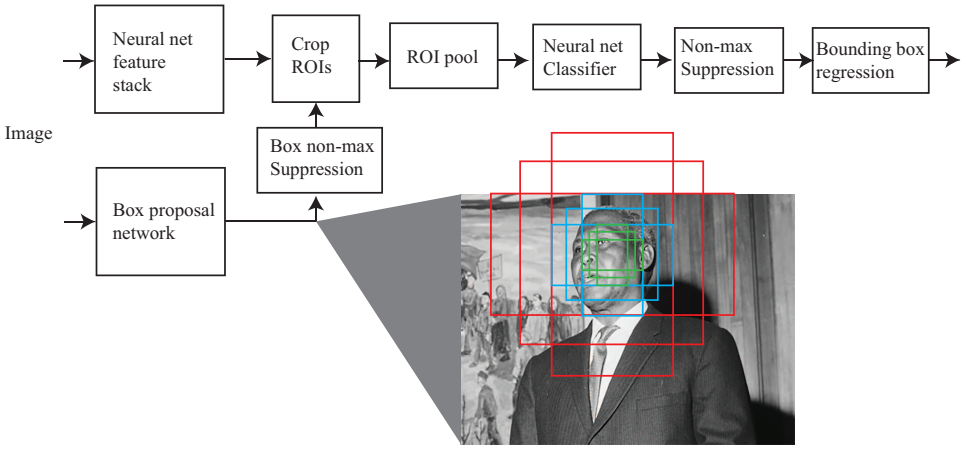

FIGURE 18.8: *Faster RCNN uses two networks. One uses the image to compute "objectness" scores for a sampling of possible image boxes. The samples (called "anchor boxes") are each centered at a grid point. At each grid point, there are nine boxes (three scales, three aspect ratios). The second is a feature stack that computes a representation of the image suitable for classification. The boxes with highest objectness score are then cut from the feature map, standardized with ROI pooling, then passed to a classifier. Bounding box regression means that the relatively coarse sampling of locations, scales and aspect ratios does not weaken accuracy.*

Each tile also reports a class-posterior, $p(\text{class}|\text{tile})$ for that tile. The score linking each of the boxes b in a tile to a class is then computed as

$$c(b(\text{tile})) \times p(\text{class}|\text{tile}).$$

Notice how the box scoring process has been decoupled from the object class process. Each tile is scoring *what* object overlaps the tile and also scoring which boxes linked to the tile are important. But these scores are computed separately – the method does not know which box is being used when it computes the object scores. This means the method can be extremely fast, and YOLO offers relatively easy tradeoffs between speed and accuracy, which are often helpful (for example, one can use more or fewer network layers to make features; more or fewer boxes per tile; and so on).

Decoupling boxes from classes comes with problems. YOLO tends to handle small objects poorly. There is a limited number of boxes, and so the method has difficulties with large numbers of small objects. The decision as to whether an object is present or not is based on the whole tile, so if the object is small compared to the tile, the decision might be quite inaccurate. YOLO tends not to do well with new aspects or new configurations of familiar objects. This is caused by the box prediction process. If the method is trained on (say) all vertical views of trees (tall thin boxes), it can have trouble with a tree lying on its side (short wide box).

Remember this: *The YOLO family of detectors work very differently from the R-CNN family. In Yolo, image tiles produce objectness scores for boxes and a classification score for objects independently; these are then multiplied. The advantage is speed, and tunable payoffs between speed and accuracy. The disadvantages are that many small objects are hard to detect, and new configurations of familiar objects are often missed.*

18.2.5 Evaluating Detectors

Evaluating object detectors takes care. An object detector takes an image, and, for each object class it knows about, produces a list of boxes each of which has a score. Evaluating the detector involves comparing these boxes with ground truth boxes 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 propose an awful lot of boxes. Getting this right takes a fair amount of careful work, which won't appeal to (or be useful to) all. The rest of the section is skippable if you're not that interested in object detection.

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 boxes, the other ($\mathcal{D}$) is the list of boxes the detector produces, which has already been subject to non-maximum suppression, bounding box regression, and anything else 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 boxes, and produced some boxes 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 boxes in $\mathcal{D}$ with `relevant` if they match ground truth boxes and `irrelevant` otherwise, and then summarize the lists.

The boxes that the detector predicts are unlikely to match ground truth exactly, and we need some way of telling whether the boxes are good enough. The standard method for doing this is to test the **IoU** (Intersection over Union). Write B_g for the ground truth box and B_p for the predicted box. The IoU is

$$\text{IoU}(B_p, B_g) = \frac{\text{Area}(B_g \cap B_p)}{\text{Area}(B_g \cup B_p)}.$$

Choose some threshold t. If $\text{IoU}(B_p, B_g) > t$, then B_p could match the ground truth box B_g.

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

- Choose a threshold t.
- Order $\mathcal{D}$ by the score of each box, and mark every element of $\mathcal{D}$ with `irrelevant`. Choose a threshold t.

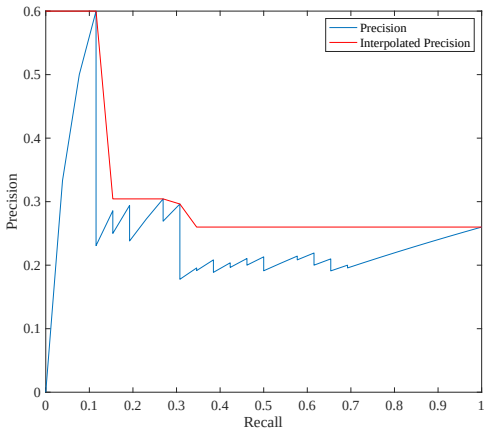

FIGURE 18.9: *Two plots for an imaginary search process. The precision plotted against recall shows a characteristic sawtooth shape. Interpolated precision measures the best precision you can get by increasing the recall, and so smoothes the plot. Interpolated precision is also a more natural representation 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.*

- For each element of $\mathcal{D}$ in order of score, compare that box against all ground truth boxes. If any ground truth box has IoU $> t$, mark the detector box `relevant` and remove that ground truth box from $\mathcal{G}$. Proceed until there are no more ground truth boxes.

Now every box 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. 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 18.9). 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 18.9). 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 boxes — to do well, a detector should have high precision even when it is forced to predict every box.

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.

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 boxes. Evaluation scores boxes produced by the detector for relevance (is this the right box in the right place?) using IoU scores to evaluate how well boxes 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.*

18.3 FURTHER READING

To proceed further, you really should be reading original papers, which is how this subject is communicated. Here's a reading list to get started with.

- **Origins of CNN's:** *Gradient-based learning applied to document recognition*, by Yann LeCun, Léon Bottou, Yoshua Bengio and Patrick Haffner, Proceedings of the IEEE 86 (11), 2278-2324
- **Batch normalization:** *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, by Sergey Ioffe, Christian Szegedy, Proc Int. Conf. Machine Learning, 2015. You can find a version at https://arxiv.org/abs/1502.03167.
- **ImageNet:** *ImageNet Large Scale Visual Recognition Challenge*, by Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg and Li Fei-Fei in International Journal of Computer Vision December 2015, Volume 115, Issue 3, pp 211–252.

- **Pascal:** *The Pascal Visual Object Classes (VOC) Challenge*, by Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn and Andrew Zisserman, International Journal of Computer Vision, June 2010, Volume 88, Issue 2, pp 303–338
- **VGGNet:** *Very Deep Convolutional Networks for Large-Scale Image Recognition* by Karen Simonyan and Andrew Zisserman, Proc. Int. Conf. Learned Representations, 2015. You can find a version of this at https://arxiv.org/pdf/1409.1556.pdf.
- **Inception:** *Going Deeper with Convolutions*, by Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke and Andrew Rabinovich, Proc Computer Vision and Pattern Recognition, 2015. You can find a version of this at https://arxiv.org/abs/1409.4842.
- **ResNets:** *Deep Residual Learning for Image Recognition* by Kaiming He, Xiangyu Zhang, Shaoqing Ren and Jian Sun, Proc Computer Vision and Pattern Recognition, 2015. You can find a version of this at https://arxiv.org/abs/1512.03385.
- **Selective search:** *Selective Search for Object Recognition* by J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers and A. W. M. Smeulders International Journal of Computer Vision September 2013, Volume 104, Issue 2, pp 154–171.
- **R-CNN:** *Rich feature hierarchies for accurate object detection and semantic segmentation*, by R. Girshick, J. Donahue, T. Darrell and J. Malik, IEEE Conf. on Computer Vision and Pattern Recognition, 2014. You can find a version of this at https://arxiv.org/abs/1311.2524.
- **Fast R-CNN:** *Fast R-CNN*, by Ross Girshick, IEEE Int. Conf. on Computer Vision (ICCV), 2015, pp. 1440-1448. You can find a version of this at https://www.cv-foundation.org/openaccess/content_iccv_2015/html/Girshick_Fast_R-CNN_ICCV_2015_paper.html.
- **Faster R-CNN:** *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, by Shaoqing Ren, Kaiming He, Ross Girshick and Jian Sun, Advances in Neural Information Processing Systems 28 (NIPS 2015). You can find a version of this at http://papers.nips.cc/paper/5638-faster-r-cnn-towards-real-time-obj pdf.
- **YOLO:** *You Only Look Once: Unified, Real-Time Object Detection*, by Joseph Redmon, Santosh Divvala, Ross Girshick and Ali Farhadi, Proc Computer Vision and Pattern Recognition, 2016. You can find a version of this at https://www.cv-foundation.org/openaccess/content_cvpr_2016/papers/Redmon_You_Only_Look_CVPR_2016_paper.pdf. There's a home page at https://pjreddie.com/darknet/yolo/.

18.4 YOU SHOULD

18.4.1 remember these terms:

18.4.2 remember these facts:

18.4.3 be able to:

- Run an image classifier in your chosen environment.
- Explain how current object detectors work.
- Run an object detector in your chosen environment.

PROBLEMS

18.1. Modify the backpropagation algorithm to deal with directed acyclic graphs like that of 18.3. Note the layers are numbered there, and I will denote the parameters of the i'th layer as θ_i.

(a) The first step is to deal with layers that have one output, but two inputs. If we can deal with two inputs, we can deal with any number. Write the inputs of layer 6 as $\mathbf{x}_1^{(6)}$ and $\mathbf{x}_2^{(6)}$. Write $J_{\mathbf{o}^{(6)};\mathbf{x}_i^{(6)}}$ for the Jacobian of the output with respect to the i'th input. Explain how to compute $\nabla_{\theta_4}\mathcal{L}$ using this Jacobian.

(b) Layer 2 has two outputs. Write the outputs $\mathbf{o}_1^{(2)}$ and $\mathbf{o}_2^{(2)}$. Write $J_{\mathbf{o}_i^{(2)};\mathbf{x}^{(2)}}$ for the Jacobian of the i'th output with respect to its input. Explain how to compute $\nabla_{\theta_2}\mathcal{L}$ using this Jacobian (and others!).

(c) Can you backpropagate through a layer has two inputs *and* two outputs?

(d) What goes wrong with backpropagation when the computation graph has a cycle?

PROGRAMMING EXERCISES

General remark: *These exercises are suggested activities, and are rather open ended. It will be difficult to do them without a GPU. You may have to deal with some fun installing software environments, etc. It's worthwhile being able to do this, though.*

Minor nuisance: *At least in my instance of the ILSVRC-2012 validation dataset, some images are grey level images rather than RGB. Ensure the code you use turns them into RGB images by making the R, G and B channel the same as the original intensity channel, or funny things can happen.*

18.2. Download a pretrained VGGNet-19 image classifier for your chosen programming framework.

(a) Run this classifier on ILSVRC-2012 validation dataset. Each image needs to be reduced to 224×224 block. Do this by first resizing the image uniformly so that the smallest dimension is 224, then cropping the right half of the image. Ensure that you do whatever pre-processing your instance of VGGNet-19 requires on this crop (this should be subtracting the mean RGB at each pixel from each pixel; i.e. follow the procedure on page 431, but don't divide by the standard deviation). In this case, what is the top-1 error rate? What is the top-5 error rate?

(b) Now investigate the effect of multiple crops. For each image in the validation dataset, crop to 224×224 for five different crop windows. One of these is centered in the image; the other four are obtained by place a corner of the crop window at each corner of the image respectively. Ensure that you do whatever pre-processing your instance of VGGNet-19 requires on each crop (this should be subtracting the mean RGB at each pixel from each pixel; i.e. follow the procedure on page 431, but don't divide by the standard deviation). Pass each crop through the network, then average the predicted class posteriors and use that score. In this case, what is the top-1 error rate? What is the top-5 error rate?

18.3. Download a pretrained ResNet image classifier for your chosen programming framework.

(a) Run this classifier on ILSVRC-2012 validation dataset. Each image needs to be reduced to 224×224 block. Do this by first resizing the image uniformly so that the smallest dimension is 224, then cropping the right half

of the image. Ensure that you do whatever pre-processing your instance of ResNet requires on this crop (this should be subtracting the mean RGB at each pixel from each pixel; i.e. follow the procedure on page 431, but don't divide by the standard deviation). In this case, what is the top-1 error rate? What is the top-5 error rate?

(b) Now investigate the effect of multiple crops. For each image in the validation dataset, crop to 224×224 for five different crop windows. One of these is centered in the image; the other four are obtained by place a corner of the crop window at each corner of the image respectively. Ensure that you do whatever pre-processing your instance of ResNet requires on each crop (this should be subtracting the mean RGB at each pixel from each pixel; i.e. follow the procedure on page 431, but don't divide by the standard deviation). Pass each crop through the network, then average the predicted class posteriors and use that score. In this case, what is the top-1 error rate? What is the top-5 error rate?

18.4. Download both a pretrained ResNet image classifier and a pretrained VGG-19 for your chosen programming framework. For each image in the validation dataset, use a center crop to 224×224 Ensure that you do whatever pre-processing your instances require. Record for every image the true class, the class predicted by ResNet, and the class predicted by VGG-19.

(a) On average, if you know VGG-19 predicted the label correctly or not, how accurately can you predict whether ResNet gets the label right? Answer this by computing $P(\text{ResNet right}|\text{VGG right})$ and $P(\text{ResNet right}|\text{VGG wrong})$ using your data.

(b) Both networks are quite accurate, even for top-1 error. This means that their errors must be correlated, because each gets most examples right. We would like to know whether the result of the previous subexercise is due to this effect, or something else. Write the VGG-19 error rate as v, and the ResNet error rate as r. Write $\mathbf{v}$ for a 50, 000 dimensional binary vector, with v 1's, where the entries are IID samples from a Bernoulli distribution with mean v. This is a model of randomly distributed errors with the same error rate as VGG-19. A similar $\mathbf{r}$ models random errors for ResNet. Draw 1000 samples of $(\mathbf{v}, \mathbf{r})$ pairs, and compute the mean and standard error of $P(r_i = 0|v_i = 1)$ and $P(r_i = 1|v_i = 0)$. Use this information to determine whether ResNet "knows" something about VGG-19 errors.

(c) What could cause the effect you see?

(d) How are errors distributed across categories?

(e) **Hard!** *(but interesting).* Obtain instances of several different ImageNet classification networks. Investigate the pattern of errors. In particular, for images that one instance mislabels, do other instances mislabel it as well? If so, how many different labels are used in total? I have found suprisingly strong agreement between instances that mislabel an image (i.e. if network A thinks an image of a `dog` is a `cat`, *and* network B gets the same image wrong as well, then network B will likely think it's a `cat`, too).

18.5. Choose 10 ImageNet classes. For each class, download 50 example images of items that belong to those classes from an internet image source (images.google.com or images.bing.com are good places to start; query using the name of the category).

(a) Classify these images into the ImageNet classes using a pretrained net-

work. What is the top-1 error rate? What is the top-5 error rate?

(b) Compare the results of this experiment with the accuracy on the validation set for these classes. What is going on?