Chapter 1

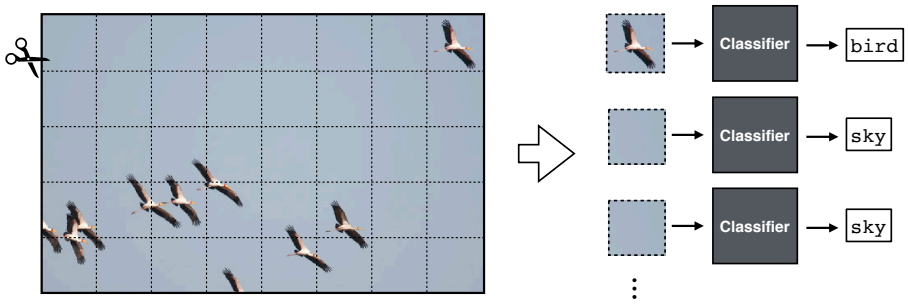
Convolutional neural nets

Draft chapter from Torralba, Isola, Freeman

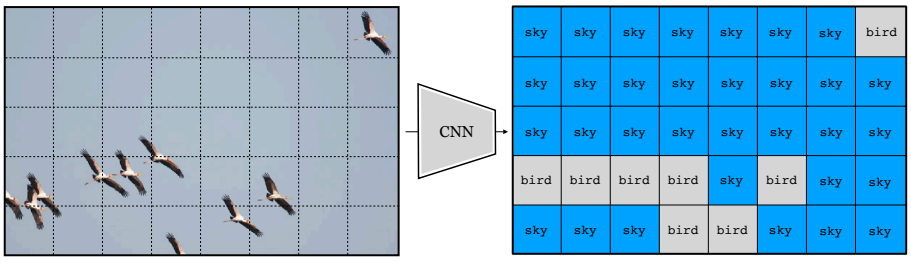
The neural nets we saw in Chapter **??** are designed to process generic data. But in many domains the data has special structure, and we can design neural net architectures that are better suited to exploiting that structure. **Convolutional neural nets**, also called **Convnets** or **CNNs**, are a neural net architecture especially suited to the structure in visual signals.

CNNs are also well suited to many other spatial or temporal signals, such as geospatial data or sounds. If there is a natural way to "scan" across a signal, processing each windowed region separately, then CNNs may be a reasonable choice.

The key idea of CNNs is to chop up the input image into little patches, and then process each patch *independently* and *identically*. The gist of this is captured in the figure below, which is an example of a CNN classifier:

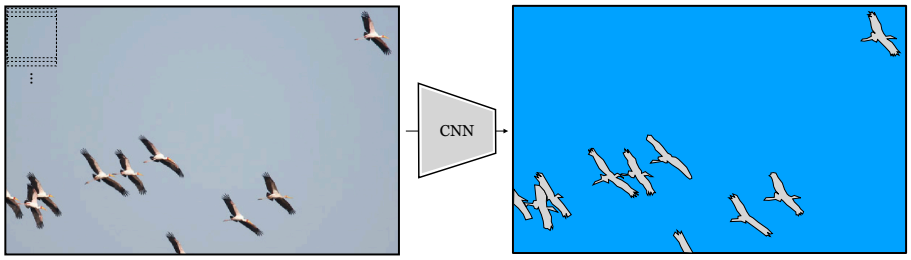


Each patch is processed independently and identically with the "Classifier" module, which is a neural net. Essentially, this neural net scans across the patches in the input and classifies each. The output is a label *for each patch in the input image*. If we rearrange these predictions back into the shape of the input image, and color code them, we get the below input-output mapping:



Notice that this is quite different than the neural nets we saw in Chapter **??**, which output a single prediction for the entire image; CNNs output a 2D *array* of predictions.

We may also chop up the image into *overlapping* patches. If we do this densely, such that each patch is one pixel offset from the last, we get a full resolution image of predictions:



Now that looks impressive! This CNN solved a task known as "semantic segmntation", which is the task of assigning a class label to each pixel in an image. One reason CNNs are powerful is because they map an input image to an output image *with the same shape*, rather than outputting a single label like in the nets we saw in previous chapters. CNNs can also be generalized to input and output other kinds of structures. The key property is that the output matches the topology of the input: an ND tensor of inputs will be mapped to an ND tensor of outputs.

Keeping in mind that chopping up and predicting is really all a CNN is doing, we will now dive into the details of how they work.

1.1 Convolution layers

Here we deviate slightly from our usual notation and use lowercase for convolutional filter $\mathbf{w}$, regardless of whether the kernel is a 1D array, a 2D array, or an ND array.

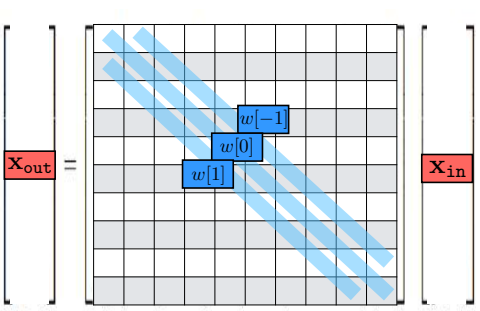
CNNs are neural networks that are composed of **convolutional layers**. A convolutional layer transforms inputs $\mathbf{x}_{\texttt{in}}$ to outputs $\mathbf{x}_{\texttt{out}}$ by convolving $\mathbf{x}_{\texttt{in}}$ with one or more filters $\mathbf{w}$. A convolutional layer with a single filter looks like this:

$$\mathbf{x}_{\texttt{out}} = \mathbf{w} \circ \mathbf{x}_{\texttt{in}} + b \qquad \triangleleft \quad \texttt{conv} \tag{1.1}$$

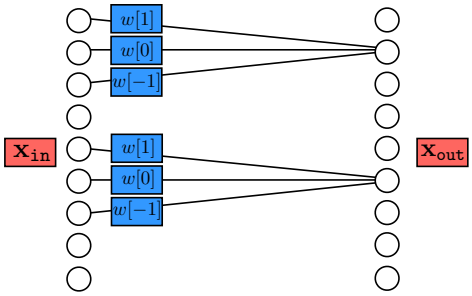
where $\mathbf{w}$ is the kernel and b is the bias; $\theta = [\mathbf{w}, b]$ are the parameters of this layer. Recalling the definition of the convolution operator $\circ$ from Chapter 12, we can write out the convolution layer in more explicit detail as:

$$\mathbf{x}_{\texttt{out}}[n, m] = b + \sum_{k,l=-N}^{N} \mathbf{w}[k, l]\mathbf{x}_{\texttt{in}}[n-k, m-l] \qquad \triangleleft \quad \texttt{conv} \quad \text{(expanded)} \tag{1.2}$$

As discussed in Chapter **??**, convolution is just a special kind of linear transform. Similarly, a convolutional layer is just a special kind of linear layer. It is a linear layer whose matrix $\mathbf{W}$ is a Toeplitz. We can view it either as a matrix or as a neural net:



$w[n-k]$ n indexes rows, k indexes columns



$w[n-k]$ Is the strength of the connection between $x[k]$ and $y[n]$

We already saw that convolutional filters are useful for image processing in the previous chapters on Image Modeling. In those chapters, we introduced a variety of hand-designed filter banks with useful properties. A CNN instead *learns* an effective filter bank.

1.1.1 Multi-channel convolutional layers

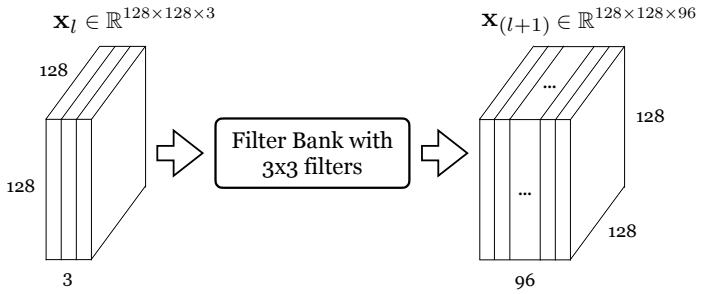
Typically, we will use convolutional layers that apply a *filter bank* to a multi-channel input signal to produce a multi-channel output signal, resulting in a convolutional layer with the following form for each of K channels:

$$\mathbf{x}_{\mathtt{out}_k} = \sum_{c=1}^{C} \mathbf{w}_{k,c} \circ \mathbf{x}_{\mathtt{in}_c} + \mathbf{b}_k \qquad \triangleleft \quad \mathtt{conv} \quad \text{(multi-channel)} \tag{1.3}$$

$\mathbf{x}_{\mathtt{out}_k}$ is the k-th **channel** of the output, also called the k-th **feature map**. $\mathbf{x}_{\mathtt{in}_c}$ is the c-th channel of the input signal. The **filter bank** is the set of multi-channel filters $[\mathbf{w}_1, \ldots, \mathbf{w}_k]$, each of which applies one convolutional filter per input channel and then sums the responses over all these filters. This conv layer maps inputs $\mathbf{x}_{\mathtt{in}} \in \mathbb{R}^{N \times M \times C}$ to outputs $\mathbf{x}_{\mathtt{out}} \in \mathbb{R}^{N \times M \times K}$.

For images, the input signal is an $N \times M \times C$ tensor of pixels, where N is the image height, M is the image width, and C is the number of color channels (usually 3), and the output signal is an $N \times M \times K$ tensor of features (neural activations).

It's important to get comfortable with the shapes of the data and parameter tensors that get processed through different neural architectures. This is essential when designing and building these architectures, and when analyzing and debugging them. Let's go through an example. Consider data $\mathbf{x}$ which is an RGB image of size 128×128 pixels. We will pass it through a conv layer that applies a bank of 3×3 filters. We omit the bias terms for simplicity. The output ends up being a $128 \times 128 \times 96$ tensor, as shown below:



To check your understanding, you should be able to answer the following two questions:

1. How many parameters does each filter have? A) 9, B) 27, C) 96, D) 864
2. How many filters are in the filter bank? A) 3, B) 27, C) 96, D) can't say

The answers are given in the footnote[1].

1.1.2 Strided convolution

Conv layers, as defined above, maintain the spatial resolution of the signal they process. However, commonly it is sufficient to output a lower resolution. This can be achieved with strided convolution:

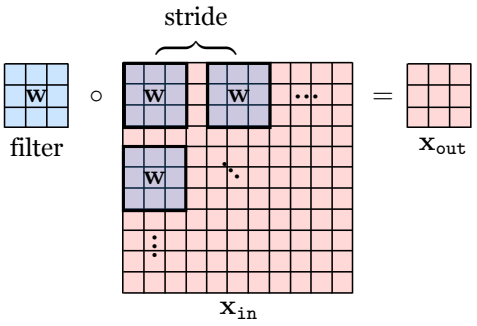
$$\mathbf{x}_{\mathtt{out}}[n, m] = b + \sum_{k,l=-N}^{N} \mathbf{w}[k, l] \mathbf{x}_{\mathtt{in}}[s_n * n - k, s_m * m - l] \qquad \triangleleft \quad \mathtt{conv} \quad \text{(strided)} \tag{1.4}$$

s_n and s_m are the strides in the vertical and horizontal directions respectively. Commonly we use the same stride $s_n = s_m = s$. A convolution layer with these strides performs a mapping $\mathbb{R}^{M \times N} \to \mathbb{R}^{N/s_n \times M/s_m}$. In order to make this mapping well defined, we require

[1] 1. B, 2. C

that N or M are divisible by s_n and s_m respectively; if they are not, we may may pad (or crop) the input until they are.

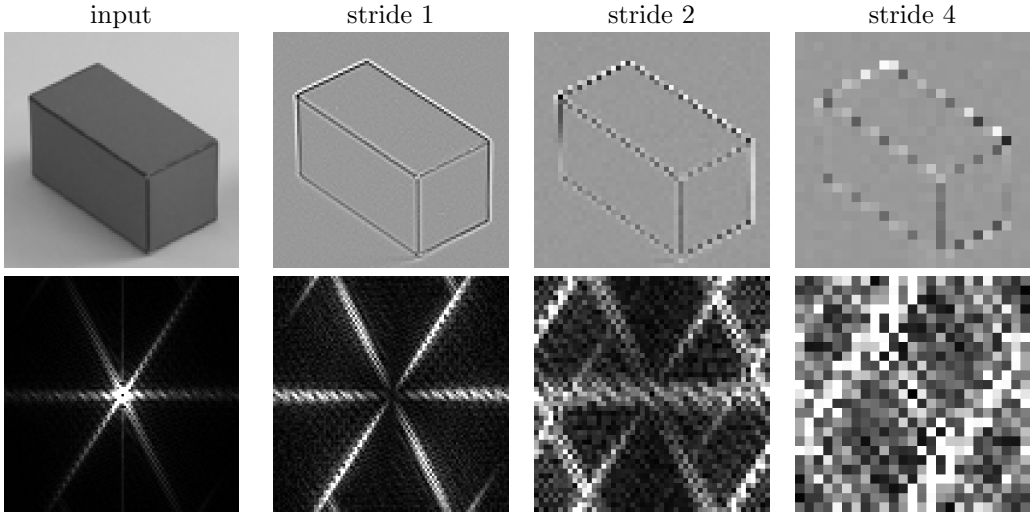
Strided convolution looks like this:



Strided convolutions can significantly reduce the computational cost and memory requirements when a neural network is large. However, strided convolution can decrease the quality of the convolution. Let's look at one concrete example where the kernel is the 2D Laplacian:

$$\mathbf{w} = \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix} \tag{1.5}$$

As we saw in Chapter XX, this filter detects boundaries on images. The next images show the input image, and the result of the strided convolution with the Laplacian kernel with strides 1, 2 and 4. The second row shows the magnitude of the discrete Fourier Transforms (DFT).



The result with stride 1 looks fine and it is the output we would expect. However stride 2 starts showing some artifacts in the boundaries and stride 4 shows very severe artifacts, with some boundaries disapearing. The DFTs make the artifacts more obvious. In the stride 2 result we can see severe aliasing artifacts that introduce new lines in the Fourier domain that are not present in the DFT of the input image.

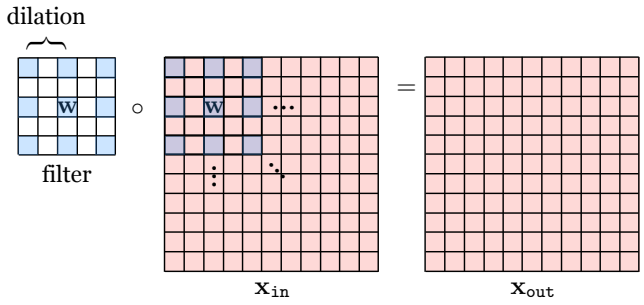
One can argue that these artifacts might not be important when the kernel is being learned. Indeed, the learning could search for kernels that minimize the artifacts due to aliasing as those probably increase the loss. Also, as each layer is composed of many channels, the set of learned kernels could learn to compensate for the aliasing produced by other channels. However, this reduces the space of useful kernels. However, the learning might not succeed in removing all the artifacts.

1.1.3 Dilated convolution

Dilated convolution is similar to strided convolution but spaces out the *filter* itself rather than spacing out where the filter is applied to the image:

$$\mathbf{x}_{\texttt{out}}[n,m] = b + \sum_{k,l=-N}^{N} \mathbf{w}[k,l]\mathbf{x}_{\texttt{in}}[n - d_k * k, m - d_l * l] \qquad \triangleleft \quad \texttt{conv} \quad \text{(dilated)} \qquad (1.6)$$

Visually it looks like this:



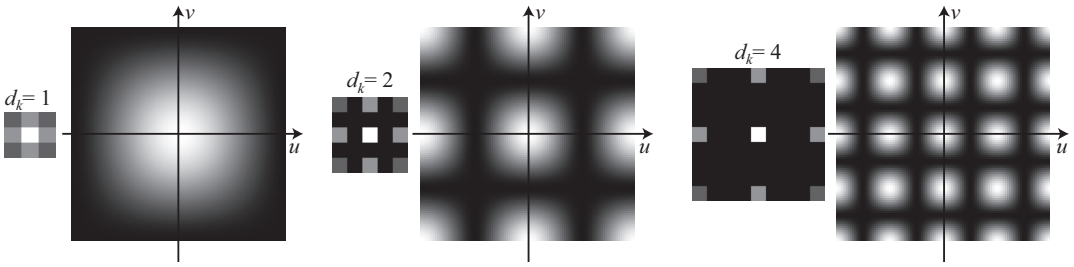
As can be seen in the visualization, dilation is a way to achieve a filter with large receptive field while only requiring a small number of weights. The weights are just spaced out so that a few will cover a bigger region of the image.

As was the case with the strided convolution, dilation can also introduce artifacts. Let's look at one example in detail that illustrates the effect of dilation on a filter. Let's consider he blur kernel, $b_{2,2}$, that we introduced in Chapter XX:

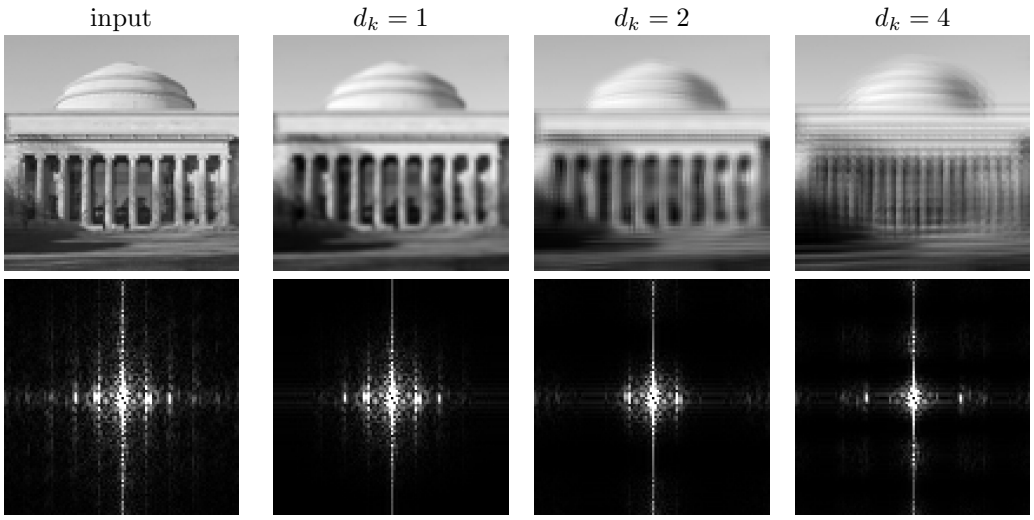
$$\mathbf{w} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \qquad (1.7)$$

As we saw in Chapter XX, this filter blurs the input image by computing the weighted average of pixel intensities around each pixel location. But, dilation transforms this filter in ways that change the behavior of the filter, which does not behave as a blur filter any longer. The next figure shows the kernel with dilations $d_k = 1$, $d_k = 2$ and $d_k = 4$ together with the magnitude of the DFT of the three resulting kernels.

We saw that the 1D signal $[-1, 1, -1, ...]$ convolved with $[1, 2, 1]$ outputs zero. However, check what happens when we convolve the input with the dilated kernel $[1, 0, 2, 0, 1]$.

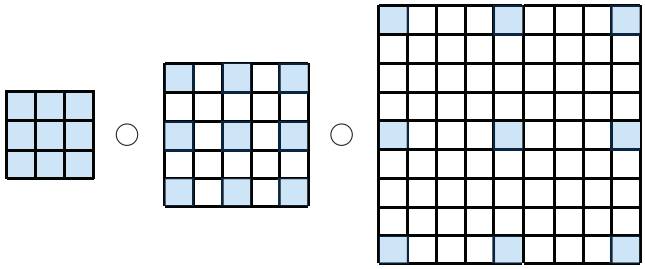


When using the original binomial filter (which corresponds to $d_k = 1$) the DFT shows that the filter is a low-pass filter. When applying dilation ($d_k = 2$) the DFT changes and it is not unimodal anymore. It has now 8 additional local maximum in high spatial frequencies. With $d_k = 4$, the DFT reveals an even more complex frequency behavior. The next images show one input image, and the result of the strided convolutions with the blur kernel, $b_{2,2}$, with dilations $d_k = 1$, $d_k = 2$ and $d_k = 4$.

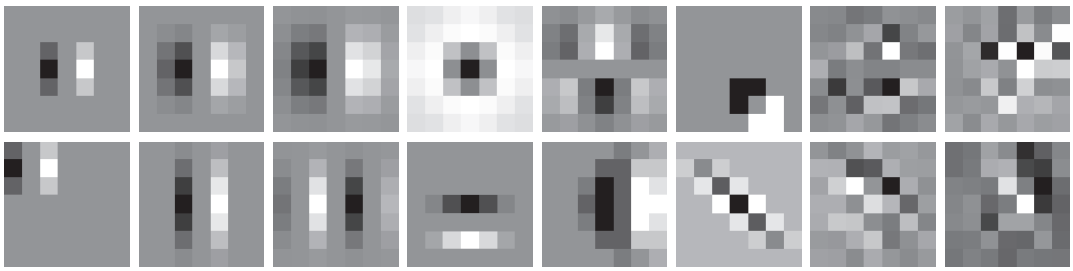


In summary, using dilation increases the size of the convolution kernels without increasing the computations (which is the original desired property) but it reduces the space of useful kernels (which is an undesired property).

There are ways in which dilation can be used to increase the family of useful filters. For instance, by concatenating three convolutions with $d_k = 1$, $d_k = 2$ and $d_k = 4$ together, one can create a kernel that can switch during learning between high and low spatial frequencies and small and large kernels.



This results on a kernel with a size of 9×9 (81 values) defined by 27 values. The relative computational efficiency increases when we cascade more filters with higher levels of dilation. The next figure shows several multiscale kernels that can be obtained by the convolutions of three dilated kernels. Can you guess which kernels were used?



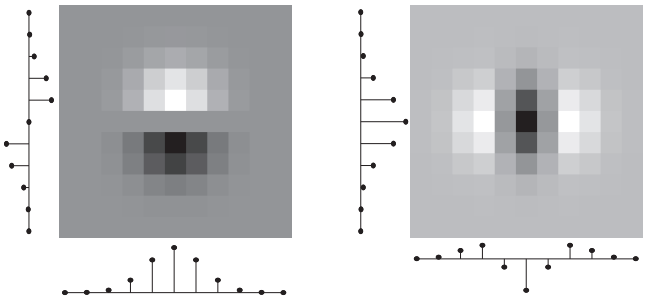
As the figure shows, the cascade of three dilated convolutions can generate a large family of filters with different scales, orientations, shifts, and also other pattern such as corner detectors, long edges and curved edge detectors. The last 4 kernels shows the result of convolving three random kernels which provides further illustration of the diversity of kernels one can build. Each kernel is 3x3 array sampled from a gaussian distribution.

1.1.4 Low-rank filters

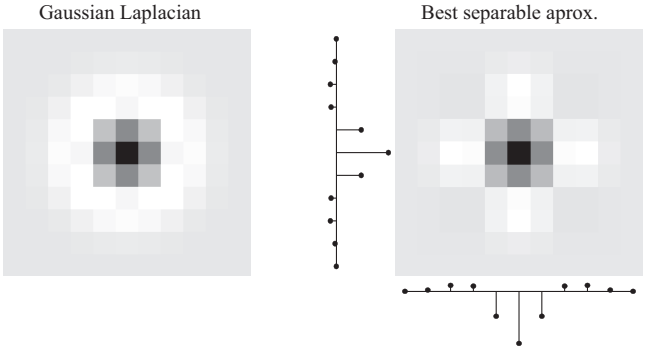
Dilation is one way to create a big filter that is parameterized by just a small number of weights, that is, a rank-deficient filter. This trick can be useful in many contexts where we know that good filters have low-rank structure. Dilation uses the trick to make big receptive fields which can capture long-range dependences.

Separable filters are another kind of low-rank filter that is useful in many applications (see Chapter XX). We can create a conv layer with separable filters by simply stacking two conv layers in sequence, with no other layers in between. The first layer is a filter bank with $K \times 1$ kernels and the second uses $1 \times K$ kernels. The composition of these layers is equivalent to a single conv layer with $K \times K$ separable filters.

When convolving one row and one column vector, $\mathbf{w} = \mathbf{u}^T \circ \mathbf{v}$, the result is the outer product: $w[n, m] = u[n]\, v[m]$.



Some important kernels are non separable but can be approximated by a linear combination of a small number of separable filters. For instance, the Gaussian Laplacian is non-separable but can be approximated by a separable filter as shown here:



The diagonal Gaussian derivative is another non-separable kernel. When using a 3×3 kernel to approximate it we have:

$$\mathbf{w} = \begin{bmatrix} 0 & -2 & -2 \\ 2 & 0 & -2 \\ 2 & 2 & 0 \end{bmatrix} \tag{1.8}$$

But we know from Chapter XX that this kernel can be written as a linear combination of two separable kernels: $\mathbf{w} = \text{Sobel}_x + \text{Sobel}_y$, as defined in Equations **??**. In general, any $M \times N$ filter can be decomposed as a linear sum of $min(N, M)$ separable filters. The separable filters can be obtained by applying the SVD decomposition to the kernel array $\mathbf{w}$. The SVD decomposition results in three matrices, $\mathbf{U}$, $\mathbf{S}$ and $\mathbf{V}$, so that $\mathbf{w} = \mathbf{U}\mathbf{S}\mathbf{V}^T$, where the columns of $\mathbf{U}$ and $\mathbf{V}$ are the separable 1D filters and the diagonal values of the diagonal matrix $\mathbf{S}$ are the linear weights. Computational benefits are only obtained when using small linear combinations for large kernels. Also, in a neural network, one could use only separable filters for all the units and the learning could discover ways of combining them in order to build more complex, non-separable kernels.

1.1.5 Cross-correlation layers

Most implementations of "convolution" layers actually use cross-correlation rather than convolution, computing $\mathbf{w} \star \mathbf{x}_{\texttt{in}} + b$. The reason we do not worry about the misnomer is that whether you implement the layers with convolution or cross-correlation usually makes no difference for learning. This is because convolution and cross-correlation *span an identical hypothesis space* (any cross-correlation can be converted to an equivalent convolution by flipping the filter horizontally and vertically). However, sometimes it will be important to know whether the filters are doing convolution or cross-correlation. This may come up when examining the learned filters and trying to understand them, or when manipulating filters post-learning.

1.1.6 Downsampling and upsampling layers

In Chapter XX we saw image pyramids and showed how they can be used for analysis and synthesis. CNNs can also be structured as analysis and synthesis pyramids, and this is a very powerful tool. To create a pyramid we just need to introduce a way of downsampling the signal during analysis and upsampling during synthesis. In CNNs this is done with **downsampling and upsampling layers**.

Downsampling layers transform the input tensor to an output tensor that is smaller in the spatial dimensions: $\mathbb{R}^{N \times M} \rightarrow \mathbb{R}^{N/s_n \times M/s_m}$. We already saw one kind of downsampling layer, strided convolution, which is equivalent to convolution followed by subsampling. Another common kind of downsampling layer is **pooling**, which we will encounter below.

Upsampling layers perform the opposite transformation, outputting a tensor that is larger in the spatial dimensions than the input: $\mathbb{R}^{N \times M} \rightarrow \mathbb{R}^{N * s_n \times M * s_m}$. One kind of upsampling layer can be made as the analogue of strided convolution. Strided convolution convolves then subsamples; this upsampling layer instead dilates the signal then convolves:

$$\mathbf{h}[n, m] = \mathbf{x}_{\texttt{in}}[n * s_n, m * s_m] \quad \triangleleft \ \texttt{dilation} \tag{1.9}$$

$$\mathbf{x}_{\texttt{out}} = \mathbf{w} \circ \mathbf{h} + b \quad \triangleleft \ \texttt{conv} \tag{1.10}$$

Sometimes the combination of these two layers is called an "UpConv" layer or a "Deconvolution" layer (but note that deconvolution has a different meaning in traditional signal processing).

1.2 Nonlinear filtering layers

All the operations we have covered above are linear (or affine). It is also possible to define filters that are nonlinear. Like linear convolutional filters, these filters slides across the input tensor and process each window identically and independently, but the operation they appear is a nonlinear function of the local window.

1.2.1 Pooling layers

Pooling layers are downsampling layer that summarize the information in a patch using some aggregate statistic, such as the patch's mean value, called **mean pooling**, or its max value, called **max pooling**, defined as follows:

$$x_{\texttt{out}_i} = \max_{i \in \mathcal{N}(i)} x_{\texttt{in}_i} \quad \triangleleft \ \texttt{max pooling} \tag{1.11}$$

$$x_{\texttt{out}_i} = \frac{1}{|\mathcal{N}|} \sum_{i \in \mathcal{N}(i)} x_{\texttt{in}_i} \quad \triangleleft \ \texttt{mean pooling} \tag{1.12}$$

Like all downsampling layers, pooling layers can be used to reduce the resolution of the input tensor, removing high-frequency information in the signal. Pooling is also particularly useful as a way to achieve *invariance*. Convolutional layers produce outputs that are

equivariant to translations of the their input. Pooling is a way to convert equivariance into invariance. For example, suppose we have run a convolutional filter that detects vertical edges. The output is a response map that is large wherever there was a vertical edge in the input image. Now if we run a max pooling filter across this response map, it will coarsen the map, resulting in a large response anywhere *near* where there was a vertical edge in the input image. If we use a max pooling filter with large enough receptive field, the output will be invariant to the location of the edge in the input image.

Pooling can also be performed across channels, and this can be a way to achieve additional kinds of invariance. For example, suppose we have a convolutional layer that applies a filter bank of oriented edge detector filters, where each filter looks for edges at a different orientation. Now if we max pool across the channels output by this filter bank, the resulting feature map will be large wherever an edge of *any* orientation was found. Normally, we are not looking for edges but for more complicated patterns, but the same logic applies. First run a bank of filters that look for the pattern at k different orientations. Then pool across these k channels to detect the pattern regardless of its orientation. This can be a great way for a CNN to recognize objects even if they appear with various rotations within the image. Of course we usually do not hand-define this strategy but it is one the CNN can learn to use if given channel-wise pooling layers.

1.2.2 Global pooling layers

One extreme of pooling is to pool over the entire spatial extend of the feature map. Global pooling is a function that maps a $M \times N \times C$ tensor into a vector of length C, where C is the number of channels in the input.

Global pooling is generally used in layers very close to the output. As before, global pooling can be **average global pooling**, averaging over all the responses of the feature map, or **global max pooling**, taking the max of the feature map.

Global pooling removes spatial information from each channel. However, spatial information about input features might be still be available within the output vector.

1.2.3 Local normalization layers

Another kind of nonlinear filter is the **local normalization layer**. These layers normalizes each activation in a feature map by statistics the adjacent activations within some neighborhood. There are many different choices for the type of normalization – L_1 norm, L_2 norm, standardization, etc – and many different choices for the shape of the neighborhood – a square patch in the spatial dimensions, a column of channels, etc. Each of these choices leads to different kinds of normalization filters with different names. One that is historically important but no longer frequently used is the **Local Response Normalization**, or **LRN**, filter that was introduced in the AlexNet paper [Krizhevsky et al. 2012]. This filter has the following form:

$$x_{\texttt{out}_k}[n,m] = x_{\texttt{in}_k}/(\gamma + \alpha \sum_{i=\max(1,k-l)}^{\max(K,k+l)} x_{\texttt{in}_i}[n,m]^2)^\beta \qquad \triangleleft \texttt{LRN} \tag{1.13}$$

α, β, γ, and l are hyperparameters of the layer. This layer normalizes each activation by the sum of squares of the activations in a window of adjacent *channels*.

Although local normalization is a common structure within the brain, it is not very frequent in current neural networks, which more often use global normalization layers like batchnorm (which we saw in Chapter **??**).

1.3 CNNs: using convolutional layers in deep nets

Stacking sequences of layers `conv`, `non-linearity`, `subsample` over and over results in a prototypical motif of CNNs: This structure is common for CNNs whose goal is to take an

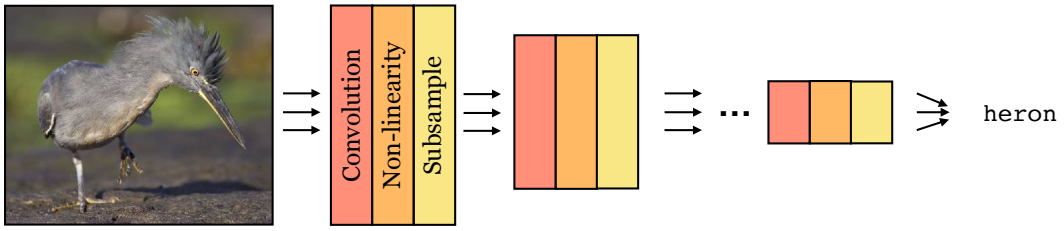


image as input and produce a label as output, such as in the problem of image classification. We will now walk through an example:

Let $\mathbf{x} \in \mathbb{R}^{M \times N}$ be a black and white image. To process this image, we could use a simple CNN with two convolutional layers, defined as follows:

$$\mathbf{z}_{1_i} = \mathbf{w}_i \circ \mathbf{x} + \mathbf{b}_i \qquad \triangleleft \quad \texttt{conv} : [M \times N] \to [M \times N \times C] \tag{1.14}$$

$$\mathbf{h}_i = \max(\mathbf{z}_{1_i}, 0) \qquad \triangleleft \quad \texttt{relu} : [M \times N \times C] \to [M \times N \times C] \tag{1.15}$$

$$\mathbf{z}_{2_i} = \frac{1}{NM} \sum_{n,m} \mathbf{h}_i[n, m] \qquad \triangleleft \quad \texttt{gap} : [M \times N \times C] \to [C] \tag{1.16}$$

$$\mathbf{z}_3 = \mathbf{W}\mathbf{z}_2 + \mathbf{c} \qquad \triangleleft \quad \texttt{fc} : [C] \to [K] \tag{1.17}$$

$$\mathbf{y}_i = \frac{e^{-\tau \mathbf{z}_{3_i}}}{\sum_{k=1}^{K} e^{-\tau \mathbf{z}_{3_k}}} \qquad \triangleleft \quad \texttt{softmax} : [K] \to [K] \tag{1.18}$$

This network has one convolutional layer with C channels followed by relus. The next layer performs spatial global average pooling (`gap`), and each channel gets projected into a single number that contains the sum of the outputs of the relu. This results in a representation given by a vector of length C. This vector is then processed by a linear fully connected layer (described by a $C \times K$ array).

This neural net could be used to solve a K-way image classification problem (because the output is a K-way softmax for each input image). We could train it using gradient descent to find the parameters $\theta = [\mathbf{w}_1, \ldots, \mathbf{w}_C, \mathbf{b}_1, \ldots, \mathbf{b}_C, \mathbf{W}, \mathbf{c}]$ that optimize a cross-entropy loss over training data.

Here it is in code:

```
# w, b, W, c : parameters of the CNN
# x : input data

# first define parameterized layers
conv1 = nn.conv(w, b)
fc1 = nn.fc(W,b)

# then run data through network
z1 = conv1(x)
h = nn.relu(z1)
z2 = nn.AvgPool2d(h)
z3 = fc1(z2)
y = nn.softmax(z3)
```

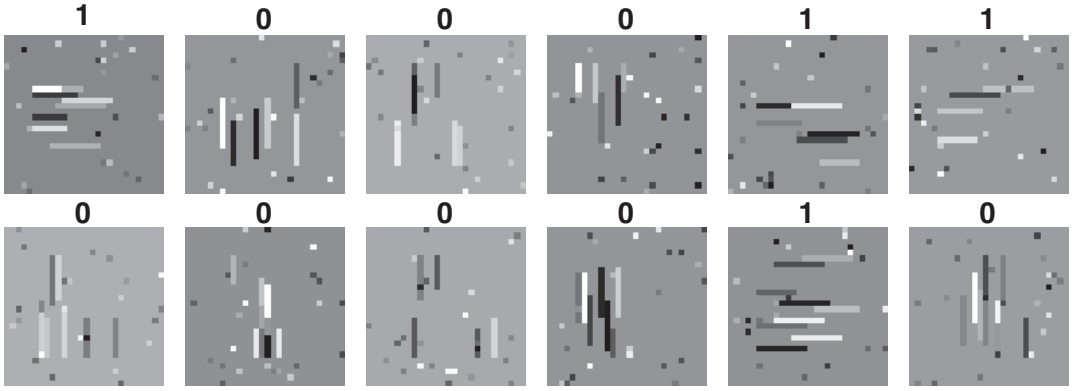
1.4 A running example

In this section, we will analyze the simple network described above trained to discriminate between horizontal and vertical lines. Each subsection will tackle one aspect of the analysis that should be part of training any large system: 1) training and evaluation, 2) visualize and understand the network, 3) out of domain generalization and 4) identifying vulnerabilities.

1.4.1 Training, and evaluation

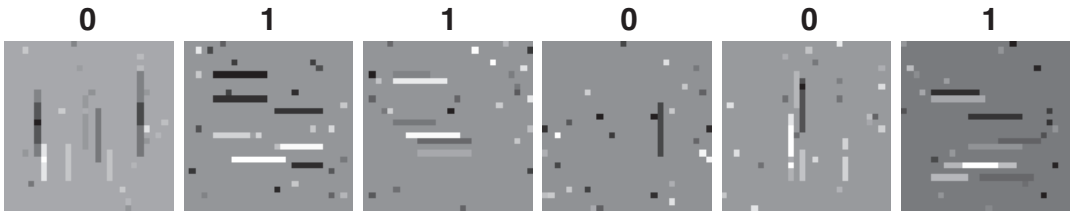
Let's study one simple classification task. We design a simple image dataset than contains images with lines. The lines can be horizontal or vertical. Each image will contain only one type of lines.

We want to design a CNN that will classify the image according to the orientation of the lines that it contains. We define the two output classes as: 0 (vertical) and 1 (horizontal). The training set looks like this:



The training set defines the concept we want to learn. In this case we look for lines. But images of lines might not be enough to describe the concept of a line. What is a line? This lack of a precise definition will hunt us later.

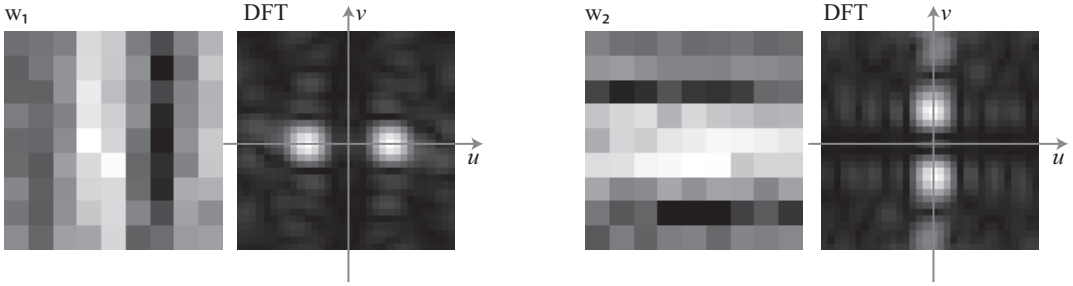
To solve this problem we use the CNN defined before with two convolutional channels $C = 2$ in the first layer. Once we train the network, we can see that it has solve the task perfectly and that the output on the test set is 100% correct (there are only 3 errors out of 10000 test images). Here there are some examples from the test set:



1.4.2 Network visualization

What has the network learnt? How is it solving the problem? One important part of developing a system is to have tools to prove, understand and debug it.

To understand the network it is useful to **visualize** the kernels. The following image shows the two learned 9×9 kernels. The first one looks like an horizontal derivative of a Gaussian filter (as we saw in chapter YY) and the second one looks like a vertical derivative of a Gaussian (maybe closer to a 2nd derivative). In fact, the DFT of each kernel shows that they are quite selective to a particular band on frequency content in the image.



The fully connected layer has learnt the weights:

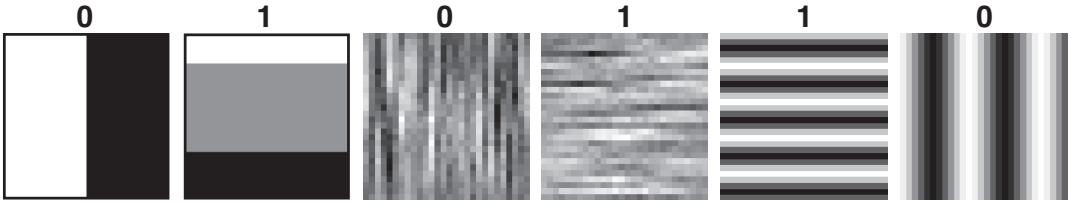
$$\mathbf{W} = \begin{bmatrix} 2.83 & -2.36 \\ -0.60 & 1.14 \end{bmatrix}$$

which corresponds to two channel oppositions: the first feature is the vertical output minus the horizontal output, and the second feature computes the horizontal output minus the vertical one.

1.4.3 Out of domain generalization

What do we learn by analyzing how the trained network works? One interesting outcome is that can we predict how the network we defined before generalizes beyond the distribution of the training set: **out of domain test samples**.

For instance, it seems natural to think that the network should still perform well in classifying whether the image contains vertical or horizontal structures even if they are not lines. We can test this hypothesis by generating images that match our idea of orientation. The following test images contain different oriented structures but no lines, and still captures our notion of what should be the correct generalization of the behavior:

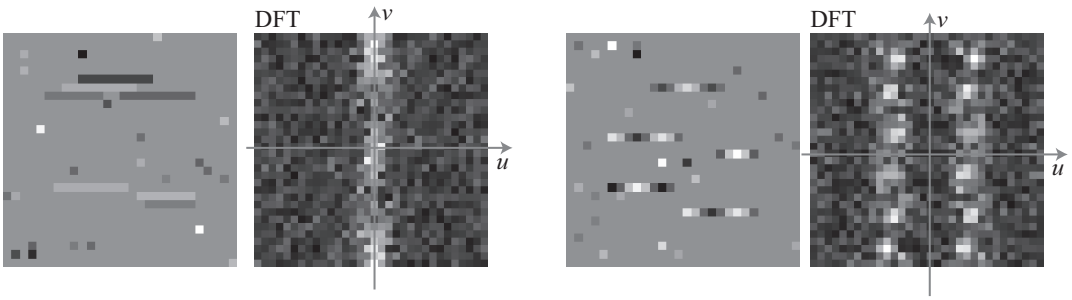


In fact, the network seems to perform correctly even with this new images that come from a distribution different from the training set.

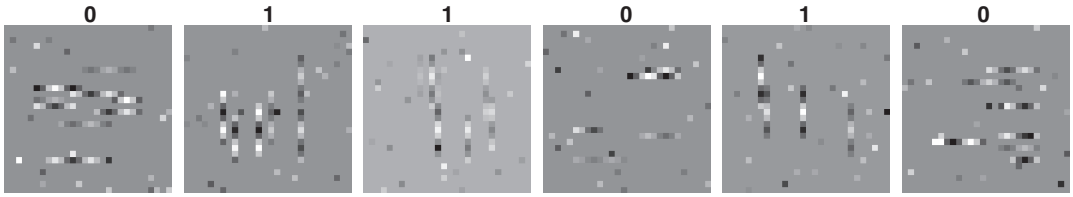
1.4.4 Identifying vulnerabilities

Does the network solve the task that we had in mind? Can we predict what inputs will make the output fail? Can we produce test examples that to us look right but that the network produces the wrong classification output? The goal of this analysis is to identify weaknesses in the learned representation and in our training set (missing training examples, biases in our data, limitations of the architecture, ...).

We see that the output of the first layer does not really looks for *lines*, instead it looks at where the energy is in the Fourier domain. So, we could fool the classifier by creating lines that for us look like vertical lines, but that have the energy content in the wrong side of the Fourier domain. We say one trick to do this on chapter XXX: modulation. If we multiply an image containing horizontal lines, by a sinusoidal wave, $\cos(\pi n/3)$, we can move the spectral content horizontally as shown in the next figure:



The lines still look horizontal to us, but the spectral content now is higher in the region that overlaps with the vertical line detector learned by the network. Indeed, when the network processes images that have lines with this *sinusoidal texture* produces the wrong classification results for all the images!



We have just designed an **adversarial example** manually! The question then could be: if it is not detecting line orientations, what is it really detecting? Our analysis of the learned kernels had the answer.

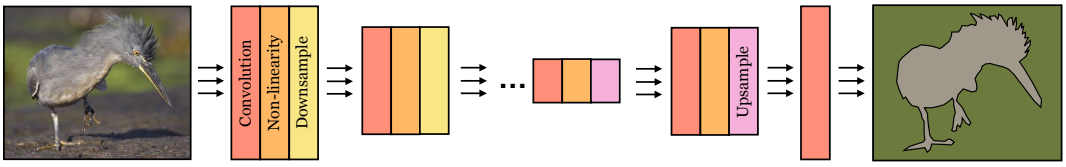
One way of avoiding this would be to introduce these types of images in the training set and to repeat the whole process.

For complex architectures, **Adversarial examples** are obtained as an optimization problem: what is the minimal perturbation of an input that will produce the wrong output in the network?

1.5 Encoder-Decoders

An **encoder maps** from data to a representation of that data, which is usually lower dimensional. A **decoder** performs the inverse operation, mapping from the representation back to the data. In the image modeling chapters we used the terms "analysis" for encoding and "synthesis" for decoding; in the context of neural nets, "encoder" and "decoder" are the more common terms but the meaning is the same.

A common architecture for an encoder is a sequence of conv layers that downsample. This is a way to reduce dimensionality. Decoders are the mirror image: they may be a sequence of conv layers that upsample. The result looks like this:



1.5.1 Skip connections

Encoder-decoders force the signal to pass through an **information bottleneck**: the encoder maps a high-dimensional signal to a low-dimensional encoding, which has limited capacity to store information, and the decoder must somehow reconstruct a high-dimensional output from the encoding. Although this can be a very efficient way to process data it also can cause the net to fail to be able to output high frequency details, so, e.g., a predicted label map might be very coarse.

To circumvent this problem, we can add **skip connections** that shuttle information directly across blocks of layers in the net. A skip connection f is simply an identity pathway that connects two disparate layers of a net, $f(\mathbf{x}) = x$. The output of a skip connection must somehow be reintegrated into the network. Two common choices are concatenating the output with the activations on the later layer, or adding the outputs.

Adding skip connections to an encoder-decoder results in an architecture known as a **U-net** [Ronneberger et al. 2015]. In this architecture, skip connections are arranged in a mirror pattern, where layer l is connected directly to layer $(L - l)$. This architecture can maintains the information-bottleneck of the encoder-decoder, with its incumbent benefits in terms of memory and compute efficiency and forced abstraction, while also allowing residual

information to flow through the skip connections, thereby not sacrificing the ability to output high-frequency spatial predictions. U-nets look a lot like the steerable pyramids from Chapter **??**, which also consistent of a downsampling *analysis* path followed by an upsampling *synthesis* path, with skip connections in between mirror image stages in the pathways. The main difference is that the U-net has *learned filters* and *nonlinearities*.

ResNets were also invented under the name Highway Networks [Srivastava et al. 2015]

Another popular architecture that uses skip connections is called **Residual Networks** or **ResNets** [He et al. 2016]. In the context of ResNets, skip connections are called **residual connections**. This kind of skip connection is added to the output of a block of layers F:

$$\mathbf{x}_{\texttt{out}} = F(\mathbf{x}_{\texttt{in}}) + \mathbf{x}_{\texttt{in}} \qquad \triangleleft \texttt{residual block} \tag{1.19}$$

In a **residual block** like this, You can think of F as a *residual* that additively perturbs $\mathbf{x}_{\texttt{in}}$ to transform it into an improved $\mathbf{x}_{\texttt{out}}$. If $\mathbf{x}_{\texttt{out}}$ does not have the same dimensionality as $\mathbf{x}_{\texttt{in}}$, then we can add a linear mapping to convert the dimensions: $\mathbf{x}_{\texttt{out}} = F(\mathbf{x}_{\texttt{in}}) + \mathbf{W}\mathbf{x}_{\texttt{in}}$.

It is easy for a residual block to simply perform an identity mapping, it just has to learn to set F to zero. Because of this, if we stack many residual blocks in a row, it can end up that the net learns to use only a subset of them [**?**]. If we set the number of stacked residual blocks to be very large then the net can essentially learn how deep to make itself, using as many blocks as necessary to solve the task. ResNets often exploit this fact by being very deep; for example, hundreds of residual blocks deep is not uncommon.

1.6 CNNs as patch processing

As mentioned at the start of this chatper, one way to think about convolutional layers is that they are *patch processing*: an input signal is cut up into chunks and then each chunk is processed independently and identically. To see this, we can rewrite Eqn. 1.1 as a chopping up and processing algorithm:

Algorithm 1: Convolution layer

Input: $\mathbf{x}_{\texttt{in}} \in \mathbb{R}^{M\times N}$, $\mathbf{w} \in \mathbb{R}^{2p+1\times 2p+1}$, $b \in \mathbb{R}$, **Output:** $\mathbf{x}_{\texttt{out}} \in \mathbb{R}^{M\times N}$
for $i = 1, \ldots, M$ **do**
  for $j = 1, \ldots, N$ **do**
    $\tilde{\mathbf{x}}_{\texttt{in}}[i * N + j] = \mathbf{x}_{\texttt{in}}[i - p : i + p, j - p : j + p]$ ◁ Chop up image into patches
for $i = 1, \ldots, MN$ **do**
  $\tilde{\mathbf{x}}_{\texttt{out}}[i] = \mathbf{w}^T \tilde{\mathbf{x}}_{\texttt{in}}[i] + b$ ◁ Process patches
for $i = 1, \ldots, MN$ **do**
  $k = \texttt{floor}(i/N)$, $l = i - kN$
  $\mathbf{x}_{\texttt{out}}[k - p : k + p, l - p) : l + p] = \tilde{\mathbf{x}}_{\texttt{out}}[i]$ ◁ Recombine patches into image

Note that padding must be applied for the values that extend outside the image boundaries. *Exercise*: convince yourself of the equivalence between this algorithm and Eqn. 1.1.

So far we have demonstrated that a *convolution layer* is equivalent to parallel patch processing. What about an entire *CNN*? It turns out a full CNN can also be rewritten as chopping up and processing each patch with the same sub-function, but this sub-function is not just an affine transformation – instead it is a neural net.

Suppose we have a CNN, which we define as a stack of conv layers, potentially with pointwise nonlinearities in between. By Alg. 1 , The final conv layer in the stack can be rewritten as breaking its inputs into patches and processing them with an affine transformation. Each patch of $\mathbf{x}_{\texttt{in}}$ for this layer is an array of neural activations output from the previous layer. Now we can trace back to see which activations input to layer $l-2$ contribute to the activations intput to layer $l-1$ in a given patch. If layer $l-2$ is pointwise, then it will be the same patch on layer $l-2$. If layer $l-2$ is convolutional, then it the receptive field

will grow by a factor related to the kernel size, padding, stride, and dilation of that layer. Now we can see that the mapping from layer $l-2$ to l is equivalent to chopping up layer $l-2$ activations into patches (slightly bigger than the patches before) and processing each with a two-layer CNN. Repeating this logic, we can see that the whole CNN is equivalent to chopping up the input into patches at some size and with some overlap, and then running an identical function f on each patch.

From this perspective, CNNs have two important properties: 1) each patch is processed independently from all the others, and 2) each patch is processed identically. We will next describe why each property is useful.

1.6.1 Property #1: treating patches as independent

This is a divide-and-conquer strategy. If you were to try to understand a complex problem, you might break it up into small pieces and solve each one separately. That's all a CNN is doing. We split up a big problem – "interpret this whole photo" – into a bunch of smaller problems – "interpret each small patch in the image."

Why is this a good strategy?

1. The small problems are easier to solve than the original problem.
2. The small problems can all be solved in parallel.
3. This approach is *agnostic to signal length* – you can solve an arbitrarily large problem just by breaking it down to bite size pieces and solving them "bird by bird"[Lamott 1980].

Chopping up into small patches like this is sufficient for many vision problems because the world exhibits **locality**: related things clump together, i.e. within a single patch; far apart things can be often be safely assumed to be independent.

1.6.2 Property #2: processing each patch identically

For images, convolution is an especially suitable strategy because visual content tends to be *translation invariant*, and, as we learned in previous chapters, the convolution operator is also translation invariant.

Typically, objects can appear anywhere in an image and look the same, like the birds in the photo above. This is because as the birds fly across the frame their position changes but their identity and appearance does not. More generally, as a camera pans across a scene, the content shifts in position but is otherwise unchanged.

Because the visual world is roughly translation invariant, it is justified to process each patch the same way, regardless of its position (i.e. its translation away from some canonical center patch).

"Translation invariant" just means we process each patch identically, using the same function f. Some texts instead use the term **translation equivariant** to describe convolutions. This places emphasis on the fact that if we shift the input signal by some translation, then the output signal will get shifted by the same amount. That is, if f is a convolution, we have the property:

$$f(\texttt{translate}(x)) = \texttt{translate}(f(x)) \tag{1.20}$$

1.7 Concluding remarks

We will see in later chapters that several new kinds of models are recently supplanting convnets are the most successful architectures for vision problems. One such architecture is the "transformer" (Chapter **??**). It may be tempting to think "why did we bother learning about convnets then, if transformers are better!" The reason we cover convnets is not

because the exact architecture presented here will last, but because the underlying principles it embodies are ubiquitous in sensory processing. The two key properties mentioned above are in fact present in transformers and many other "non-convolutional" architectures. Transformers also include stages that process each patch identically and independently, but they interleave these stages with other stages that globalize information across patches. It comes down to preferences whether you want to call these newer architectures convolutional or not, and there is currently some heated debate about it in the community. For us it doesn't matter, because if we learn the principles we can recognize them in all the systems we encounter and need not get hung up on names.

Bibliography

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105, 2012.

A. Lamott. *Bird by Bird*. Bantam Doubleday Dell Publishing Group, 1980.

O. Ronneberger, P. Fischer, and T. Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.

R. K. Srivastava, K. Greff, and J. Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.