Chapter 3
Deep Neural Networks for Computer Vision

Computer vision problems are related to the understanding of digital images, video, or similar inputs such as 3D point clouds, solving problems such as image classification, object detection, segmentation, 3D scene understanding, object tracking in videos, and many more. Neural approaches to computer vision were originally modeled after the visual cortex of mammals, but soon became a science of their own, with many architectures already developed and new ones appearing up to this day. In this chapter, we discuss the most popular architectures for computer vision, concentrating mainly on ideas rather than specific models. We also discuss the first step towards synthetic data for computer vision: data augmentation.

3.1 Computer Vision and Convolutional Neural Networks

Computer vision is one of the oldest and most important subfields of artificial intelligence. In the early days of AI, even leading researchers believed that computer vision might prove to be easy enough: Seymour Papert, one of the fathers of AI, initially formulated several basic computer vision problems as a student project in "an attempt to use our summer workers effectively" [654] (see also Section 5.1). But it soon became apparent that computer vision is actually a much more ambitious endeavour, and despite decades of effort and progress the corresponding problems are still not entirely solved.

One of the most important advances in the study of the visual cortex was made by David H. Hubel and Torsten N. Wiesel who, in their Nobel Prize-winning collaboration, were the first to analyze the activations of individual neurons in the visual cortex of mammals, most famously cats [376, 377, 925]. They studied the early layers of the visual cortex and realized that individual neurons on the first layer of processing react to simple shapes while neurons of the second layer react to certain combinations of first layer neurons. For example, one first layer neuron might

S. I. Nikolenko, *Synthetic Data for Deep Learning*, Springer Optimization and Its Applications 174, https://doi.org/10.1007/978-3-030-75178-4_3

react to a horizontal line in its field of view (called a *receptive field*, a term that also carried over to artificial intelligence), and another first layer neuron might be activated by a vertical line. And if these two neurons are activated at the same time, a second layer neuron might react to a cross-like shape appearing in its receptive field by implementing something close to a logical AND (naturally, I'm simplifying immensely but that's the basic idea). In other words, first layer neurons pick up on very simple features of the input, and second layer neurons pick up on combinations of first layer neurons. Hubel and Wiesel were wise enough not to go much farther than the first two layers because signal processing in the brain becomes much more complicated afterwards. But even these initial insights were enough to significantly advance artificial intelligence...

The basic idea of *convolutional neural networks* (CNN) is quite simple. We know, e.g., from the works of Hubel and Wiesel that a reasonable way to process visual information is to extract simple features and then produce more complicated features as combinations of simple ones. Simple features often correspond to small receptive fields: for instance, we might want a first layer neuron to pick up a vertical gradient in a window of size 5×5 pixels. But then this feature extraction should be applied equally to *every* 5×5 window, that is, instead of training a neural network to perform the same operation for each window across, say, a 1024×1024 image we could apply *the same learnable transformation* to every window, with shared weights. This idea works as a structural regularizer, saving an immense number of weights in the CNN compared to an equivalent fully connected network. Mathematically, this idea can be expressed as a convolution between the input and the small learnable transformation, hence the name.

Figure 3.1 illustrates the basic idea of a convolutional network: a 5×5 input image is broken down into 3×3 windows, and each window is passed through the same small neural network, getting a vector of features as a result. After this transformation, the 5×5 image becomes a 3×3 output in terms of width and height.

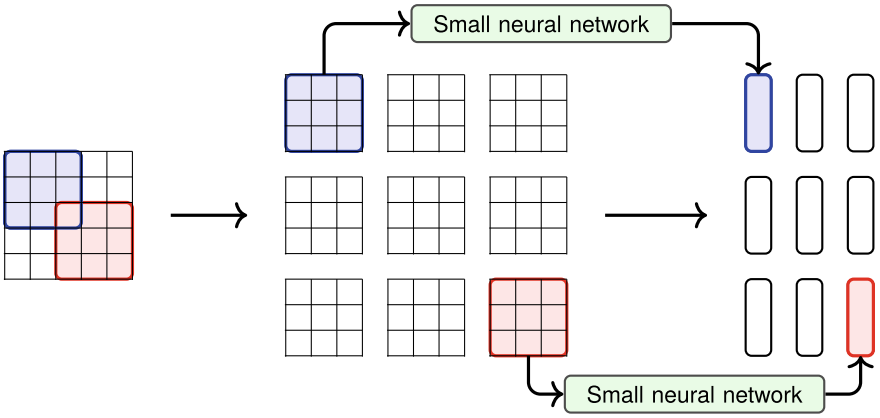


Fig. 3.1 The basic idea of a convolutional neural network; blue and red colors follow the transformations of two 3×3 windows, and the same small neural network is applied to the other windows as well.

A *convolutional layer* is actually just one way to implement this idea, albeit the most popular one by far. It is a layer defined by a set of learnable *filters* (or *kernels*) that are convolved across the entire input tensor. Convolutions can come in any dimension, but the most popular and intuitive ones for computer vision are two-dimensional convolutions, so we will use them in our examples. In this case, the input is a three-dimensional tensor width × height × channels (a grayscale image has one channel, a color image three, and intermediate representations inside a neural network can have arbitrarily many), and the convolution is best thought of as a four-dimensional tensor of dimension

$$\text{input channels} \times \text{width} \times \text{height} \times \text{output channels}.$$

Figure 3.2 shows a toy numerical example of a convolutional layer consisting of a linear convolution with a $3 \times 3 \times 2$ tensor of weights and a ReLU activation, applied to a $5 \times 5 \times 1$ input image.

For the first "real" example, let us consider the Sobel operator, a classical computer vision tool dating back to 1968 [809]. It is a discrete computation of the image gradient, used for edge detection in classical computer vision. For our example, we note that the main components of the Sobel operator are two 3×3 convolutions with matrices

$$S_x = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}, \qquad S_y = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

Basically, S_x shows the horizontal component of the image gradient and S_y shows the vertical component.

If we take an image such as a handwritten digit from the MNIST dataset as shown in Fig. 3.3a, and apply a convolution with matrix S_x, we get the result shown in Fig. 3.3b. The result of convolving with matrix S_y is shown in Fig. 3.3c. In this example, we see how convolutions with small matrices give rise to meaningful feature

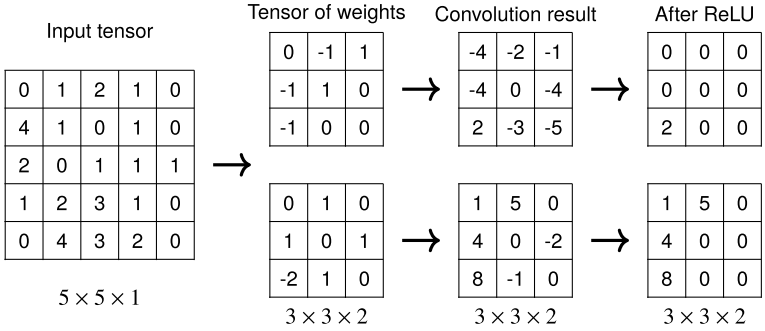


Fig. 3.2 Sample application of a convolutional layer consisting of a linear convolution with a $3 \times 3 \times 2$ tensor of weights and a ReLU activation.

Fig. 3.3 Sample application of convolutions: (a) original handwritten digit; (b) convolution with matrix S_x, the horizontal component of the image gradient; (c) convolution with matrix S_y, the vertical component of the image gradient.

extraction: the meaning of the features shown in Fig. 3.3b, c is quite clear, and they indeed reflect the variation of the input image's pixel intensities in the corresponding direction. Note that in a neural network, both convolutions would be treated as a single tensor of dimension $3 \times 3 \times 2$, and the result of applying it to an image of size 28×28 would be a tensor with two channels (feature maps) shown in Fig. 3.3b, c. Note that in this example, the width and height of the output stay at 28 instead of being reduced by 1 on every side because the results are computed with *padding*, i.e., an extra row and column of zeroes is added on every side of the input; this is a common trick in convolutional networks used when it is desirable to leave the input size unchanged.

In a trainable neural network, weights in the matrices S_x and S_y would not be set in advance but would represent weights that need to be learned with gradient descent, as we discussed in the previous chapter. Actually, the first introduction of convolutions to artificial neural networks happened a long time ago, when even backpropagation had not been universally accepted. This was the *Neocognitron* developed by Kunihiko Fukushima in the late 1970s [248–250]. The *Neocognitron* was a pioneering architecture in many respects: it was a deep neural network in times when deep networks were almost nonexistent, it had feature extraction from small windows of the input—precisely the idea of convolutional networks—it was training in an unsupervised fashion, learning to recognize different kinds of patterns presented, and it actually already had ReLU activations—all this in the 1970s! The *Neocognitron* is widely regarded as a predecessor to CNNs, and although it did take a few years to adapt all these ideas into modern CNN architectures, they actually appeared in the 1980s pretty much in their modern form.

In a classical convolutional network such as *LeNet* [501], convolutional layers are usually interspersed with nonlinear activation functions (applied componentwise) and *pooling* layers that reduce the dimension. The most popular is the *max-pooling* layer that does not have any trainable parameters and simply covers the input tensor with windows (often 2×2) and chooses the largest value of every feature in each

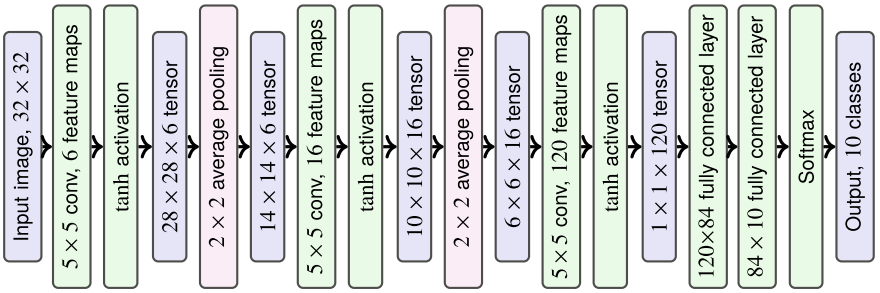


Fig. 3.4 The *LeNet*-5 architecture [501].

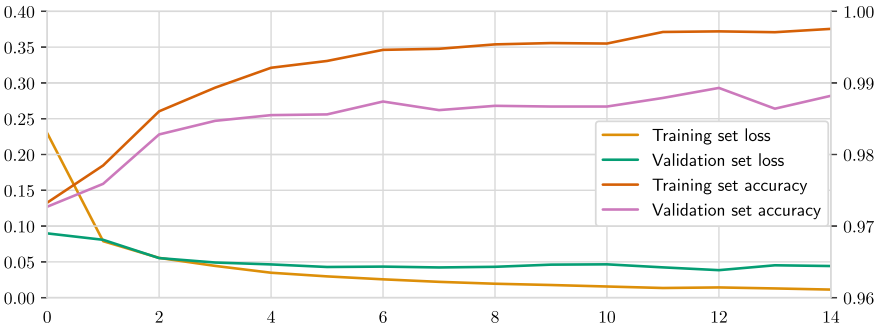


Fig. 3.5 The *LeNet*-5 training process on MNIST dataset of handwritten digits.

window. The basic intuition is that we want the features to correspond to certain properties that extend from smaller to larger windows; for example, if there is a cat present in a 128×128 window in the image, there is also a cat in every 256×256 window containing it. Max-pooling also induces a lot of sparsity that helps keep the computations more efficient.

For an extended example, let us implement and study the (slightly modified) *LeNet*-5 network operating on 32×32 grayscale images (we will be using MNIST handwritten digits), a simple convolutional architecture shown in Figure 3.4. In the figure, layers that perform transformations are shown as rectangles filled in green, and dimensions of current feature maps are shown as rectangles filled in blue. As you can see, each 5×5 convolution reduces the width and height of the input tensor by 4 because there is no padding here, and each 2×2 pooling layer (average pooling in the case of *LeNet*) halves the input dimensions.

Figure 3.5 shows the learning process of this network, trained on the MNIST dataset with *Adam* optimizer and batch size 32. As you can see, the loss function (average cross-entropy between the correct answers and network predictions) on the training set decreases steadily, but the loss function on the held-out validation set is far from monotone. The best result on the validation set is achieved in this experiment

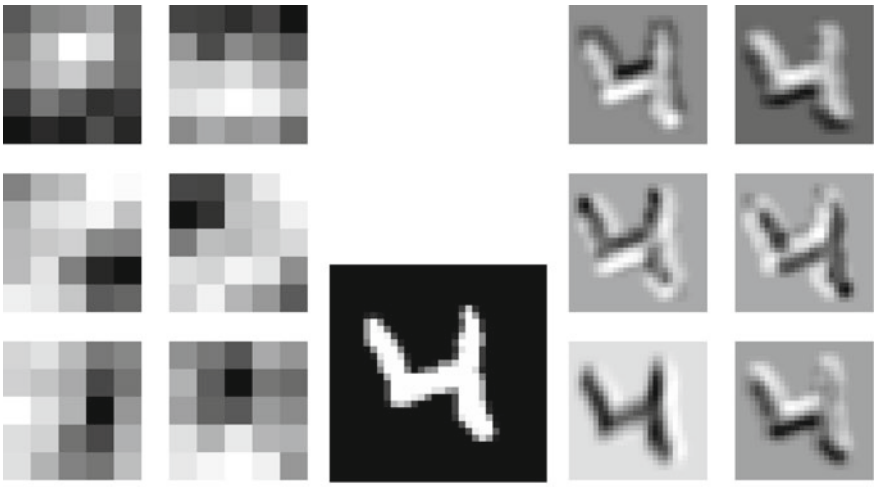


Fig. 3.6 A view into the first layer of *LeNet*-5: (a) weights of the six convolutions; (b) sample input image; (c) activations produced by convolutions from (a) on the image from (b).

(specific results might change after a restart with random re-initialization of the weights) after 12 epochs of training.

Figure 3.6 shows the first layer of the resulting trained network. It shows the weights of the six 5×5 convolutions trained on the first layer in Fig. 3.6a and the results of applying them to a sample digit shown in Fig. 3.6b (the same as in Fig. 3.3, only padded with zeroes to size 32×32) are shown in Fig. 3.6c. You can see that the first layer has learned to extract simple geometric features, in many ways similar to the image gradients shown in Fig. 3.3.

Modern networks used in computer vision employ very similar convolutional layers. They usually differ from *LeNet* in that they use ReLU activation functions or its variations rather than sigmoidal activations (recall the discussion in Section 2.1). The ReLU nonlinearity was re-introduced into deep learning first for Boltzmann machines in [616] and widely popularized in deep convolutional networks by *AlexNet* (see Section 3.2 below).

Over the last decade, CNNs have been dominating computer vision and have been rising in popularity in many other fields of machine learning. For example, in 2014–2016 one-dimensional CNNs were becoming increasingly crucial for natural language processing (NLP), supplementing and even replacing recurrent networks [423, 1010]; after 2017, the best results in NLP were produced by architectures based on self-attention such as Transformer, especially BERT and GPT families [192, 697, 891]. But BERT-like models are still often used to pretrain word embeddings, and embeddings are then processed by CNNs and/or RNNs to solve downstream tasks.

However, there still remain valid criticisms even for the basic underlying idea of convolutional architectures. Two of the most important criticisms deal with the

lack of translational invariance and *loss of geometry* along a deep CNN. Lack of translational invariance means that units in a CNN that are supposed to produce a given feature (say, recognize a cat's head on a photo) might not activate if the head is slightly moved or rotated. In machine learning practice, translational invariance in CNNs is usually achieved by extensive data augmentation that always includes simple geometric transformations such as re-cropping the image, rescaling by a factor close to 1, and so on (we will discuss augmentations in detail in Section 3.4). However, achieving full translational invariance by simply extending the dataset is far from guaranteed and appears extremely wasteful: if we are dealing with images we already know that translational invariance should be in place, why should we learn it from scratch for every single problem in such a roundabout way?

The "loss of geometry" problem stems from the fact that standard convolutional architectures employ pooling layers that propagate information about low-level features to high-level feature maps with smaller resolutions. Therefore, as the signal travels from bottom to top layers, networks progressively lose sight of the exact locations where features have originated. As a result, it is impossible for a high-level feature to activate on specific geometric interrelations between low-level features, a phenomenon sometimes called the "Picasso problem": a convolutional feature can look for two eyes, nose, and mouth on a face but cannot ensure that these features are indeed properly positioned with respect to each other. This is because pooling layers, while they are helpful to reduce the redundancy of feature representation in neural networks, prevent overfitting, and improve the training process, and at the same time represent a fixed and very crude way of "routing" low-level information to high-level features.

These two problems have been pointed out already in 2014 by Geoffrey Hinton [343]. An attempt to alleviate these problems has led Hinton to develop a new approach to architectures that perform feature composition: *capsule networks*. In a capsule network, special (trainable) routing coefficients are used to indicate which low-level features are more important for a given high-level feature, and the features (capsules) themselves explicitly include the orientations and mutual positions of features and explicitly estimate the likelihood of the resulting composite feature [247, 349, 452, 748]. As a result, translational invariance is built in, routing is dynamic, capsule networks have much fewer parameters than CNNs, and the entire process is much more similar to human vision than a CNN: capsules were designed with cortical columns in mind.

However, at present it still appears too hard to scale capsule networks up to real-world problems: computational tricks developed for large-scale CNNs do not help with capsule networks, and so far they struggle to scale far beyond MNIST and similar-sized datasets. Therefore, all modern real-life computer vision architectures are based on CNNs rather than capsules or other similar ideas, e.g., other equivariant extensions such as spherical CNNs [163] or steerable CNNs [164], and applications of capsule networks are only beginning to appear [227]. Thus, we will not be discussing these alternatives in detail, but I did want to mention that the future of computer vision may hold something very different from today's CNNs.

3.2 Modern Convolutional Architectures

In this section, we give an overview of the main ideas that have brought computer vision to its current state of the art. We will go over a brief history of the development of convolutional architectures during the deep learning revolution, but will only touch upon the main points, concentrating on specific important ideas that each of these architectures has brought to the smörgåsbord of CNNs that we have today. For a more detailed introduction, we refer to [153, 225, 631] and other sources (Fig. 3.7).

MCDNN. The deep learning revolution in computer vision started during 2010–2011, when recent advances in deep learning theory and the technology of training and using neural networks on highly parallel graphical processors (GPUs) allowed training much deeper networks with much more units than before. The first basic problem that was convincingly solved by deep learning was image classification. In 2011, a network by Dan Cireşan from Jürgen Schmudhuber's group won a number of computer vision competitions [159]. In particular, this network was the first to achieve superhuman performance in a practically relevant computer vision problem, achieving a mere 0.56% error in the IJCNN Traffic Sign Recognition Competition, while the average human error on this dataset was 1.16% [160].

Architecturally, Ciresan's network, called *Multi-Column Deep Neural Network* (MCDNN), is a committee of deep convolutional networks with max-pooling. It showcases several important ideas:

- MCDNN uses a basic convolutional architecture very similar to the *LeNet* family of networks (so we do not show a separate figure for it), but it was one of the first to consistently use max-pooling instead of average-pooling or other variations;
- the architecture contains several identical networks trained on differently preprocessed inputs, where preprocessing variations include different combinations of color normalization and contrast adjustment; thus, MCDNN was already showing

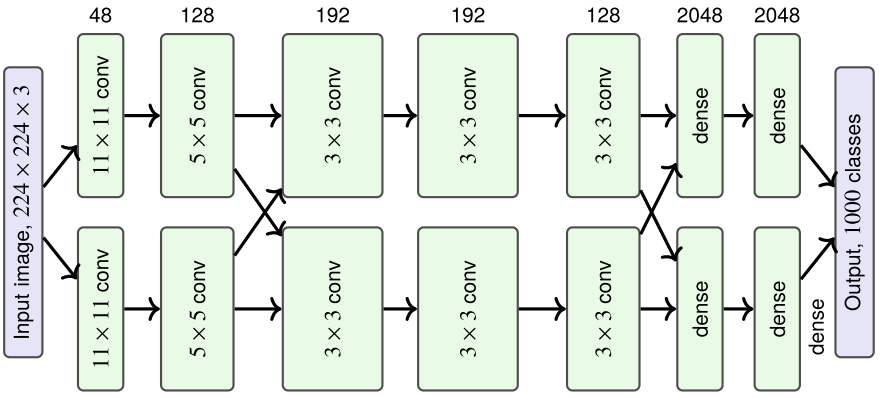


Fig. 3.7 The *AlexNet* architecture [477].

the power of *data augmentation* for computer vision, a theme that remains crucial to this day and that represents one of the motivations for synthetic data.

AlexNet. However, MCDNN operated on very small images, cutting out traffic sign bounding boxes of size 48×48 pixels. The development of large-scale modern architectures that could deal with higher resolution images started with *AlexNet* [477], a network developed by Alex Krizhevsky in Prof. Hinton's group (see Fig. 3.7 for an illustration). With 8 trainable layers, *AlexNet* became one of the first successful truly deep convolutional networks. It was introduced at the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012, where *AlexNet* beat all competitors with an unprecedented margin: two submitted versions of *AlexNet* had test set errors (measured as classification accuracy for top-5 guesses) about 15–16%, while the nearest competitor could only achieve an error of 26%[1]! Architecturally, *AlexNet* again introduced several new ideas:

- it introduced and immediately popularized ReLU activations as nonlinearities used in convolutional layers; previously, tanh activations had been most often used in convolutional networks;
- it emphasized the crucial role of data augmentation in training neural networks for computer vision problems; we will discuss the case of *AlexNet* in detail in Section 3.4;
- it was one of the first large-scale networks to consistently use dropout for additional regularization;
- finally, it was one of the first neural networks to feature model parallelization: the model was distributed between two GPUs; back in 2012, it was a real engineering feat, but since then it has become a standard feature of deep learning frameworks such as *PyTorch* or *Tensorflow*.

AlexNet's resounding success marked the start of a new era in computer vision: since 2012, it has been dominated by convolutional neural architectures. CNNs have improved and defined state of the art in almost all computer vision problems: image classification, object detection, segmentation, pose estimation, depth estimation, object tracking, video processing, and many more. We will talk in more detail about object detection and segmentation architectures in Section 3.3. For now, the important part is that they all feature a convolutional backbone network that performs feature extraction, often on several layers simultaneously: bottom layers (nearest to input) of a CNN extract local features and can produce high-resolution feature maps, while features extracted on top layers (nearest to output) have large receptive fields, generalize more information, and thus can learn to have deeper semantics, but lose some of the geometry along the way (we have discussed this problem above in Section 3.1).

VGG. The next steps in deep CNN architectures were also associated with the ILSVRC challenge: for several years, top results in image classification were marked by new important ideas that later made their way into numerous other architectures as well. One of the most fruitful years was 2014, when the best ImageNet classification

[1] http://image-net.org/challenges/LSVRC/2012/results.html.

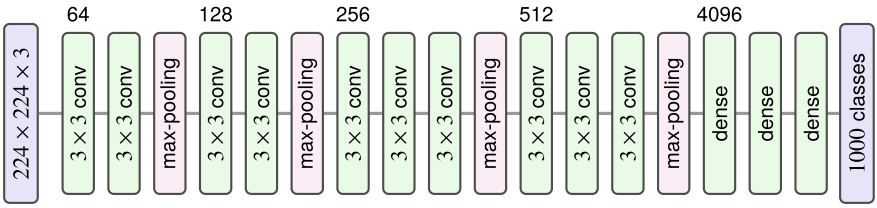


Fig. 3.8 VGG: decomposing large convolutions [802].

was achieved by the VGG network by Simonyan and Zisserman [802]; the name originates from the Visual Geometry Group in the University of Oxford. The main idea that defined the VGG family is that individual convolutions in a CNN virtually never need to be large: a 5×5 convolution can be expressed as a composition of two 3×3 convolutions without any pooling or nonlinearities in between, a 7×7 convolution is just three layers of 3×3 convolutions, and so on. Figure 3.8 shows the first successful network from the VGG family; note how max-pooling layers come after groups of two or three convolutional layers, thus decomposing larger convolutions. This results in much deeper networks with fewer weights, serving as additional regularization and at the same time making training and inference more efficient. Later architectures also experimented with expressing $n \times n$ two-dimensional convolutions as compositions of $1 \times n$ and $n \times 1$ one-dimensional convolutions, and this trick is also common in modern CNN architectures.

Inception and GoogLeNet. In the same year, Google presented *GoogLeNet*, a network by Szegedy et al. that won the object detection track of ILSVRC 2014 [836]. Together with a precursor work on "network-in-network" architectures by Lin et al. [522], it had three important ideas that have stayed with us ever since: Inception modules, 1×1 convolutions, and auxiliary classifiers.

First, "network-in-network" architectures take the basic idea of convolutional networks—applying the same simple transformation to all local windows over the input—and run with it a bit further than regular CNNs. Instead of just using a matrix of weights, they design special architectures for the "simple transformation" (not so simple anymore), so that a single layer is actually applying a whole neural network to each window of the input, hence the name. The architecture of these small networks from [836], called *Inception modules*, is shown in Fig. 3.9. Since then, there have been plenty of modifications, including Inception v2 and v3 [837] and later combinations of Inception with ResNet (see below).

Second, 1×1 convolutions play an important part in all variations of network-in-network modules. At first glance, it may appear that 1×1 convolutions are pointless. However, while they indeed do not collect any new features from neighboring pixels, they provide additional expressiveness by learning a (nonlinear, otherwise it is pointless indeed) transformation on the vector of features in a given pixel. In practice, this is usually needed to change the dimension of the feature vector, often reducing it before performing more computationally demanding transformations.

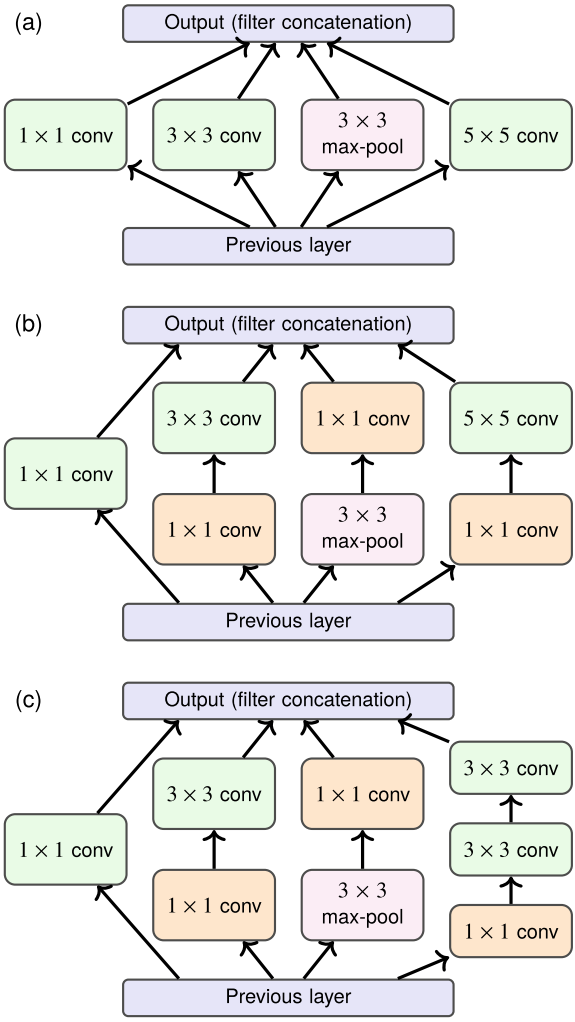


Fig. 3.9 Inception modules: (a) basic "naive" Inception v1 module [836]; (b) Inception v1 module with dimension reductions via 1×1 convolutions [836]; (c) sample Inception v2 module [837].

For example, a 3×3 convolution that maps a 512-dimensional vector into a 512-dimensional vector has $512 \times 3 \times 3 \times 512 = 9 \cdot 2^{18} \approx 2.36$M weights. But if we first apply a 1×1 convolution to reduce the dimension to 64 and then map the result back into dimension 512, we add two convolutions with $512 \times 1 \times 1 \times 64 = 2^{15} = 32768$ weights each but reduce the 3×3 convolution to $64 \times 3 \times 3 \times 64 = 9 \cdot 2^{12}$ weights, for a total of $2 \cdot 2^{15} + 9 \cdot 2^{12} \approx 102$K weights, a reduction by a factor of more than 20! The additional approximation that this kind of dimensionality reduction implies usually does not hurt and may even serve as additional structural regularization.

This idea has been widely used in architectures that try to minimize the memory footprint or latency of convolutional neural models. Figure 3.9b shows the Inception v1 module with 1×1 convolutions that perform these dimension reductions, and Figure 3.9c shows how Inception v2 has modified this module with the VGG basic idea of decomposing larger convolutions into compositions of 3×3 convolutions [837]. We do not show all variations of Inception modules here and refer to [836, 837] for more details.

Third, *GoogLeNet* is a deep network, it has 22 layers with trainable parameters, or 27 if you count pooling layers. When training by gradient descent, *GoogLeNet* faces problems that we discussed in Section 2.4 in relation to deep neural networks in general: error propagation is limited, and when top layers reach saturation it becomes very hard for bottom layers to train. To overcome this problem, Szegedy et al. [836] proposed to use auxiliary classifiers to help the loss gradients reach bottom layers. The *GoogLeNet* architecture (see Fig. 3.10) has two auxiliary classifiers that have separate classification heads (shallow networks ending in a classification layer). The loss functions are the same (binary cross-entropy classification loss), and they are simply added together with the main loss function to form the objective function for the whole network:

$$\mathcal{L}^{\text{GoogLeNet}} = \mathcal{L}^{\text{MainBCE}} + \alpha_1 \mathcal{L}^{\text{AuxBCE}_1} + \alpha_2 \mathcal{L}^{\text{AuxBCE}_2}.$$

The α coefficient was initialized to 0.3 and gradually reduced during training. This trick was intended to speed up the training of bottom layers on early stages of training and improve convergence, but Szegedy et al. report that in practice, convergence rate did not improve significantly, but the final performance of the network was better, so auxiliary classifiers served more like a regularizer.

ResNet. Despite these promising results, auxiliary classifiers are not widely used in modern architectures. The reason is that the main problem that they had been intended to solve, problems with error propagation after the saturation of top layers, was solved in a different way that proved to be much better. A *Microsoft Research* team led by Kaiming He developed and implemented the idea of *deep residual learning* [330] that was the main driving force behind the success of *ResNet* architectures that won ILSVRC 2015 in both classification (reducing the ImageNet Top-5 error rate to 3.5%) and object detection (with the Faster R-CNN detection architecture that we will discuss below in Section 3.3).

The basic structure of *ResNet* is simple: it is a composition of consecutive layers, and each of them is usually simply a convolutional layer, perhaps with batch normalization on top. The main difference is that in a residual unit, the layer that computes a function $F(\mathbf{x})$ for some input $\mathbf{x}$ (shown schematically in Fig. 3.11a) is supplemented with a direct residual connection that goes around the layer, so that the overall function that produces the kth layer output, denoted as $\mathbf{y}^{(k)}$, from the input vector $\mathbf{x}^{(k)}$ is computed as

$$\mathbf{y}^{(k)} = F(\mathbf{x}^{(k)}) + \mathbf{x}^{(k)},$$

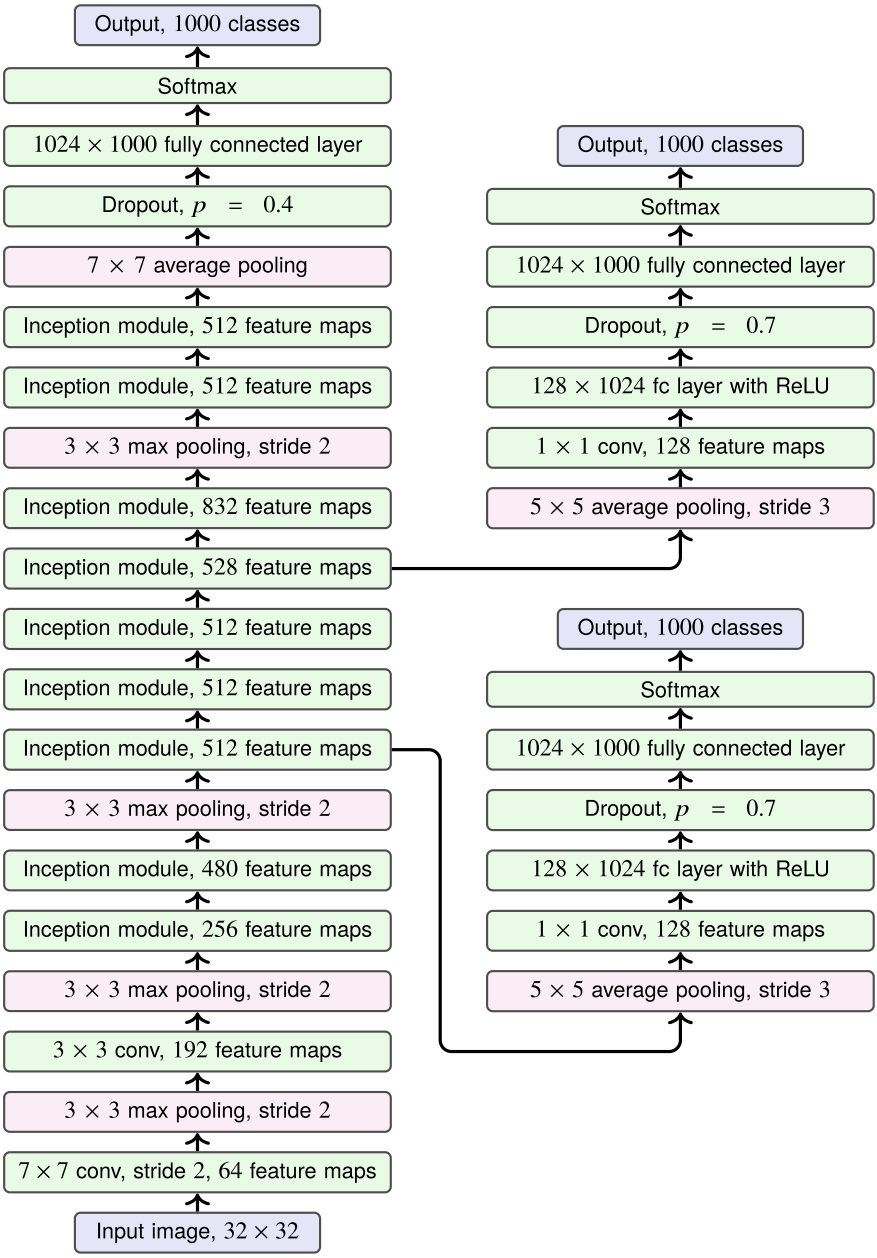


Fig. 3.10 The *GoogLeNet* general architecture.

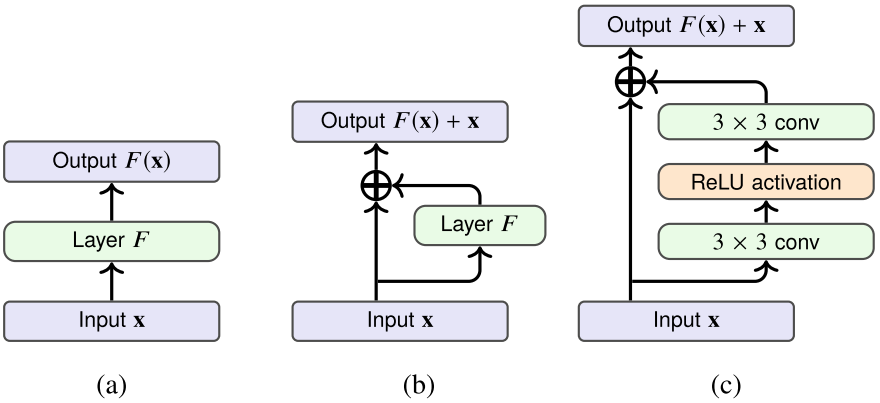


Fig. 3.11 Deep residual learning: (a) schematics of a simple layer; (b) schematics of a layer with a residual connection; (c) sample ResNet layer from [330].

where $\mathbf{x}^{(k)}$ is the input vector of the kth layer, $F(x)$ is the function that the layer computes, and $\mathbf{y}^{(k)}$ is the output of the residual unit that will later become $\mathbf{x}^{(k+1)}$ and will be fed to the next layer in the network (see Fig. 3.11b).

The name comes from the fact that if the layer as a whole is supposed to approximate some function $H(\mathbf{x})$, it means that the actual neural layer has to approximate the *residual*, $F(\mathbf{x}) \approx H(\mathbf{x}) - \mathbf{x}$; this does not complicate the problem for $F(\mathbf{x})$ much (if at all), but at the same time provides a direct way for the gradient to flow around $F(\mathbf{x})$. Now

$$\frac{\partial \mathbf{y}^{(k)}}{\partial \mathbf{x}^{(k)}} = 1 + \frac{\partial F(\mathbf{x}^{(k)})}{\partial \mathbf{x}^{(k)}},$$

and even if the layer F becomes completely saturated, its near-zero derivatives will not hinder training: the gradient will simply flow down to the previous layer unchanged.

Residual learning was not a new idea: it is the same *constant error carousel* idea that had been used in recurrent architectures for a long time, in particular in the famous long short-term memory (LSTM) architectures developed in the late 1990s by Hochreiter, Gers, and Schmidhuber [273, 350]. A recurrent network is, in essence, a very deep network by default (consider its computational graph when unrolled along the entire input sequence), and the same phenomenon leads to either exploding or vanishing gradients that effectively limit the propagation of information ("memory") in recurrent networks. The constant error carousel is precisely the idea of having a "direct path" for the gradient to flow.

However, He et al. were the first to apply this idea to "unrolled" deep convolutional networks, with great success. Note that a comparison of several residual architectures performed in [331] shows that the best results are achieved with the simplest possible residual connections: it is usually best to leave the direct path as free from any transformations (such as nonlinearities, batch normalizations, and the

like) as possible. It even proved to be a bad idea to use control gates that could potentially learn to "open" and "close" the path around the layer $F(\mathbf{x})$, an idea that had been successful in LSTM-like recurrent architectures. Figure 3.11c shows a simple sample residual layer from [330], although, of course, many variations on this idea have been put forward both in the original paper and subsequent works.

Architecturally, this has led to the possibility of training very deep networks. Kaiming He coined the term "revolution of depth": VGG had 19 trainable layers, GoogLeNet had 22, but even the very first version of *ResNet* contained 152 layers. It is still a popular feature extraction backbone, usually referred to as *ResNet-152*, with a popular smaller variation *ResNet-101* with 101 layer (there is really neither space nor much sense in presenting the architectures of deep residual networks in the graphical form here). Theoretically, residual connections allow to train networks with hundreds and even thousands of layers, but experiments have shown that there is no or very little improvement in performance starting from about 200 layers.

Some of the best modern convolutional feature extractors result from a combination of the network-in-network idea coming from *Inception* and the idea of residual connections. In 2016, Szegedy et al. [835] presented *Inception-v4* and several versions of *Inception ResNet* architectures with new architectures for both small network units and the global network as a whole. The resulting architectures are still among the best feature extractors and often serve as backbones for object detection and segmentation architectures.

Striving for efficiency: MobileNet, SqueezeNet, and others. The very best results in basic problems such as image classification are achieved by heavy networks such as the *Inception ResNet* family. However, one often needs to make a trade-off between the final performance and available computational resources; even a desktop GPU may be insufficient for modern networks to run smoothly, and computer vision is often done on smartphones or embedded devices. Therefore, the need arises to develop architectures that save on the network size (memory, usually related to the number of weights) and its running time (usually depending on the number of layers) without losing much in terms of performance. Naturally, it would be great to have the best of both worlds: excellent performance and small networks. Below, we will not present the exact architectures (I believe that after giving one full example with *GoogLeNet*, a further presentation of complete architectures would only clutter the book with information that is easy to find and unnecessary to remember) but only survey the ideas that researchers have used in these architectures.

How can one save weights? We have discussed above that convolutions are a great structural regularizer: by applying the same weights across a large image, convolutions can extract features in an arbitrarily large input with a fixed and relatively small number of weights. But that's not all: convolutions themselves can also grow to be quite large.

Suppose, for instance, that you have a layer with 256 channels (a very reasonable number, on the low side even), and you want to apply a 5×5 convolution to get another 256 channels at the output. A straightforward four-dimensional convolution would have, as we discussed in Section 3.1,

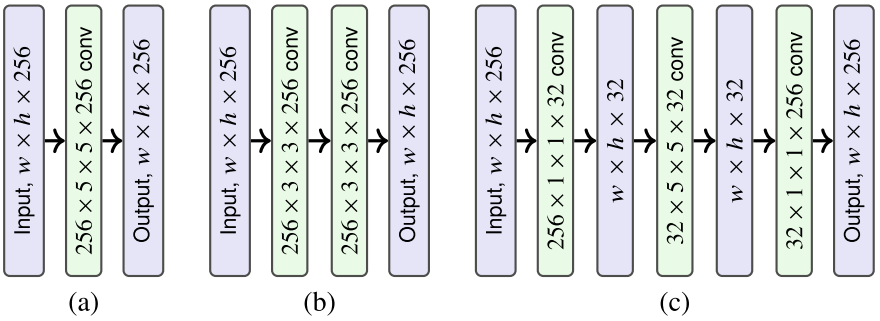


Fig. 3.12 Illustration for reducing the convolutions: (a) basic single convolution; (b) the VGG trick; (c) the bottleneck trick.

$$256 \times 5 \times 5 \times 256 = 1638400$$

weights (as shown in Fig. 3.12a). This is, of course, a big improvement compared to a feedforward layer that would have 256 × width × height × 256 weights, but it is often desirable to reduce these 1.6M weights further.

Let us briefly go through the main ideas used for this purpose in modern architectures. Note that all methods shown below, strictly speaking, are not equivalent to a single convolution, which is only natural: a network with 1.6M weights can be more expressive than a network with ten times fewer weights. Fortunately, it turns out that this added expressiveness usually does not improve performance and actually can deteriorate it due to overfitting or insufficient data to train so many weights.

First, we can use the VGG trick and represent a 5×5 convolution with a composition of two 3×3 convolutions (see Fig. 3.12b). This reduces the number of weights to $2 \times (256 \times 3 \times 3 \times 256) = 1179648$. It can be reduced even further if we represent 3×3 convolutions as compositions of 1×3 and 3×1, following [837].

Second, we can use the *bottleneck* trick that was first popularized by the *Inception* family of architectures. The 1.6 million weights in the layer above result from the fact that we have to multiply all dimensions of the convolution. But we can turn some of these multiplications into additions if we first compress the 256 channels down to a more manageable size with a 1×1 convolution, then do the spatial 5×5 convolution on the reduced tensor, again producing a tensor with a small number of channels (say 32 again), and only then expand it back with another 1×1 convolution. This method, illustrated in Fig. 3.12c, is somewhat akin to a low-rank approximation for the convolution tensor. Suppose that the bottleneck part has 32 channels; then the total number of weights in the three resulting convolutions will be

$$256 \times 1 \times 1 \times 32 + 32 \times 5 \times 5 \times 32 + 32 \times 1 \times 1 \times 256 = 41984,$$

with minor further reductions again available if we apply the VGG trick to the 5×5 convolution in the middle. At this point, we have already achieved a dramatic

reduction in network size, reducing the total number of weights by a factor of more than 28.

The bottleneck idea was presented and successfully used in the *SqueezeNet* architecture that replaced *Inception* units with *Fire* modules that have a "squeeze-then-expand" structure: first use 1×1 convolutions to reduce the number of channels and then expand them back, applying a number of different convolutions and concatenating the outputs [382].

But even that's not all! Third, we can take the bottleneck approach even further by using *depthwise separable convolutions*. The idea is now to further decompose the tensor in the middle, a $32 \times 5 \times 5 \times 32$ convolution that still has all four factors present. This convolution mixes all channels together; but what if we leave the mixing for 1×1 convolutions (after all, that's exactly what they do) and concentrate only on the spatial part? Formally speaking, we replace a single convolution with 32 separate 5×5 convolutions, each applied only to a single channel. This definitely reduces the expressiveness of the convolution in the middle since now each channel in the result has access to only one channel in the input; but since the channels can freely exchange information in the 1×1 convolution, it usually does not lead to any significant loss of performance. In our running example, we could apply this idea to the bottleneck, shaving off one of the 32 factors and getting a total of

$$256 \times 1 \times 1 \times 32 + 32 \times 5 \times 5 + 32 \times 1 \times 1 \times 256 = 17184$$

weights. Alternatively, we could just forget about the whole bottleneck idea and do 256 depthwise separable convolutions instead of one of the 1×1 convolutions and the bottleneck, getting

$$256 \times 1 \times 1 \times 256 + 256 \times 5 \times 5 = 71936$$

weights. The second approach looks worse in this case, but, first, it depends on the actual dimensions, and second, compressing all features to an exceedingly small bottleneck does tend to lose information, so if we can achieve the same result without compressing the features it might result in better performance.

Depthwise separable convolutions were introduced by Francois Chollet in [152], where he noted that a basic *Inception* module can be represented as a depthwise separable convolution that mixes subsets of channels and presented the *Xception* modules that take this idea to its logical conclusion as we have just discussed. They also became the main tool in the construction of the *MobileNet* family of networks that were designed specifically to save on memory and still remain some of the best tools for the job [358].

Neural architecture search and EfficientNet. In the survey above, basic ideas such as compressing the channels with 1×1 convolutions are easy to understand, and we can see how researchers might come up with ideas like this. A more difficult question is how to come up with actual architectures. Who and how could establish that for *GoogLeNet* you need exactly two convolutional layers in the stem followed by nine basic *Inception* modules interspersed with max-pooling in just the right way?

The actual answer is simple: there is no theorem that shows which architecture is best; you just have to come up with a wide spectrum of different architectures that "make sense" in terms of dimensions, test a lot of them in practice, and choose the one that performs best.

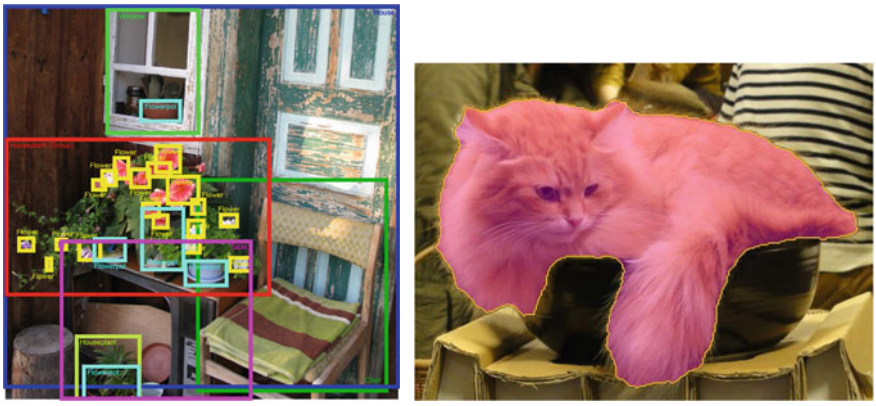
This looks suspiciously like a machine learning problem: you have the building blocks (various convolutions, pooling layers, etc.) and a well-defined objective function (say, performance on the *ImageNet* test set after the training converges). Naturally, it was not long before researchers decided to automate this process. This problem is quite naturally formulated as a *reinforcement learning* problem: while we do not have a ready-to-use dataset, we can compute the objective function on any network. But computing the objective function is quite expensive (you need to train a large model to convergence). This approach to developing neural networks is known as *neural architecture search* (NAS); I will not go into more details about it and will refer to the main sources on NAS [532, 846, 930, 1031].

In convolutional architectures, neural architecture search yielded the *EfficientNet* family, proposed in 2019 by Tan and Le [847]. They introduced the *compound scaling* method, basically generalizing all of the above discussion into a single approach that scales network width, depth, and resolution according to a set of scaling coefficients. This approach by itself already allowed to improve existing architectures, but even more importantly, this generalization allowed the authors to formulate the problem of finding the best network in an efficient parameter space. The resulting networks outperformed all predecessors, setting a whole new Pareto frontier for the performance/efficiency trade-off.

To sum up, in this section we have seen the main ideas that constitute the state of the art in convolutional architectures. But note that everything that we have been talking about could be formulated in terms of "training on *ImageNet*", that is, all networks mentioned above solve the image classification problem. But this is only one problem in computer vision, and hardly even the most important one... how do we solve object detection, segmentation, and all the rest? Let's find out.

3.3 Case Study: Neural Architectures for Object Detection

In subsequent chapters, we will consider the use of synthetic data for computer vision. We have seen above which convolutional architectures are regarded as the best state-of-the-art feature extractors for images. However, computer vision encompasses many problems, and feature extraction is usually just the beginning. Indeed, even the basic setting of computer vision introduced in the 1960s—teaching a robot to look around itself and navigate the environment—involves much more than just image classification. When I am typing this text, I do not just recognize a "monitor" although it does take up most of my field of view: I can also see and distinguish the keyboard, my own hands, various objects on the screen all the way down to individual letters, and so on, all in the same image.



(a) (b)

Fig. 3.13 Sample training set annotations from the *OpenImages* dataset [61, 473, 489]: (a) object detection; (b) segmentation.

The real high-level problems in this basic computer vision setting are

- *object detection*, i.e., finding the location of an object in the image, usually formalized as a *bounding box* (rectangle defined by four numbers, usually the coordinates of two opposing angles), and classifying the object inside each bounding box;
- *instance segmentation*, i.e., finding the actual silhouette of every object in the image; this can be formalized as a separate classification problem for every pixel in the image: which object (or background) does this specific pixel belong to?

Figure 3.13 shows sample training set annotations from the *OpenImages* dataset, which is currently one of the largest available datasets of real data with object detection and segmentation labeling [61, 473, 489].

Note that in these new problems, the *output* of the network suddenly takes up a much higher dimension than ever before. In an ImageNet classification problem with 1000 classes, the output is a vector of probabilities assigned to these classes, so it has dimension 1000. In an object detection problem, the output has the same dimension 1000 plus four numbers defining the bounding box *for every object*, and as we will see shortly, it is usually even higher than that. In a classification problem, the output has, formally speaking, dimension 1000 *per pixel*, although in practice segmentation is rarely formalized in this straightforward way.

As much as I would like to, I have neither the space nor the willpower to make this chapter into a wide survey of the entire research field of computer vision. So in this section, I will attempt a more in-depth survey of one specific computer vision problem, namely *object detection*. This will showcase many important ideas in modern computer vision and will align with the case study in Section 6.4, where we will see how object detection architectures that we consider here benefit from the use of synthetic data.

Both object detection and segmentation have been around forever, at least since the 1960s. I will not dwell on classical approaches to these problems since, first, our focus is on deep learning, and second, most classical approaches have indeed been obsoleted by modern neural networks. I want to mention only one classical approach, the *selective search* algorithm developed in 2004 [234]. In brief, it represents the image as a graph where edge weights show similarities between small patches, starting from single pixels and gradually uniting them until the image is broken into a lot (usually several hundred) small patches. This is known as *pre-segmentation* or *sub-segmentation*, and the resulting patches are often called *superpixels*, i.e., the assumption is that the patches are so uniform that they definitely should belong to the same object. This may be a useful approach even today, and it is still used in some cases as preprocessing even for deep learning approaches [184], because after pre-segmentation the dimension of the problem is greatly reduced, from millions of pixels to hundreds or at most a few thousand of superpixels.

In 2012, selective search became the basis for a classical object detection algorithm [884] that worked as follows:

- use selective search to do pre-segmentation;
- greedily unite nearest neighbors in the resulting graph of patches; there can be lots of different proximity measures to try, based on color, texture, fill, size, and other properties;
- as a result, construct a large set of bounding boxes out of the superpixels; this is the set of candidates for object detection, but at this stage, it inevitably contains a lot of false positives;
- choose positive and negative examples, taking care to include hard negative examples that overlap with correct bounding boxes;
- train a classifier (SVM in this case) to distinguish between positive and negative examples; during inference, each candidate bounding box is run through the SVM to filter out false positives as best we can.

This pipeline will bring us to our first series of neural networks. But before we do that, we need to learn one more trick.

Convolutionalization and OverFeat. In the previous section, we have seen many wonderful properties of convolutional neural networks. But there is one more important advantage that we didn't mention there. Note how when we were counting the weights in a CNN, we never used the width and height of the input or output image, only the number of channels and the size of the convolution itself. That is because convolutions *don't care* about the size of their input: they are applied to all windows of a given size with shared weights, and it does not matter how many such windows the image contains. A network is called *fully convolutional* if it does not contain any densely connected layers with fixed topology and therefore can be applied to an input of arbitrary size.

But we can also turn regular convolutional neural networks, say *AlexNet* for image classification, into fully convolutional networks! In a process known as *convolutionalization*, we simply treat fully connected layers as 1×1 convolutions. The default *AlexNet* takes 224×224 images as input, so we can cover the input image

by 224×224 windows and run every window through *AlexNet*; the fully connected layers at the end become 1×1 convolutions with the corresponding number of channels and have the same kind of computational efficiency. As a result of this process, we will transform the original image into a heatmap of various classes: every 224×224 window will become a vector of probabilities for the corresponding classes.

This procedure is very helpful; in particular, one of the first successful applications of modern deep neural networks to object detection, *OverFeat*, did exactly this, replacing the final classifier with a regression model that predicts bounding boxes and postprocessing the results of this network with a greedy algorithm that unites the proposed bounding boxes (naturally, such a procedure will lead to a lot of greatly overlapping candidates) [782]. This approach won the ILSVRC 2013 challenge in both object detection and object localization (a variant of object detection where it is known *a priori* that there is only one significant object in the picture, and the problem is to locate its bounding box).

Most modern architectures do not take this idea to its logical conclusion, i.e., do not produce vectors of class probabilities for input windows. But basically, all of them use convolutionalization to extract features, i.e., run the input image through the first layers of a CNN, which is often one of the CNNs that we discussed in the previous section. This CNN is called the *backbone* of an object detection or segmentation pipeline, and by using a fully convolutional counterpart of a backbone the pipelines can extract features from input images of arbitrary size.

Two-stage object detection: the R-CNN family. Let us now recall the object detection pipeline based on selective search from [884] and see how we can bring CNNs into the mix.

The first idea is to use a CNN to extract the features for object classification inside bounding boxes and perhaps also the final SVM that weeds out false positives. This was exactly the idea of R-CNN [276], a method that defined new state of the art for object detection around 2013–2014. The pipeline, illustrated in Figure 3.14a, runs as follows:

- run a selective search to produce candidate bounding boxes as above;
- run each region through a backbone CNN such as *AlexNet* (pretrained for image classification and possibly fine-tuned on the training set); on this stage, the original R-CNN actually warped each region to make its dimensions match the input of the CNN;
- train an SVM on the features produced by the CNN for classification to remove the false positives;
- train a separate bounding box regression on the same features used to refine the bounding boxes, i.e., shift their corners slightly to improve the localization of objects.

This approach was working very well but was very fragile in training (it had quite a few models that all had to be trained separately but needed to work well in combination) and hopelessly slow: it took about 45–50 seconds to run the *inference* of the R-CNN pipeline on a single image, even on a GPU! This was definitely

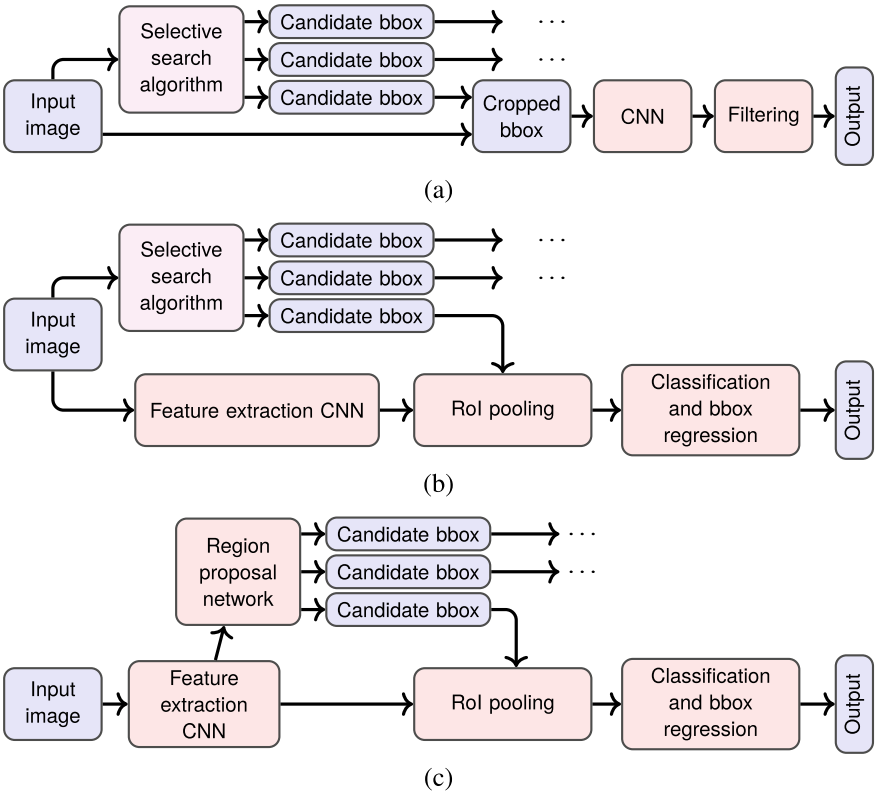


Fig. 3.14 The R-CNN family of architectures: (a) original R-CNN pipeline [276]; (b) Fast R-CNN [277]; (c) Faster R-CNN [718].

impractical, and further developments in this family of approaches tried to make R-CNN work faster.

The main reason for this excessive running time was that R-CNN needs to make a pass through the CNN for every region. Therefore, *Fast R-CNN* [277], illustrated in Fig. 3.14b, was designed so that it could use a single pass of the main backbone CNN for the whole image. The main idea of Fast R-CNN is to introduce a *region of interest (RoI) projection* layer that collects features from a region. The RoI projection layer does not have any weights; it simply maps a given bounding box to a given layer of features, translating the geometry of the original image to the (reduced) geometry in this layer of features. As a result, the tensors of features corresponding to different bounding boxes will have different dimensions.

To be able to put them through the same classifier, *Fast R-CNN* introduced the *RoI pooling* layer that performs max-pooling with dimensions needed to reduce all bounding boxes to the same size. As a result, for every bounding box we get a tensor of features with the same dimensions that can now be put through a network that

performs object classification and bounding box regression (which means that it has four outputs for bounding boxes and C outputs for the classes). Only this last part of the network needs to be run for every bounding box, and the (much larger) part of the network that does feature extraction can be run once per image.

Fast R-CNN was two orders of magnitude faster than regular R-CNN at no loss of quality. But it was still relatively slow, and now the bottleneck was not in the neural network. The slowest part of the system was now the selective search algorithm that produced candidate bounding boxes.

The aptly named *Faster R-CNN* [718] removed this last bottleneck by producing candidate bounding boxes as part of the same neural network. In the *Faster R-CNN* architecture (now it is indeed a neural architecture rather than a pipeline of different models and algorithms), shown in Fig. 3.14c, the input image first goes through feature extraction and then the tensor of features is fed to a separate *region proposal network* (RPN). The RPN moves a sliding window of possible bounding boxes along the tensor of features, producing a score of how likely it is to have an object in this box and, at the same time, exact coordinates of this box. Top results from this network are used in the RoI projection layer, and then it works exactly as discussed above. Note that all of this processing now becomes part of the same computational graph, and the gradients flow through all new layers seamlessly: they are all at the end just differentiable functions.

To me, this is a perfect illustration of the expressive power of neural networks: if you need to do some additional processing along the way, you can usually just do it as part of the neural network and train the whole thing together, in an end-to-end fashion. Soon after *Faster R-CNN* appeared, it was further improved and sped up with R-FCN (region-based fully convolutional network), which introduced position-sensitive feature maps that encode information regarding a specific position in the bounding box ("left side of the object", "bottom right corner", etc.) [177]; we will not go into the details here. *Faster R-CNN* and R-FCN remain relevant object detection frameworks up to this day (they are considered to be slow but good), only the preferred backbones change from time to time.

One-stage object detection: YOLO and SSD. The R-CNN family of networks for object detection is known as *two-stage* object detection because even *Faster R-CNN* has two clearly distinguishable separate stages: one part of the network produces candidate bounding boxes, and the other part analyzes them, ranks their likelihood to be a true positive, and classifies the objects inside.

But one can also do object detection in a single pass, looking for both bounding boxes and the objects themselves at the same time. One of the first successful applications of this approach was the original YOLO ("you only look once") object detector by Redmon et al. [709]. This was, again, a single neural network, and it implemented the following pipeline:

- split the image into an $S \times S$ grid, where S is a fixed small constant (e.g., $S = 7$);
- in each cell, predict both bounding boxes and probabilities of classes inside them; this means that the network's output is a tensor of size

$$S \times S \times (5B + C),$$

where C is the number of classes (we do classification inside every cell separately, producing the probabilities $p(\text{class}_i \mid \text{obj})$ of various classes assuming that this cell does contain an object) and $5B$ means that each of B bounding boxes is defined by five numbers: four coordinates and the score of how certain the network is in that this box is correct;

- then the bounding boxes can be ranked simply by the overall probability

$$p(\text{class}_i \mid \text{obj})\, p(\text{obj})\text{IoU},$$

where $p(\text{obj})\text{IoU}$ is the target for the certainty score mentioned above: we want it to be zero if there is no object here and if there is, to reflect the similarity between the current bounding box and the ground truth bounding box, expressed as the intersection-over-union score (also known as the Jaccard similarity index).

All this could be trained end-to-end, with a single loss function that combined penalties for incorrectly predicted bounding boxes, incorrect placement of them, and wrong classes. Overall, YOLO minimizes

$$\begin{aligned}
\mathcal{L}(\theta) =& \lambda_{\text{coord}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} [\![\text{Obj}_{ij}]\!] \left(\left(x_i - \hat{x}_i(\theta)\right)^2 + \left(y_i - \hat{y}_i(\theta)\right)^2 \right) \\
&+ \lambda_{\text{coord}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} [\![\text{Obj}_{ij}]\!] \left(\left(\sqrt{w_i} - \sqrt{\hat{w}_i(\theta)}\right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i(\theta)}\right)^2 \right) \\
&+ \sum_{i=1}^{S^2} \sum_{j=1}^{B} [\![\text{Obj}_{ij}]\!] \left(C_i - \hat{C}_i(\theta)\right)^2 + \lambda_{\text{noobj}} \sum_{i=1}^{S^2} \sum_{j=1}^{B} [\![\neg\text{Obj}_{ij}]\!] \left(C_i - \hat{C}_i(\theta)\right)^2 \\
&+ \sum_{i=1}^{S^2} [\![\text{Obj}_{ij}]\!] \sum_{c} \left(p_i(c) - \hat{p}_i(c;\theta)\right)^2,
\end{aligned}$$

where θ denotes the weights of the network, and network outputs are indicated as functions of θ.

Let us go through the original YOLO loss function in more detail as it provides an illustrative example of such loss functions in other object detectors as well:

- $[\![\text{Obj}_{ij}]\!]$ is the indicator of the event that the jth bounding box (out of B) in cell i (out of S^2) is "responsible" for an actual object appearing in this cell, that is, $[\![\text{Obj}_{ij}]\!] = 1$ if that is true and 0 otherwise;
- similarly, $[\![\text{Obj}_i]\!] = 1$ if and only if an object appears in cell i;
- the first two terms deal with the bounding box position and dimensions: if bounding box j in cell i is responsible for a real object, the bounding box should be correct, so we are bringing the coordinates of the lower left corner, width, and height of the predicted bounding box closer to the real one;

- the third and fourth terms are related to $\hat{C}_i(\theta)$, the network output that signifies the confidence that there is an object in this cell; it is, obviously, brought closer to the actual data C_i; note that since cells with and without objects are imbalanced (although this imbalance cannot hold a candle to the imbalance that we will see in SSDs), there is an additional weight λ_{coord} to account for this fact;
- the fifth term deals with classification: the internal summation runs over classes, and it brings the vector of probabilities $\hat{p}_i(c)$ that the network outputs for cell i closer to the actual class of the object, provided that there is an object in this cell;
- λ_{coord} and λ_{noobj} are hyperparameters, constants set in advance; the original YOLO used $\lambda_{\text{coord}} = 5$ and $\lambda_{\text{noobj}} = \frac{1}{2}$.

The original YOLO had a relatively simple feature extractor, and it could achieve results close to the best *Faster R-CNN* results in real time, with 40–50 frames per second while *Faster R-CNN* could do less than 10.

Further development of the idea to predict everything at once led to *single-shot detectors* (SSD) [540]. SSD uses a set of predefined *anchor boxes* that are used as default possibilities for each position in the feature map. It has a single network that predicts both class labels and the corresponding refined positions for the box angles for every possible anchor box. Applied to a single tensor of features, this scheme would obviously detect only objects of a given scale since anchor boxes would take up a given number of "pixels". Therefore, the original SSD architecture already applied this idea on several different scales, i.e., several different layers of features. The network is, again, trained in an end-to-end fashion with a single loss function.

Note that SSD has *a lot* of outputs: it has $M \times N \times (C + 4)$ outputs for an $M \times N$ feature map, which for the basic SSD architecture with a 300×300 input image came to 8732 outputs per class, that is, potentially millions of outputs per image. But this does not hinder performance significantly because all these outputs are computed in parallel, in a single sweep through the neural network. SSD worked better than the original YOLO, on par with or even exceeding the performance of *Faster R-CNN*, and again did it at a fraction of the computational costs.

Since the original YOLO, the YOLO family of one-stage object detectors has come a long way. I will not explain each in detail but will mention the main ideas incorporated by Joseph Redmon and his team into each successive iteration:

- YOLOv2 [710] tried to fix many localization errors and low recall of the original YOLO; they changed the architecture (added batch normalization layers and skip connections from earlier layers to increase the geometric resolution, predicted bounding box offsets rather than coordinates, etc.), pretrained their own high-resolution (448×448) classifier instead of using one pretrained on ImageNet (256×256), and added *multi-scale training*, i.e., trained on different image sizes;
- YOLO9000, presented in the same paper [710], generalized object detection to a large number of classes (9000, to be precise) by using the *hierarchical softmax* idea: instead of having a single softmax layer for 9000 classes, a hundred of which are various breeds of dog, let us first classify if the object is a living thing, then if it is an animal, get to a specific breed only after going down several layers in the decision tree;

- YOLOv3 [711] changed the feature extraction architecture and introduced a number of small improvements, in particular a multi-scale architecture similar to the feature pyramid networks that we will discuss below.

Another important addition to the family of one-stage object detectors was *RetinaNet* by Lin et al. [524]. The main novelty here is a modified loss function for object detection known as *focal loss*. One problem with one-stage object detection is that the output is wildly imbalanced: we have noted several times that one-stage detectors have a huge number of outputs that represent a wide variety of candidate bounding boxes, but how many of them can actually be correct?

The mathematical side of this problem is that even correctly classified examples (a bounding box in the background correctly classified as "no object") still contribute to the classification loss function: the usual cross-entropy loss equals $-\log p$ for an example where the correct class gets probability p. This is a small value when p is close to 1 (which it should be when the network is sure), but negative examples outweigh positive examples by a thousand to one, and these relatively small values add up. So focal loss downweighs the loss on well-classified examples, bringing it close to zero with an additional polynomial factor in the loss function: the focal loss for a correctly classified example is $-(1-p)^\gamma \log p$ for some $\gamma > 0$ instead of the usual cross-entropy loss $-\log p$. Focal loss has proved to be an extremely useful idea, used in a wide variety of deep learning models since the original work [524].

The YOLO family of networks and *RetinaNet* have defined state of the art in real-time object detection for a long time. In particular, YOLOv3 was the model of choice for at least 2 years, and this situation has begun to change only very recently. We will go back to the story of YOLO at the end of this section, but for now let us see the other main ideas in modern object detection.

Object detection at different scales: feature pyramid networks. To motivate the next set of ideas, let us analyze which specific problems have historically plagued object detection. If you look at the actual results in terms of how many objects are usually detected, you will see that the absolute numbers in object detection are pretty low: long after image classifiers beat *ImageNet* down to superhuman results of less than 5% top-5 test error (human level was estimated at about 5.1%) [329], the best object detectors were getting mean average precision of about 80% on the (relatively simple) PASCAL VOC dataset and struggled to exceed 35% mean average precision on the more realistic Microsoft COCO dataset. At the time of writing (summer of 2020), the best mAP on Microsoft COCO is about 55%, still very far from perfect [848, 900]. Why are the results so low?

One big problem lies in the different *scales* of objects that need to be recognized. Small objects are recognized very badly by most models discussed above, and there are plenty of them in real life and in realistic datasets such as *Microsoft COCO*. This is due to the so-called *effective stride*: by the time the region proposal network kicks in, the original geometry has already been compressed a lot by the initial convolutional layers. For example, the basic *Faster R-CNN* architecture has effective stride 16, i.e., a 16×16 object is only seen as a single pixel by the RPN. We could try to reduce effective stride mechanically, by doing convolutional layers without pooling

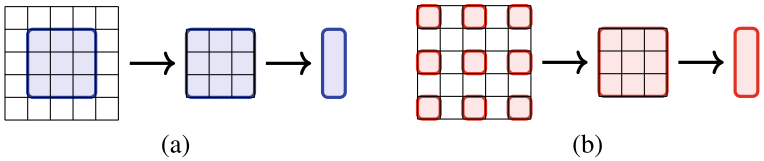


Fig. 3.15 Types of convolutions: (a) regular convolutions; (b) dilated convolutions.

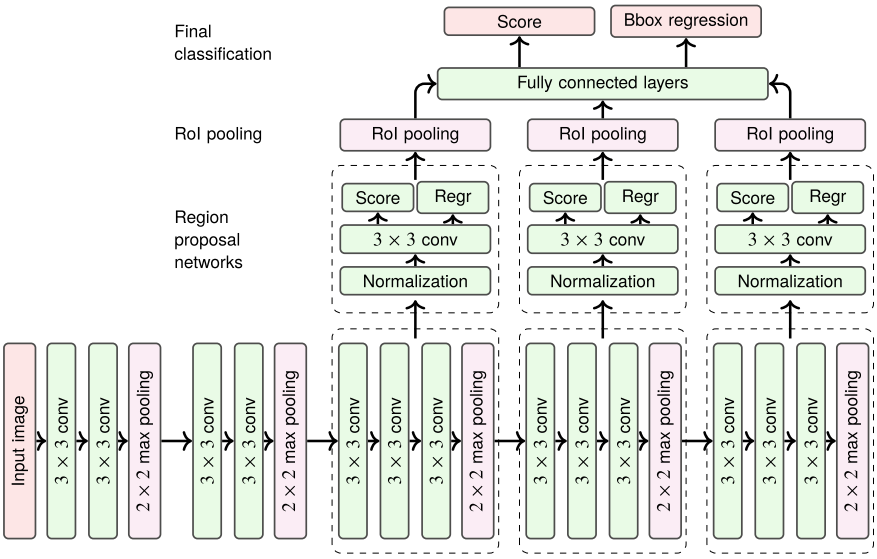


Fig. 3.16 Multi-scale object recognition: a sample architecture from [222] that does not have top-down connections.

and without reducing the geometry, but then the networks become huge and basically infeasible. On the other hand, there are also large objects: a car may be taking up either 80% of the photo or just a tiny 30 × 30 pixel spot somewhere; sometimes both situations happen on the same photo. What do we do?

One idea is to change how we do convolutions. Many object detection architectures use *dilated* (sometimes also called *atrous*) convolutions, i.e., convolutions whose input window is not a contiguous rectangle of pixels but a strided set of pixels from the previous layer, as shown in Fig. 3.15; see, e.g., [211] for a detailed explanation. With dilated convolutions, fewer layers are needed to achieve large receptive fields, thus saving on the network size. Dilated convolutions are used in Faster R-CNN, R-FCN, and other networks; see, e.g., a work by Yu and Koltun where dilated convolutions were successfully applied to semantic segmentation [980].

But this is just a trick that can improve the situation, not a solution. We still need to cope with multi-scale objects in the same image. The first natural idea in this direction is to gather proposals from several different layers in the backbone feature extraction

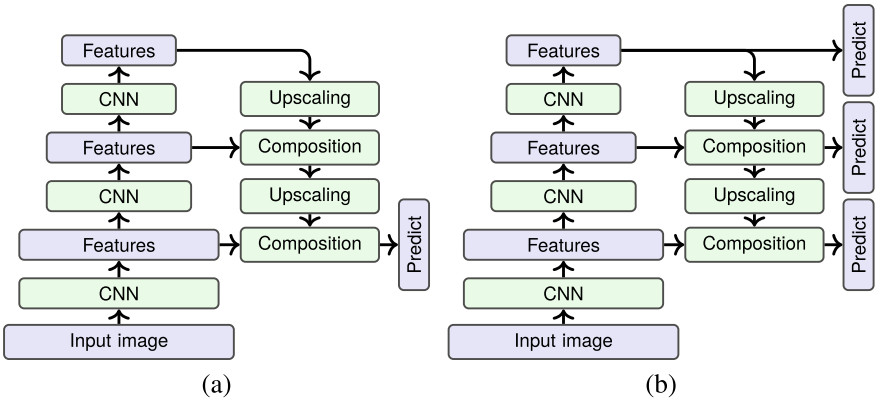


Fig. 3.17 Feature pyramid networks: (a) top-down architecture with predictions on the bottom level [108, 674]; (b) feature pyramid network [523].

network. In a pure form, this idea was presented by Eggert et al. [222], where the architecture has several (three, to be precise) different region proposal networks, each designed to recognize a given scale of objects, from small to large. Figure 3.16 shows an outline of their architecture; the exact details follow *Faster R-CNN* quite closely. The results showed that this architecture improved over basic *Faster R-CNN*, and almost the entire improvement came from far better recognition of small objects. A similar idea in a different form was implemented by Cai et al. [108], who have a single region proposal network with several branches with different scales of outputs (the branches look kind of like *GoogLeNet*).

But the breakthrough came when researchers realized that top layers can inform lower layers in order to make better proposals and better classify the objects in them. An early layer of a convolutional backbone can only look at a small part of the input image, and it may lack the semantic expressiveness necessary to produce good object proposals. Therefore, it would be beneficial to add *top-down* connections, i.e., use the semantically rich features produced on top layers of the CNN to help the early layers understand what objects they are looking at.

This is exactly the idea of *feature pyramid networks* (FPN), presented in 2016 by Lin et al. [523]. Actually, there are two ways to do that. One way is to use skip connections to get from the top down; the work [108] already contains an architecture that implements this idea, and it appeared in other previous works as well [674]; we illustrate it in Fig. 3.17a. The difference in the feature pyramid approach is that instead of sending down semantic information to the bottom layer, Lin et al. make RoI predictions independently on several layers along this top-down path, as shown in Fig. 3.17b; this lets the network better handle objects of different scales. Note that the upscaling and composition parts do not have to have complicated structure: in [523], upscaling is a single convolutional layer, and composition is done by using a single 1×1 convolution on the features and adding it to the result of upscaling.

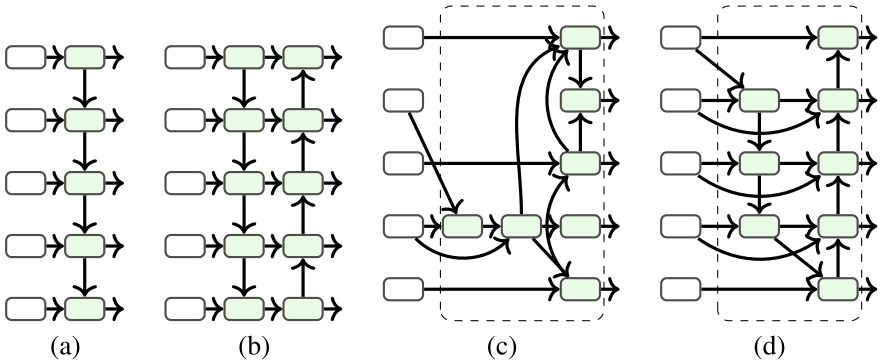


Fig. 3.18 The top-down pathway in various pyramid architectures: (a) FPN [523]; (b) PANet [538]; (c) NAS-FPN [274]; (d) BiFPN from *EfficientDet* [848]. Dashed rectangles show repeated blocks.

The "predict" part can be complicated if needed, e.g., it can represent the RPN from *Faster R-CNN*.

Feature pyramids have become a staple of two-stage object detection. One important advance was Path Aggregation Network (PANet) [538], which added new pathways between layers of the pyramid and introduced adaptive feature pooling; specifically, they added another bottom-up pathway to the top-down one, as shown in Fig. 3.18b. Another advance was NAS-FPN [274], which used neural architecture search to find a better feature pyramid architecture. Its resulting architecture is shown in Fig. 3.18c; note that only one block is shown but it is repeated several times in the final architecture.

At present, research into feature pyramid architectures has culminated with *EfficientDet*, a network developed by *Google Brain* researchers Tan et al. [848]. Variations of this network currently show record performance, sometimes losing to solutions based on CSPNet, a new recently developed CNN backbone (we will not go into details on this one) [900]. *EfficientDet* introduced a new solution for the multi-scale feature fusion problem, an architecture called BiFPN (weighted bidirectional feature pyramid network); the outline of one of its repeated blocks is shown in Fig. 3.18d.

YOLOv4, YOLOv5, and current state of the art. I would like to finish this section with an interesting story that is unfolding right as I'm writing this book.

After YOLOv3, Joseph Redmon, the main author of all YOLO architectures that we have discussed above, announced that he stopped doing computer vision research[2]. For about 2 years, YOLOv3 basically defined the state of the art in object detection: detectors that could outperform it worked much slower. But in 2020, Alexey Bochkovskiy et al. released YOLOv4, a significantly improved (and again sped up) version of YOLO [78]. By using new methods of data augmentation, including even GAN-based style transfer (we will discuss such architectures in Chapter 10),

[2]He said that the main reason was that "the military applications and privacy concerns eventually became impossible to ignore". Indeed, modern computer vision can bring up certain ethical concerns, although it is far from obvious on which side the scales tip.

using the mixup procedure in training [993], and actually a large and diverse collection of other recently developed tricks (see Section 3.4), Bochkovskiy et al. managed to reach performance comparable to *EfficientDet* with performance comparable to YOLOv3.

But that's not the end of the story. In just two months, a different group of authors released an object detector that they called YOLOv5. At the time of writing (summer of 2020), there is still no research paper or preprint published about YOLOv5, but there is a code repository[3], and the blog posts by the authors claim that YOLOv5 achieves the same performance as YOLOv4 at a fraction of model size and with better latencies [622, 623]. This led to the usual controversy about naming, credit, and all that, but what is probably more important is that it is still not confirmed which of the detectors is better; comparisons are controversial, and third-party comparisons are also not really conclusive. Matters are not helped by the subsequent release of PP-YOLO [547], an improved reimplementation of YOLOv3 that also added a lot of new tricks and managed to outperform YOLOv4...

By the time you are reading this, the controversy has probably already been settled, and maybe you are already using YOLOv6 or later, but I think this snapshot of the current state of affairs is a great illustration of just how vigorous and alive modern deep learning is. Even in such a classical standard problem as object detection, a lot can happen very quickly!

3.4 Data Augmentations: The First Step to Synthetic Data

In the previous section, we have already mentioned *data augmentation* several times. Data augmentations are defined as transformations of the input data that change the target labels in predictable ways. This allows to significantly (often by several orders of magnitude) increase the amount of available data at zero additional labeling cost. In fact, I prefer to view data augmentation as the first step towards synthetic data: there is no synthetic data generation *per se*, but there is recombination and adaptation of existing real data, and the resulting images often look quite "synthetic".

The story of data augmentation for neural networks begins even before the deep learning revolution; for instance, Simard et al. [801] used distortions to augment the MNIST training set in 2003, and I am far from certain that this is the earliest reference. The MC-DNN network discussed in Section 3.2, arguably the first truly successful deep neural network in computer vision, also used similar augmentations even though it was a relatively small network trained to recognize relatively small images (traffic signs).

But let us begin in 2012, with *AlexNet* [477] that we have discussed in detail in Section 3.2. *AlexNet* was the network that made the deep learning revolution happen in computer vision... and even with a large dataset such as *ImageNet*, even back in

[3] https://github.com/ultralytics/yolov5.

2012 it would already be impossible without data augmentation! *AlexNet* used two kinds of augmentations:

- horizontal reflections (a vertical reflection would often fail to produce a plausible photo) and
- image translations; that is the reason why the *AlexNet* architecture, shown in Fig. 3.7, uses $224 \times 224 \times 3$ inputs while the actual *ImageNet* data points have 256 pixels by the side: the 224×224 image is a random crop from the larger 256×256 image.

With both transformations, we can safely assume that the classification label will not change. Even if we were talking about, say, object detection, it would be trivial to shift, crop, and/or reflect the bounding boxes together with the inputs—that is exactly what we mean by "changing in predictable ways". The resulting images are, of course, highly interdependent, but they still cover a wider variety of inputs than just the original dataset, reducing overfitting. In training *AlexNet*, Krizhevsky et al. estimated that they could produce 2048 different images from a single input training image. What is interesting here is that even though *ImageNet* is so large (*AlexNet* trained on a subset with 1.2 million training images labeled with 1000 classes), modern neural networks are even larger (*AlexNet* has 60 million parameters). Krizhevsky et al. had the following to say about their augmentations: "Without this scheme, our network suffers from substantial overfitting, which would have forced us to use much smaller networks" [477].

By now, this has become a staple in computer vision: while approaches may differ, it is hard to find a setting where data augmentation would not make sense at all.

To review what kind of augmentations are commonplace in computer vision, I will use the example of the *Albumentations* library developed by Buslaev et al. [103]; although the paper was only released in 2020, the library itself had been around for several years and by now has become the industry standard.

The first candidates are color transformations. Changing the color saturation, permuting color channels, or converting to grayscale definitely does not change bounding boxes or segmentation masks, as we see in Figure 3.19. This figure also shows two different kinds of blurring, jpeg compression, and various brightness and contrast transformations.

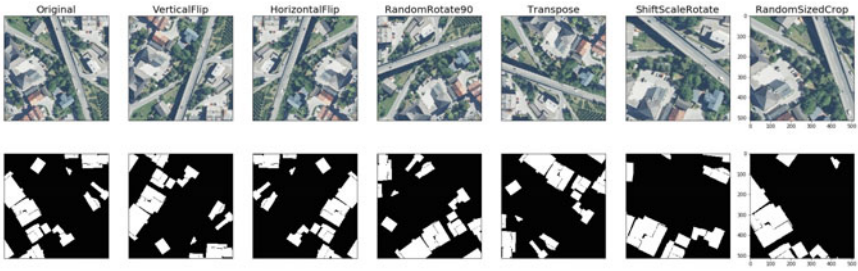
The next obvious category are simple geometric transformations. Again, there is no question about what to do with segmentation masks when the image is rotated or cropped: we can simply repeat the same transformation with the labeling. Figure 3.20 shows examples of both global geometric transformations such as flipping or rotation (Fig. 3.20a) and local distortions defined according to a grid (Fig. 3.20b). The same ideas can apply to other types of labeling; for instance, keypoints (facial landmarks, skeletal joints in pose estimation, etc.) can be treated as a special case of segmentation and also changed together with the input image.

All of these transformations can be chained and applied multiple times. Figure 3.21 shows a more involved example produced with the *Albumentations* library; it corresponds to the following chain of augmentations:

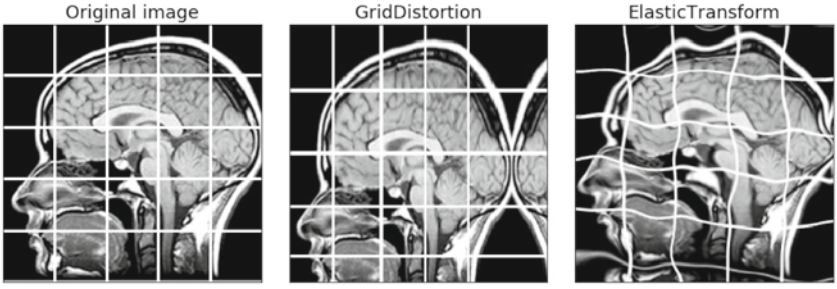
Fig. 3.19 Sample color transformations and blurring provided by the *Albumentations* library [103].

- take a random crop from a predefined range of sizes;
- shift, scale, and rotate the crop to match the original image dimension;
- apply a (randomized) color shift;
- add blur;
- add Gaussian noise;
- add a randomized elastic transformation for the image;
- perform *mask dropout*, removing a part of the segmentation masks and replacing them with black cutouts on the image.

That's quite a few operations! But how do we know that this is the best way to approach data augmentation for this particular problem? Can we find the best possible sequence of augmentations? Indeed, recent research suggests that we can look for a meta-strategy for augmentations that would take into account the specific problem setting; we will discuss these approaches in Section 12.2.



(a)



(b)

Fig. 3.20 Sample geometric transformations provided by the *Albumentations* library [103]: (a) global transformations; (b) local distortions.

But even that is not all! What if we take augmentation one step further and allow augmentations to produce more complex combinations of input data points? In 2017, this idea was put forward in the work titled "Smart Augmentation: Learning an Optimal Data Augmentation Strategy" by Lemley et al. [506]. Their basic idea is to have two networks, "Network A" that implements an augmentation strategy and "Network B" that actually trains on the resulting augmented data and solves the downstream task. The difference here is that "Network A" does not simply choose from a predefined set of strategies but operates as a generative network that can, for instance, blend two different training set examples into one in a smart way. I will not go into the full details of this approach, but Figure 3.22 provides two characteristic examples from [506].

This kind of "smart augmentation" borders on synthetic data generation: transformations are complex, and the resulting images may look nothing like the originals. But before we turn to actual synthetic data generation in subsequent chapters, let us discuss other interesting ideas one could apply even at the level of augmentation.

Mixup, a technique introduced by MIT and FAIR researchers Zhang et al. in 2018 [993], looks at the problem from the opposite side: what if we mix the labels together with the training samples? This is implemented in a very straightforward way: for two labeled input data points, Zhang et al. construct a convex combination

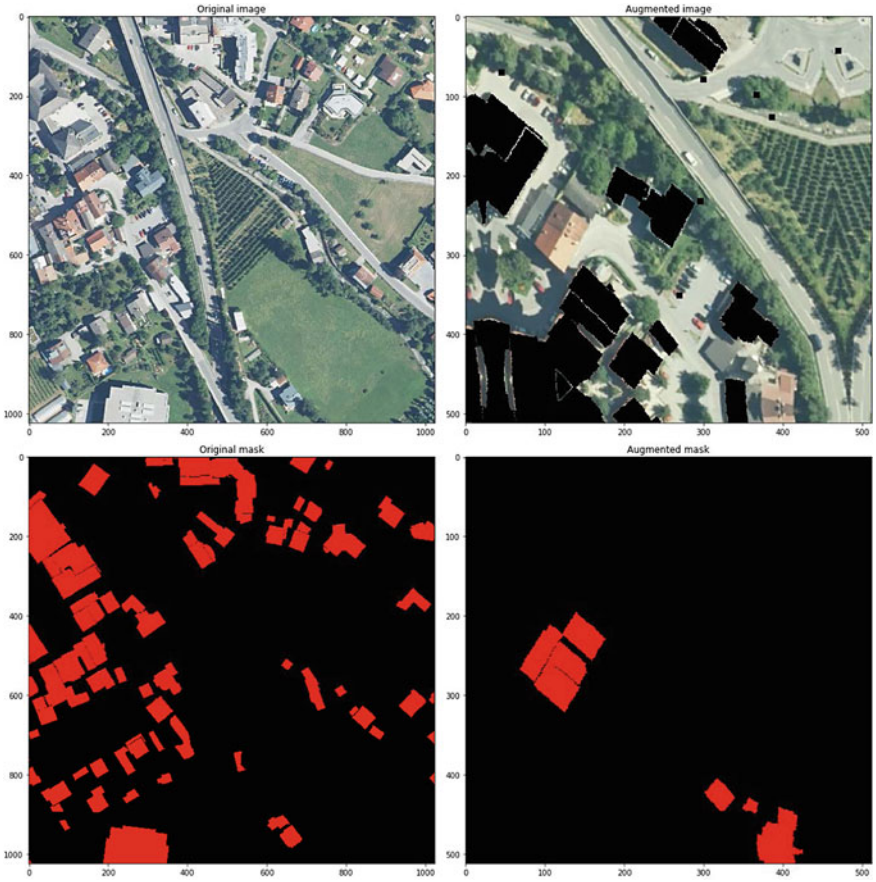


Fig. 3.21 An involved example of data augmentation by transformations from the *Albumentations* library [103].

Fig. 3.22 An example of "smart augmentations" by Lemley et al. [506]: the image on the left is produced as a blended combination of two images on the right.

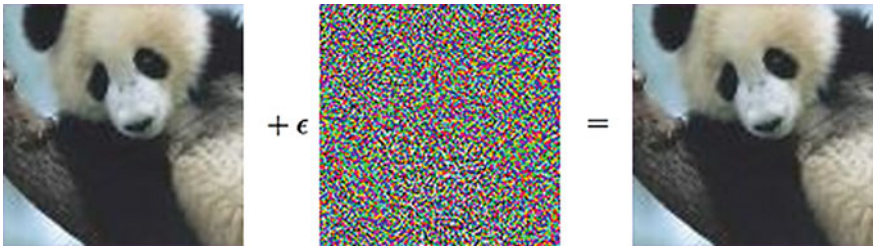


Fig. 3.23 The famous "panda-to-gibbon" adversarial example [292]: the image on the left is recognized by *AlexNet* as a panda with 57.7% confidence, but after adding small random-looking noise the network recognizes the image on the right as a gibbon with 99.3% confidence.

of both the inputs and the labels:

$$\begin{aligned}\tilde{\mathbf{x}} &= \lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2, \quad \text{where } \mathbf{x}_1, \mathbf{x}_2 \text{ are raw input vectors,}\\ \tilde{\mathbf{y}} &= \lambda\mathbf{y}_1 + (1-\lambda)\mathbf{y}_2, \quad \text{where } \mathbf{y}_1, \mathbf{y}_2 \text{ are one-hot label encodings.}\end{aligned}$$

The blended label does not change either the network architecture or the training process: binary cross-entropy trivially generalizes to target discrete distributions instead of target one-hot vectors.

The resulting labeled data covers a much more robust and continuous distribution, and this helps the generalization power. Zhang et al. report especially significant improvements in training GANs. By now, the idea of mixup has become an important part of the deep learning toolbox: you can often see it as an augmentation strategy, especially in the training of modern GAN architectures.

The last idea that I want to discuss in this section is *self-adversarial training*, an augmentation technique that incorporates *adversarial examples* [292, 484] into the training process. Adversarial examples are a very interesting case that showcases certain structural and conceptual problems with modern deep neural networks. It turns out that for most existing artificial neural architectures, one can modify input images with small amounts of noise in such a way that the result looks to us humans completely indistinguishable from the originals but the network is very confident that it is something completely different. The most famous example from one of the first papers on adversarial examples by Goodfellow et al. [292], with a panda turning into a very confident gibbon, is reproduced in Fig. 3.23.

By now, adversarial examples and ways to defend against them have become a large field of study in modern deep learning; let me refer to [781, 963] for recent surveys on adversarial examples, attacks, and defenses.

In the simplest case, such adversarial examples are produced by the following procedure:

- suppose that we have a network and an input $\mathbf{x}$ that we want to make adversarial; let us say that we want to turn a panda into a gibbon;

- formally, it means that we want to increase the "gibbon" component of the network's output vector $\mathbf{y}$ (at the expense of the "panda" component);
- so we fix the weights of the network and start regular gradient ascent, but with respect to $\mathbf{x}$ rather than the weights!

This is the key idea for finding adversarial examples; it does not explain why they exist (it is not an easy question) but if they do, it is really not so hard to find them in the "white box" scenario, when the network and its weights are known to us, and therefore we can perform this kind of gradient ascent with respect to the input image.

So how do we turn this idea into an augmentation technique? Given an input instance, let us make it into an adversarial example by following this procedure for the current network that we are training. Then we train the network on this example. This may make the network more resistant to adversarial examples, but the important outcome is that it generally makes the network more stable and robust: now we are explicitly asking the network to work robustly in a small neighborhood of every input image. Note that the basic idea can again be described as "make the input data distribution cover more ground", but by now we have come quite a long way since horizontal reflections and random crops.

Note that unlike basic geometric augmentations, this may turn out to be a quite costly procedure. But the cost is entirely borne during training: yes, you might have to train the final model for two weeks instead of one, but the resulting model will, of course, work with exactly the same performance. The model architecture does not change, only the training process does.

One of the best recent examples for the combined power of various data augmentations is given by the architecture that we discussed in Section 3.3: YOLOv4 by Bochkovskiy et al. [78]. Similar to many other advances in the field of object detection, YOLOv4 is in essence a combination of many small improvements. Faced with the challenge of improving performance but not sacrificing inference speed, Bochkovskiy et al. systematize these improvements and divide them into two subsets:

- *bag of freebies* includes tricks and improvements that do not change the inference speed of the resulting network at all, modifying only the training process;
- *bag of specials* includes changes that do have a cost during inference but the cost is, hopefully, as small as possible.

The "bag of specials" includes all changes related to the architecture: new activation functions, bounding box postprocessing, a simple attention mechanism, and so on. But the majority of the overall improvement in YOLOv4 comes from the "bag of freebies"... which almost entirely consists of augmentations.

YOLOv4 uses all kinds of standard augmentations, self-adversarial training, and mixup that we have just discussed, and introduces new Mosaic and CutMix augmentations that amount to composing an input image out of pieces cropped out of other images (together with the objects and bounding boxes, of course). This is just one example but there is no doubt that data augmentations play a central role in modern computer vision: virtually every model is trained or pretrained with some kind of data augmentations.

3.5 Conclusion

In this chapter, we have seen a brief introduction to the world of deep learning for computer vision. We have seen the basic tool for all computer vision applications—convolutions and convolutional layers,—have discussed how convolutional layers are united together in deep feature extractors for images, and have seen how these feature extractors, in turn, can become key parts of object detection and segmentation pipelines.

There is really no hope to cover the entirety of computer vision in a single short chapter. In this chapter, I have decided to concentrate on giving at least a very brief overview of the most important ideas of convolutional architectures and review as an in-depth example of one basic high-level computer vision problem, namely object detection. Problems such as object detection and segmentation are very much aligned with the topic of this book, synthetic data. In synthetic images, both object detection and segmentation labeling come for free: since we control the entire scene, all objects in it, and the camera, we know precisely which object every pixel belongs to. On the other hand, hand-labeling for both problems is extremely tedious (try to estimate how long it would take you to label a photo like the one shown in Fig. 3.13!). Therefore, such basic high-level problems are key to the whole idea of synthetic data, and in Chapter 6 we will see how modern synthetic datasets help the networks that we have met in this chapter. In fact, I will devote a whole case study in Section 6.4 to just object detection with synthetic data.

In this chapter, we have also begun to discuss synthetic data, in its simplest form of data augmentations. We have seen how augmentations have become a virtually indispensable tool in computer vision these days, from simple geometric and color transformations to very complex procedures such as self-adversarial training. But still, even "smart" augmentations are just combinations and modifications of real data that someone has had to label in advance, so in Chapter 6 the time will come for us to move on to purely synthetic examples.

Before we do that, however, we need another introductory chapter. In the next chapter, we will consider generative models in deep learning, from a general introduction to the notion of generative models to modern GAN-based architectures and a case study of style transfer, which is especially relevant for synthetic-to-real domain adaptation.