CHAPTER 17

Simple Image Classifiers

There are two problems that lie at the core of image understanding. The first is **image classification**, where we decide what class an image of a fixed size belongs to. It's usual to work with a collection of images of objects. These objects will be largely centered in the image, and largely isolated. Each image will have an associated object name, using a taxonomy of classes provided in advance. You should think of catalog images of clothing or furniture. Another possible example is mugshot photos or pictures of people on websites (where the taxonomy is names). Judging by the amount of industry money pouring into image classification research, there are valuable applications for solutions.

The second problem is **object detection**, where we try to find the locations of objects of a set of classes in the image. So we might try to mark all cars, all cats, all camels, and so on. As far as anyone knows, the right way to think about object detection is that we search a collection of windows in an image, apply an image classification method to each window, then resolve disputes between overlapping windows. How windows are to be chosen for this purpose is an active and quickly changing area of research. Object detection is another problem receiving tremendous attention from industry.

Neural networks have enabled spectacular progress in both problems. We now have very accurate methods for large scale image classification and quite effective and fast methods for object detection. This chapter describes the main methods used in building these methods, and finishes with two fairly detailed examples of simple image classification. The next chapter covers modern methods for image classification and object detection.

17.1 IMAGE CLASSIFICATION

An instructive image classification dataset is the MNIST dataset of handwritten digits. This dataset is very widely used to check simple methods. It was originally constructed by Yann Lecun, Corinna Cortes, and Christopher J.C. Burges. You can find this dataset in several places. The original dataset is at http://yann.lecun.com/exdb/mnist/. The version I used was prepared for a Kaggle competition (so I didn't have to decompress Lecun's original format). I found it at http://www.kaggle.com/c/digit-recognizer.

Images have important, quite general, properties (Figure 17.1). Images of "the same thing" — in the case of MNIST, the same handwritten digit — can look fairly different. Small shifts and small rotations do not change the class of an image. Making the image somewhat brighter of somewhat darker does not change the class of the image either. Making the image somewhat larger, or making it somewhat smaller (then cropping or filling in pixels as required) does not change the class either. This means that individual pixel values are not particularly informative –

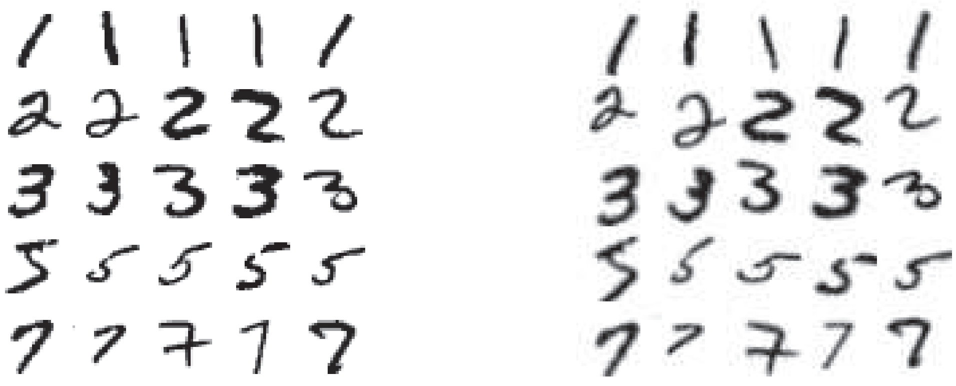


FIGURE 17.1: *On the* **left**, *a selection of digits from the MNIST dataset. Notice how images of the same digit can vary, which makes classifying the image demanding. It is quite usual that pictures of "the same thing" look quite different. On the* **right**, *digit images from MNIST that have been somewhat rotated and somewhat scaled, then cropped fit the standard size. Small rotations, small scales, and cropping really doesn't affect the identity of the digit.*

you can't tell whether a digit image is, for example, a zero by looking at a given pixel, because the ink might slide to the left or to the right of the pixel without changing the digit. In turn, you should not expect applying logistic regression directly to the pixel values to be particularly helpful. For MNIST, this approach yields an error rate that is quite poor compared to better methods (try it - `glmnet` can handle this).

Another important property of images is that they have many pixels. Building a fully connected layer where every unit sees every pixel is impractical – each unit might have millions of inputs, none of which is particularly useful. But if you think of a unit as a device for constructing features, this construction is odd, because it suggests that one needs to use every pixel in an image to construct a useful feature. This isn't consistent with experience. For example, if you look at the images in Figure 17.2, you will notice another important property of images. Local patterns can be quite informative. Digits like 0 and 8 have loops. Digits like 4 and 8 have crossings. Digits like 1, 2, 3, 5 and 7 have line endings, but no loops or crossings. Digits like 6 and 9 have loops and line endings. Furthermore, spatial relations between local patterns are informative. A 1 has two line endings above one another; a 3 has three line endings above one another. These observations suggest a strategy that is a central tenet of modern computer vision: you construct features that respond to patterns in small, localized neighborhoods; then other features look at patterns of *those* features; then others look at patterns of those, and so on.

17.1.1 Pattern Detection by Convolution

For the moment, think of an image as a two dimensional array of intensities. Write $\mathcal{I}_{uv}$ for the pixel at position u, v. We will construct a small array (a **mask** or **kernel**) $\mathcal{W}$, and compute a new image $\mathcal{N}$ from the image and the mask, using the

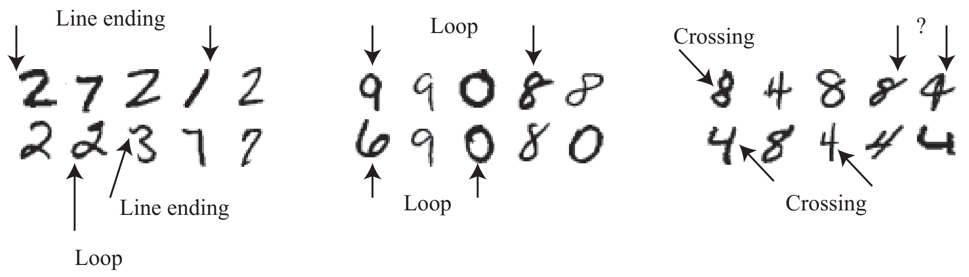


FIGURE 17.2: *Local patterns in images are quite informative. MNIST images, shown here, are simple images, so a small set of patterns is quite helpful. The relative location of patterns is also informative. So, for example, an eight has two loops, one above the other. All this suggests a key strategy: construct features that respond to patterns in small, localized neighborhoods; then other features that look at patterns of* those *features; then others that look at patterns of those, and so on. Each pattern (here line-endings, crossings and loops) has a range of appearances. For example, a line ending sometimes has a little wiggle as in the three. Loops can be big and open, or quite squashed. The list of patterns isn't comprehensive. The "?" shows patterns that I haven't named, but which appear to be useful. In turn, this suggests learning the patterns (and patterns of patterns; and so on) that are most useful for classification.*

rule

$$\mathcal{N}_{ij} = \sum_{kl} \mathcal{I}_{i-k,j-l}\mathcal{W}_{kl}.$$

Here we sum over all k and l that apply to $\mathcal{W}$; for the moment, do not worry about what happens when an index goes out of the range of $\mathcal{I}$. This operation is known as **convolution**. The form of the operation is important in signal processing mathematics, but makes it quite hard to understand what convolution is good for. We will generalize the idea.

Notice that if we flip $\mathcal{W}$ in both directions to obtain $\mathcal{M}$, we can write the new image as

$$\begin{aligned} \mathcal{N} &= \texttt{conv}(\mathcal{I}, \mathcal{M}) \\ &\quad \text{where} \\ \mathcal{N}_{ij} &= \sum_{kl} \mathcal{I}_{kl}\mathcal{M}_{k-i,l-j}. \end{aligned}$$

In what follows, I will always apply this flip, and use the term "convolution" to refer to the operator `conv` defined above. This isn't consistent with the signal processing literature, but is quite usual in the machine learning literature. Now reindex yet again, by substituting $u = k - i$, $v = l - j$, and noticing that if u runs over the range 0 to ∞, so does $u - i$ to get

$$\mathcal{N}_{ij} = \sum_{uv} \mathcal{I}_{i+u,j+v}\mathcal{M}_{uv}.$$

This operation is linear. You should check that:

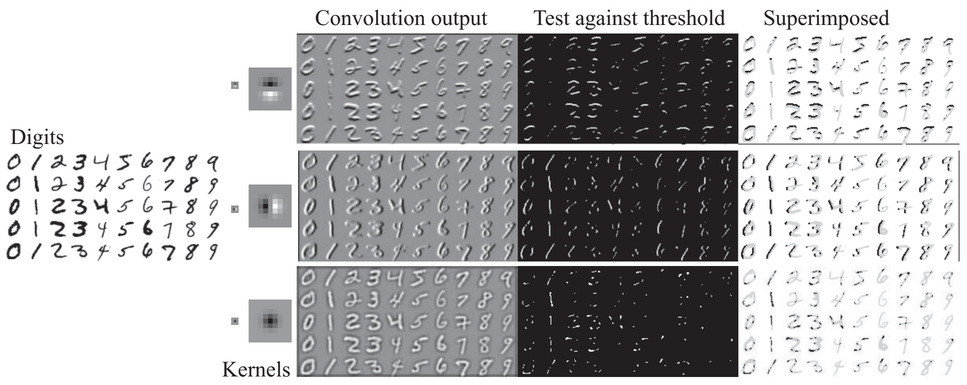


FIGURE 17.3: *On the* **far left***, some images from the MNIST dataset. Three kernels appear on the* **center left***; the small blocks show the kernels scaled to the size of the image, so you can see the size of the piece of image the kernel is applied to. The larger blocks show the kernels (mid-grey is zero; light is positive; dark is negative). The kernel in the top row responds most strongly to a dark bar above a light bar; that in the middle row responds most strongly to a dark bar to the left of a light bar; and the bottom kernel responds most strongly to a spot.* **Center** *shows the results of applying these kernels to the images. You will need to look closely to see the difference between a medium response and a strong response.* **Center right** *shows pixels where the response exceeds a threshold. You should notice that this gives (from top to bottom): a horizontal bar detector; a vertical bar detector; and a line ending detector. These detectors are moderately effective, but not perfect.* **Far right** *shows detector responses (in black) superimposed on the original image (grey) so you can see the alignment between detections and the image.*

- if $\mathcal{I}$ is zero, then $\texttt{conv}(\mathcal{I}, \mathcal{M})$ is zero;
- $\texttt{conv}(k\mathcal{I}, \mathcal{M}) = k\texttt{conv}(\mathcal{I}, \mathcal{M})$;
- and $\texttt{conv}(\mathcal{I} + \mathcal{J}, \mathcal{M}) = \texttt{conv}(\mathcal{I}, \mathcal{M}) + \texttt{conv}(\mathcal{J}, \mathcal{M})$.

The value of $\mathcal{N}_{ij}$ is a dot-product, as you can see by reindexing $\mathcal{M}$ and the piece of image that lies under $\mathcal{M}$ to be vectors. This view explains why a convolution is interesting: it is a very simple pattern detector. Assume that $\mathbf{u}$ and $\mathbf{v}$ are unit vectors. Then $\mathbf{u} \cdot \mathbf{v}$ is largest when $\mathbf{u} = \mathbf{v}$, and smallest when $\mathbf{u} = -\mathbf{v}$. Using the dot-product analogy, for $\mathcal{N}_{ij}$ to have a large and positive value, the piece of image that lies under $\mathcal{M}$ must "look like" $\mathcal{M}$. Figure 17.3 give some examples.

The proper model for `conv` is this. To compute the value of $\mathcal{N}$ at some location, you take the window $\mathcal{W}$ of $\mathcal{I}$ at that location that is the same size as $\mathcal{N}$; you multiply together the elements of $\mathcal{M}$ and $\mathcal{W}$ that lie on top of one another; and you sum the results (Figure 17.4). Thinking of this as an operation on windows allows us to generalize in very useful ways.

In the original operation, we used a window at every location in $\mathcal{I}$, but we may prefer to look at (say) a window at every second location. The centers of the windows we wish to look at lie on a grid of locations in $\mathcal{I}$. The number of pixels skipped between points on the grid is known as its **stride**. A grid with stride 1

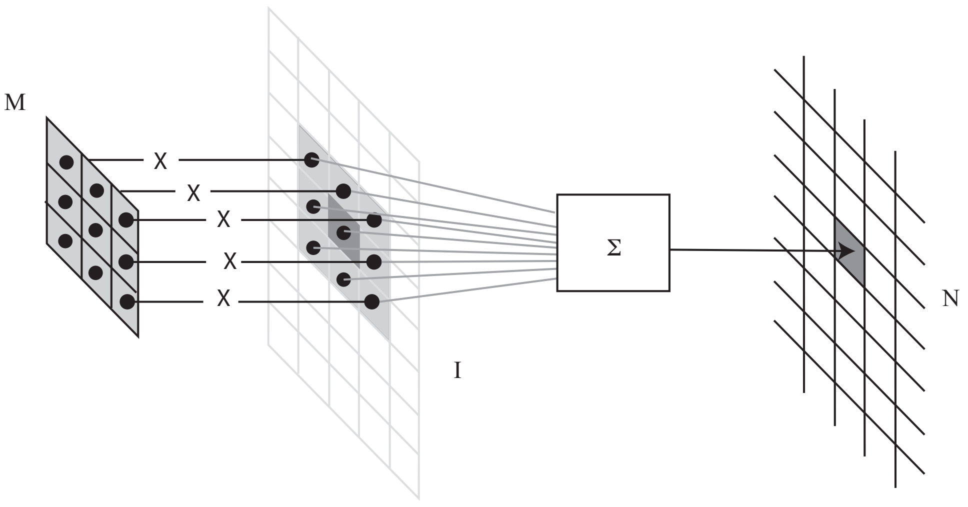


FIGURE 17.4: *To compute the value of $\mathcal{N}$ at some location, you shift a copy of $\mathcal{M}$ to lie over that location in $\mathcal{I}$; you multiply together the non-zero elements of $\mathcal{M}$ and $\mathcal{I}$ that lie on top of one another; and you sum the results.*

consists of each spatial location. A grid with stride 2 consists of every second spatial location in $\mathcal{I}$, and so on. You can interpret a stride of 2 as either performing `conv` then keeping the value at every second pixel in each direction. Better is to think of the kernel striding across the image — perform the `conv` operation as above, but now move the window by two pixels before multiplying and adding.

The description of the original operation avoided saying what would happen if the window at a location went outside $\mathcal{I}$. We adopt the convention that $\mathcal{N}$ contains entries only for windows that lie inside $\mathcal{I}$. But we can apply **padding** to $\mathcal{I}$ to ensure that $\mathcal{N}$ has the size we want. Padding attaches a set of rows (resp. columns) to the top and bottom (resp. left and right) of $\mathcal{I}$ to make it a convenient size. Usually, but not always, the new rows or columns contain zeros. By far the most common case uses $\mathcal{M}$ that are square with odd dimension (making it much easier to talk about the center). Assume $\mathcal{I}$ is $n_x \times n_y$ and $\mathcal{M}$ is $(2k+1) \times (2k+1)$; if we pad $\mathcal{I}$ with k rows on top and bottom and k columns on each side, $\texttt{conv}(\mathcal{I}, \mathcal{M})$ will be $n_x \times n_y$.

Images are naturally 3D objects with two spatial dimensions (up-down, left-right) and a third dimension that chooses a slice (R, G or B for a color image). This structure is natural for representations of image patterns, too — two dimensions that tell you where the pattern is and one that tells you what it is. The results in Figure 17.3 show a block consisting of three such slices. These slices are the response of a pattern detector *for a fixed pattern*, where there is one response for each spatial location in the block, and so are often called **feature maps**.

We will generalize `conv` and apply it to 3D blocks of data (which I will call **blocks**). Write $\mathcal{I}$ for an input block of data, which is now $x \times y \times d$. Two dimensions – usually the first two, but this can depend on your software environment – are

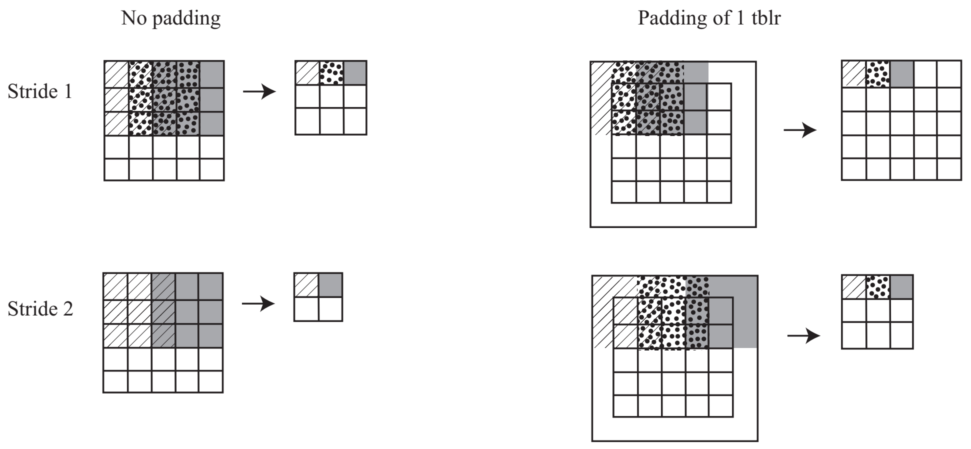


FIGURE 17.5: *The effects of stride and padding on* `conv`*. On the* **left**, `conv` *without padding accepts an $\mathcal{I}$, places a 3×3 $\mathcal{M}$ on grid locations determined by the stride, then reports values for valid windows. When the stride is 1, a 5×5 $\mathcal{I}$ becomes a 3×3 $\mathcal{N}$. When the stride is 2, a 5×5 $\mathcal{I}$ becomes a 2×2 $\mathcal{N}$. The hatching and shading show the window used to compute the corresponding value in $\mathcal{N}$. On the* **right**, `conv` *with padding accepts an $\mathcal{I}$, pads it (in this case, by one row top and bottom, and one column left and right), places a 3×3 $\mathcal{M}$ on grid locations in the padded result determined by the stride, then reports values for valid windows. When the stride is 1, a 5×5 $\mathcal{I}$ becomes a 5×5 $\mathcal{N}$. When the stride is 2, a 5×5 $\mathcal{I}$ becomes a 3×3 $\mathcal{N}$. The hatching and shading show the window used to compute the corresponding value in $\mathcal{N}$.*

spatial and the third chooses a slice. Write $\mathcal{M}$ for a 3D kernel, which is $k_x \times k_y \times d$. Now choose padding and a stride. This determines a grid of locations in the spatial dimensions of $\mathcal{I}$. At each location, we must compute the value of $\mathcal{N}$. To do so, take the 3D window $\mathcal{W}$ of $\mathcal{I}$ at that location that is the same size as $\mathcal{N}$; you multiply together the elements of $\mathcal{M}$ and $\mathcal{W}$ that lie on top of one another; and you sum the results (Figure 17.4). This sum now goes over the third dimension as well. This produces a two dimensional $\mathcal{N}$.

To make this operation produce a block of data, use a 4D block of kernels. This **kernel block** consists of D kernels, each of which is a $k_x \times k_y \times d$ dimensional kernel. If you apply each kernel as in the previous paragraph to an $x \times y \times d$ dimensional $\mathcal{I}$, you obtain an $X \times Y \times D$ dimensional block $\mathcal{N}$, as in Figure 17.6. What X and Y are depends on k_x, k_y, the stride and the padding. A **convolutional layer** takes a kernel block and a bias vector of D bias terms. The layer applies the kernel block to an input block (as above), then adds the corresponding bias value to each slice.

A convolutional layer is a very general and very useful idea. A fully connected layer is one form of convolutional layer. You can build a simple pattern detector out of a convolutional layer followed by a ReLU layer. You can build a linear map that reduces dimension can be built out of a convolutional layer.

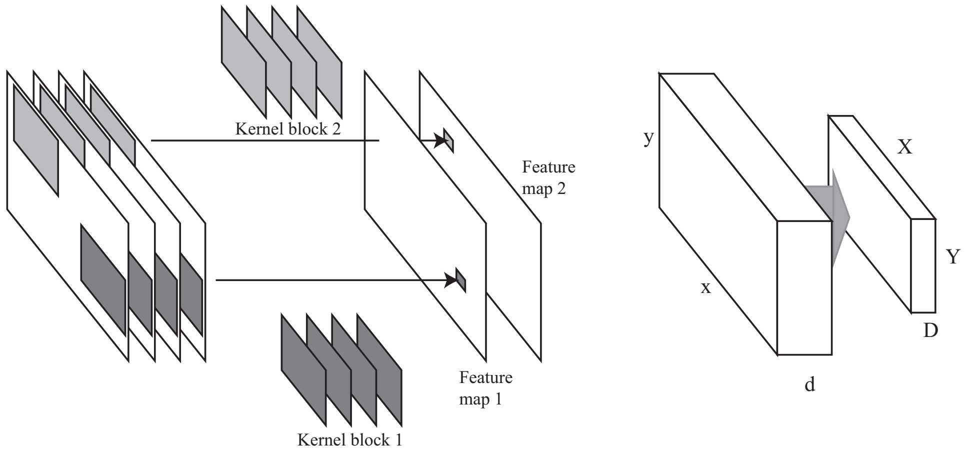


FIGURE 17.6: *On the* **left**, *two kernels (now 3D, as in the text) applied to a set of feature maps produce one new feature map per kernel, using the procedure of the text (the bias term isn't shown). Abstract this as a process that takes an $x \times y \times d$ block to an $X \times Y \times D$ block (as on the* **right**).

Useful Fact: 17.1 *Definition: Convolutional Layer*

A convolutional layer makes 3D blocks of data from 3D blocks of data, using a stride, padding, a block of kernels and a vector of bias terms. The details are in the text.

Remember this: *A fully connected layer can be thought of as a convolutional layer followed by a ReLU layer. Assume you have an $x \times y \times d$ block of data. Reshape this to be a $(xyd) \times 1 \times 1$ block. Apply a convolutional layer whose kernel block has size $(xyd) \times 1 \times D$, and then a ReLU. This pair of layers produces the same result as a fully-connected layer of D units.*

.

Remember this: *Take the output of a convolutional layer and apply a ReLU. First, think about what happens to one particular piece of image the size of one particular kernel. If that piece is "sufficiently similar" to the kernel, we will see a positive response at the relevant location. If the piece is too different, we will see a zero. This is a pattern detector as in Figure 17.3. What "sufficiently similar" means is tuned by changing the bias for that kernel. For example, a bias term that is negative with large magnitude means the image block will need to be very like the kernel to get a non-zero response. This pattern detector is (basically) a unit – apply a ReLU to a linear function of the piece of image, plus a constant. Now it should be clear what happens when all kernels are applied to the whole image. Each pixel in a slice represents the result of a pattern detector applied to the piece of image corresponding to the pixel. Each slice of the resulting block represents the result of a different pattern detector. The elements of the output block are often thought of as* **features**.

Remember this: *There isn't a standard meaning for the term convolutional layer. I'm using one of the two that are widely used. Software implementations tend to use my definition. Very often, research papers use the alternative, which is my definition followed by a non-linearity (almost always a ReLU). This is because convolutional layers mostly are followed by ReLU's in research papers, but it is more efficient in software implementations to separate the two.*

Different software packages use different defaults about padding. One default assumes that no padding is applied. This means that a kernel block of size $k_x \times k_y \times d \times D$ applied to a block of size $x \times y \times d$ with stride 1 yields a block of size $(n_x - k_x + 1) \times (n_y - k_y + 1) \times D$ (check this with a pencil and paper). Another assumes that the input block is padded with zeros so that the output block is $n_x \times n_y \times D$.

Remember this: *In Figure 17.3, most values in the output block are zero (black pixels in that figure). This is typical of pattern detectors produced in this way. This is an experimental fact that seems to be related to deep properties of images.*

Remember this: *A kernel block that is $1 \times 1 \times n_z \times D$ is known as a 1×1* **convolution**. *This is a linear map in an interesting way. Think of the input and output blocks as sets of column vectors. So the input block is a set of $n_x \times n_y$ column vectors, each of which has dimension $n_z \times 1$ (i.e. there is a column vector at each location of the input block). Write $\mathbf{i}_{uv}$ for the vector at location u, v in the input block, and $\mathbf{o}_{uv}$ for the vector at location u, v in the output block. Then there is a $D \times n_z$ matrix $\mathcal{M}$ so that the 1×1 convolution maps $\mathbf{i}_{uv}$ to*

$$\mathbf{o}_{uv} = \mathcal{M}\mathbf{i}_{uv}.$$

This can be extremely useful when the input has very high dimension, because $\mathcal{M}$ can be used to reduce dimension and *is learned from data.*

17.1.2 Convolutional Layers upon Convolutional Layers

Convolutional layers take blocks of data and make blocks of data, as do ReLU layers. This suggests the output of a convolutional layer could be passed through a ReLU, then connected to another convolutional layer, and so on. Doing this turns out to be an excellent idea.

Think about the output of the first convolutional layer. Each location receives inputs from pixels in a window about that location. The output of the ReLU, as we have seen, forms a simple pattern detector. Now if we put a second layer on top of this, each location in the second layer receives inputs from first layer values in a window about that location. This means that locations in the second layer are affected by a larger window of pixels than those in the first layer. You should think of these as representing “patterns of patterns”. If we place a third layer on top of the second layer, locations in that third layer will depend on an even larger window of pixels. A fourth layer will depend on a yet larger window, and so on. The key point here is that we can choose the patterns by *learning* what kernels will be applied at each layer.

The **receptive field** of a location in a data block (or, equivalently, a unit) is the set of image pixels that affect the value of the location. Usually, all that matters is the size of the receptive field. The receptive field of a location in the first convolutional layer will be given by the kernel of that layer. Determining the receptive field for later layers requires some bookkeeping (among other things, you must account for any stride or pooling effects).

If you have several convolutional layers with stride 1, then each block of data has the same spatial dimensions. This tends to be a problem, because the pixels that feed a unit in the top layer will tend to have a large overlap with the pixels that feed the unit next to it. In turn, the values that the units take will be similar, and so there will be redundant information in the output block. It is usual to try and

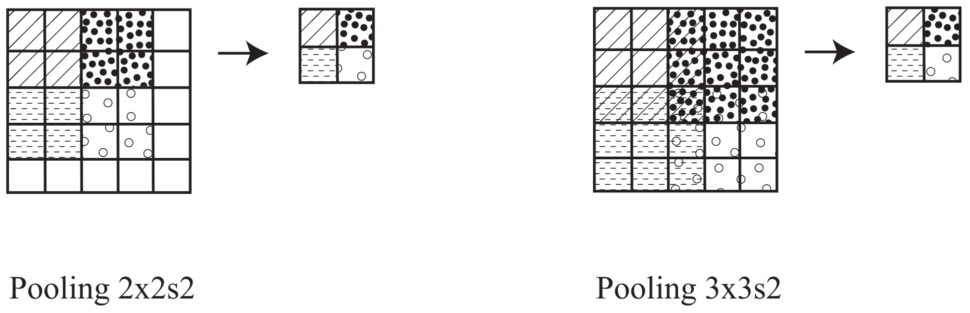


FIGURE 17.7: *In a pooling layer, pooling units compute a summary of their inputs, then pass it on. The most common case is 2x2, illustrated here on the* **left**. *We tile each feature map with 2x2 windows that do not overlap (so have stride 2). Pooling units compute a summary of the inputs (usually either the max or the average), then pass that on to the corresponding location in the corresponding feature map of the output block. As a result, the spatial dimensions of the output block will be about half those of the input block. On the* **right**, *the common alternative of pooling in overlapping 3x3 windows with stride 2.*

deal with this by making blocks get smaller. One natural strategy is to occasionally have a layer that has stride 2.

An alternative strategy is to use **pooling**. A pooling unit reports a summary of its inputs. In the most usual arrangement, a pooling layer halves each spatial dimension of a block. For the moment, ignore the entirely minor problems presented by a fractional dimension. The new block is obtained by pooling units that pool a window at each feature map of the input block to form each feature map of the output block. If these units pool a 2×2 window with stride 2 (ie they don't overlap), the output block is half the size of the input block. We adopt the convention that the output reports only valid input windows, so that this takes an $x \times y \times d$ block to an $\text{floor}(x/2) \times \text{floor}(y/2) \times d$ block. So, as Figure 17.7 shows, a $5 \times 5 \times 1$ block becomes a $2 \times 2 \times 1$ block, but one row and one column are ignored. A common alternative is pooling a 3×3 window with a stride of 2; in this case, a $5 \times 5 \times 1$ block becomes a $2 \times 2 \times 1$ block without ignoring rows or columns. Each unit reports either the largest of the inputs (yielding a **max pooling** layer) or the average of its inputs (yielding an **average pooling** layer).

17.2 TWO PRACTICAL IMAGE CLASSIFIERS

We can now put together image classifiers using the following rough architecture. A convolutional layer receives image pixel values as input. The output is fed to a stack of convolutional layers, each feeding the next, possibly with ReLU layers intervening. There are occasional max-pooling layers, or convolutional layers with stride 2, to ensure that the data block gets smaller and the receptive field gets bigger as the data moves through the network. The output of the final layer is fed to one or more fully connected layers, with one output per class. Softmax takes these outputs and turns them into class-probabilities. The whole is trained by batch

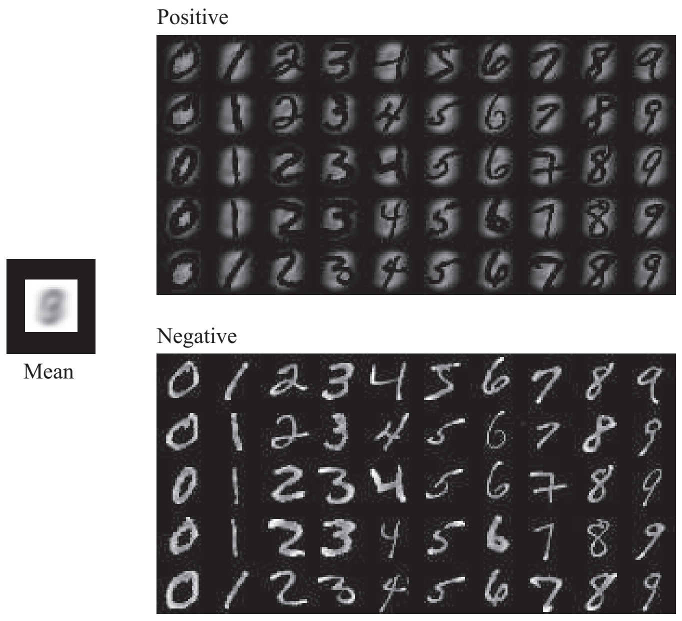


FIGURE 17.8: *The mean of MNIST training images is shown on the* **left***, surrounded by a black frame so that you can resolve it against the background. On the* **right***, the positive (***top***) and negative (***bottom***) components of the difference between mean and image for some training images. Lighter pixels have larger magnitude. Notice the blob of small positive values where there tends to be ink, and the strong negative values where this particular image has ink. This gives the network some information about where ink is expected to lie in general images, which seems to help training in practice.*

gradient descent, or a variant, as above, using a log-loss.

Notice that different image classification networks differ by relatively straightforward changes in architectural parameters. Mostly, the same thing will happen to these networks (variants of batch gradient descent on a variety of costs; dropout; evaluation). In turn, this means that we should use some form of specification language to put together a description of the architecture of interest. Ideally, in such an environment, we describe the network architecture, choose an optimization algorithm, and choose some parameters (dropout probability, etc.). Then the environment assembles the net, trains it (ideally, producing log files we can look at) and runs an evaluation. The tutorials mentioned in section 16.4.1 each contain examples of image classifiers for the relevant environments. In the examples shown here, I used Matconvnet, because I am most familiar with Matlab.

17.2.1 Example: Classifying MNIST

MNIST images have some very nice features that mean they are a good case to start with. Our relatively simple network architecture accepts images of a fixed size. This property is quite common, and applies to most classification architectures.

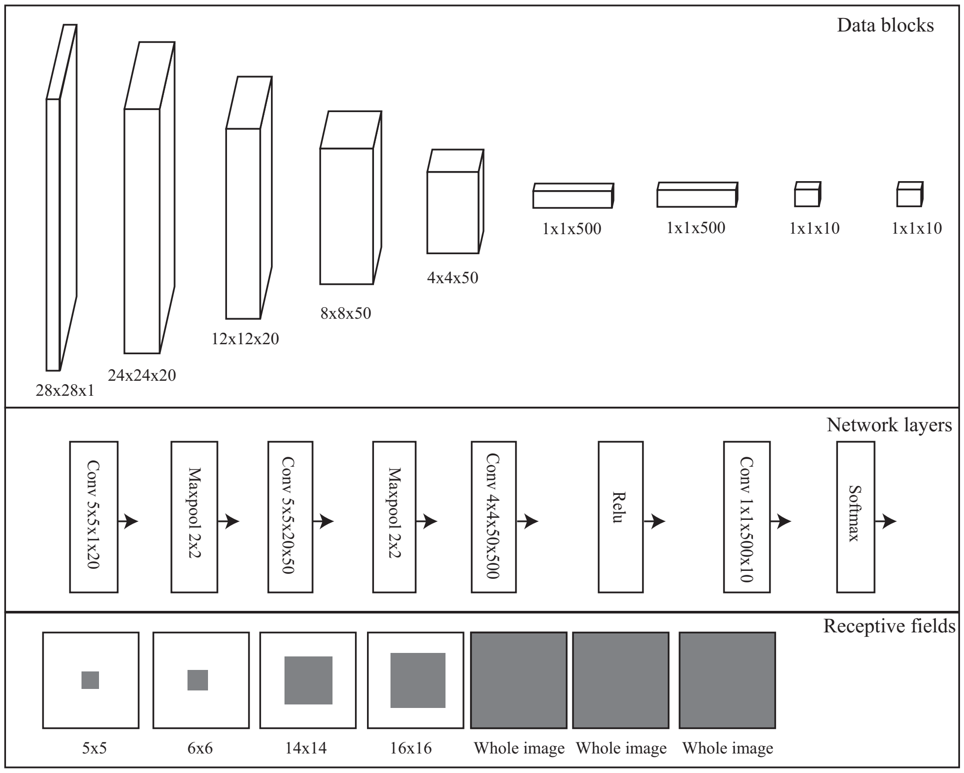


FIGURE 17.9: *Three different representations of the simple network used to classify MNIST digits for this example. Details in the text.*

This isn't a problem for MNIST, because all the MNIST images have the same size. Another nice feature is that pixels are either ink pixels or paper pixels – there are few intermediate values, and none of them are meaningful or helpful. In more general images, $\mathcal{I}$ and $0.9 \times \mathcal{I}$ show the same thing, just at different brightnesses. This doesn't happen for MNIST images. Yet another nice feature is that there is a fixed test-train split that everyone uses, so that comparisons are easy. Without a fixed split, the difference in performance between two networks might be due to random effects, because the networks see different test sets.

Much of the information in an MNIST image is redundant. Many pixels are paper pixels for every (or almost every) image. These pixels should likely be ignored by every classifier, because they contain little or no information. For other pixels, the value of the pixel is less important than how different the pixel is from the expected value at that location. Experience shows that it is surprisingly hard for neural networks to learn from heavily redundant image data. It is usual to preprocess images to remove some redundancies. For MNIST, the usual is to subtract the mean of the training images from the input image. Figure 17.8 shows how doing so seems to enhance the information content of the image.

Figure 17.9 shows the network used for this example. This network is a

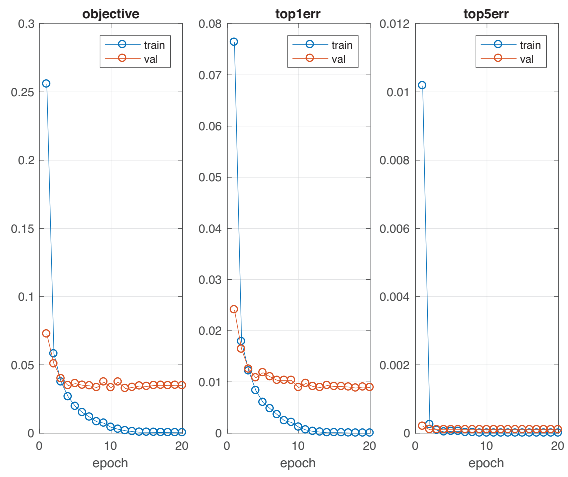


FIGURE 17.10: *This figure shows the results of training the network of Figure 17.9 on the MNIST training set. Loss, top-1 error and top-5 error for training and validation sets, plotted as a function of epoch for the network of the text. The loss (recorded here as "objective") is the log-loss. Note: the low validation error; the gap between train and validation error; and the very low top-5 error. The validation error is actually quite high for this dataset — you can find a league table at http:// rodrigob.github.io/ are_we_there_yet/ build/ classification_datasets_results.html.*

standard simple classification network for MNIST, distributed with Matconvnet. There are three different representations of the network here. The network layers representation, in the center of the figure, records the type of each layer and the size of the relevant convolution kernels. The first layer accepts the image which is a $28 \times 28 \times 1$ block of data (the data block representation), and applies a convolution. By convention, "conv $5 \times 5 \times 1 \times 20$" means a convolution layer, with a 20 different kernels each $5 \times 5 \times 1$. The effects of some of the learned kernels in this layer are visualized in Figure 17.11.

In the implementation I used, the convolution was not padded so that the resulting data block was $24 \times 24 \times 20$ (check that you know why this is correct). A value in this data block is computed from a 5×5 window of pixels, so the receptive field is 5×5. Again, by convention, every convolutional layer has a bias term, so the total number of parameters in the first layer is $(5 \times 5 \times 1) \times 20 + 20$ (check this statement, too). The next layer is a 2×2 max pooling layer, which again is not padded. This takes a $24 \times 24 \times 20$ block and produces a $12 \times 12 \times 20$ block. The receptive field for values in this block is 6×6 (you should check this with a pencil and paper drawing; it's right).

Another convolutional layer and another max-pooling layer follow, reducing the data to a $4 \times 4 \times 50$ block. Every value in this block is potentially affected by every image pixel, and this is true for all following blocks. Yet another convolutional layer reduces this to a $1 \times 1 \times 500$ block (again, where every value is potentially

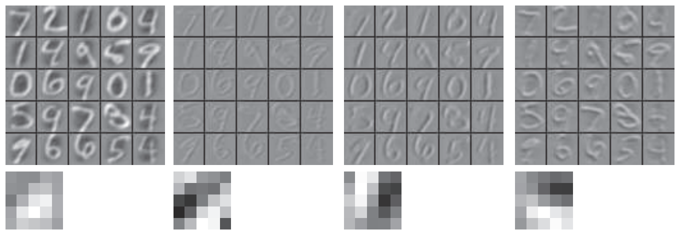


FIGURE 17.11: *Four of the 20 kernels in the first layer of my trained version of the MNIST network. The kernels are small (5×5) and have been blown up so you can see them. The outputs for each kernel on a set of images are shown above the kernel. The output images are scaled so that the largest value over all outputs is light, the smallest is dark, and zero is mid grey. This means that the images can be compared by eye. Notice that (rather roughly) the* **far left** *kernel looks for contrast;* **center left** *seems to respond to diagonal bars;* **center right** *to vertical bars; and* **far right** *to horizontal bars.*

affected by every pixel in the image). That goes through a ReLU (outputs visualized in Figure 17.12). You should think of the result as a 500 dimensional feature vector describing the image, and the convolutional layer and softmax that follow are logistic regression applied to that feature vector.

I trained this network for 20 epochs using tutorial code circulated with Matconvnet. Mini-batches are pre-selected so that each training data item is touched once per epoch, so an epoch represents a single pass through the data. It is common in image classification to report loss, top-1 error and top-5 error. Top-1 error is the frequency that the correct class has the highest posterior. Top-5 error is the frequency that the correct class appears in the five classes with largest posterior. This can be useful when the top-1 error is large, because you may observe improvements in top-5 error even when the top-1 error doesn't change. Figure 17.10 shows the loss, top-1 error and top-5 error for training and validation sets plotted as a function of epoch. This network has a low error rate, so of the 10, 000 test examples there are only 89 errors, which are shown in Figure 17.13.

17.2.2 Example: Classifying CIFAR-10

CIFAR-10 is a dataset of 32×32 color images in 10 categories, collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. It is often used to evaluate image classification algorithms. There are 50, 000 training images and 10, 000 test images, and the test-train split is standard. Images are evenly split between the classes. Figure 17.14 shows the categories, and examples from each category. There is no overlap between the categories (so "automobile" consists of sedans, etc. and "truck" consists of big trucks). You can download this dataset from https://www.cs.toronto.edu/kriz/cifar.html.

Figure 17.15 shows the network used to classify CIFAR-10 images. This network is again a standard classification network for CIFAR-10, distributed with Matconvnet. Again, I have shown the network in three different representations. The network layer representation, in the center of the figure, records the type of

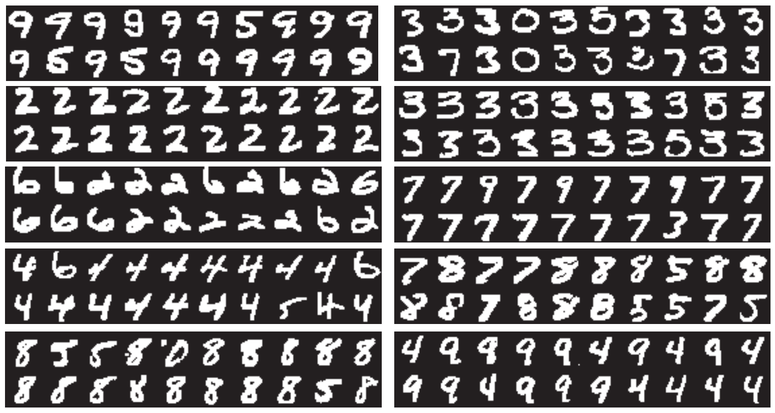
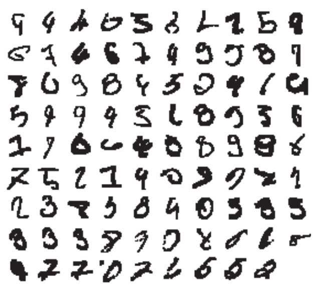


FIGURE 17.12: *Visualizing the patterns that the final stage ReLU's respond to for the simple CIFAR example. Each block of images shows the images that get the largest output for each of 10 ReLU's (the ReLU's were chosen at random from the 500 available). Notice that these ReLU outputs don't correspond to class – these outputs go through a fully connected layer before classification – but each ReLU are clearly responds to a pattern, and different ReLU's respond more strongly to different patterns.*

each layer and the size of the relevant convolution kernels. The first layer accepts the image which is a $32 \times 32 \times 3$ block of data (the data block representation), and applies a convolution.

In this network, the convolution *was* padded so that the resulting data block was $32 \times 32 \times 32$. You should check you agree with these figures, and you can tell by how much the image needed to be padded to achieve this (a drawing might help). A value in this data block is computed from a 5×5 window of pixels, so the receptive field is 5×5. Again, by convention, every convolutional layer has a bias term, so the total number of parameters in the first layer is $(5 \times 5 \times 3) \times 32 + 32$. The next layer is a 3×3 max pooling layer. The notation $3s2$ means that the pooling blocks



```
| 5 | 9 | 2 | 0 | 5 | 2 | 6 | 1 | 8 | 9 |
| 5 | 1 | 6 | 5 | 2 | 4 | 5 | 7 | 3 | 9 |
| 7 | 6 | 5 | 0 | 4 | 3 | 0 | 9 | 1 | 4 |
| 3 | 4 | 4 | 9 | 3 | 1 | 0 | 7 | 5 | 8 |
| 2 | 9 | 0 | 4 | 8 | 0 | 5 | 3 | 9 | 8 |
| 2 | 2 | 8 | 3 | 4 | 2 | 3 | 0 | 7 | 7 |
| 1 | 2 | 7 | 5 | 0 | 4 | 6 | 3 | 8 | 3 |
| 8 | 9 | 9 | 9 | 7 | 7 | 2 | 8 | 6 | 5 |
| 9 | 2 | 2 | 8 | 7 | 3 | 6 | 0 | 8 |
```

FIGURE 17.13: **Left:** *All 89 errors from the 10000 test examples in MNIST and* **right** *the predicted labels for these examples. True labels are mostly fairly clear, though some of the misclassified digits take very odd shapes.*

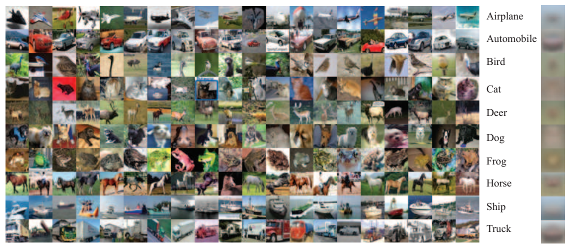


FIGURE 17.14: *The CIFAR-10 image classification dataset consists of 60, 000 images, in a total of 10 categories. The images are all 32x32 color images. This figure shows 20 images from each of the 10 categories and the labels of each category. On the* **far right***, the mean of the images in each category. I have doubled the brightness of the means, so you can resolve color differences. The per-category means are different, and suggest that some classes look like a blob on a background, and others (eg ship, truck) more like an outdoor scene.*

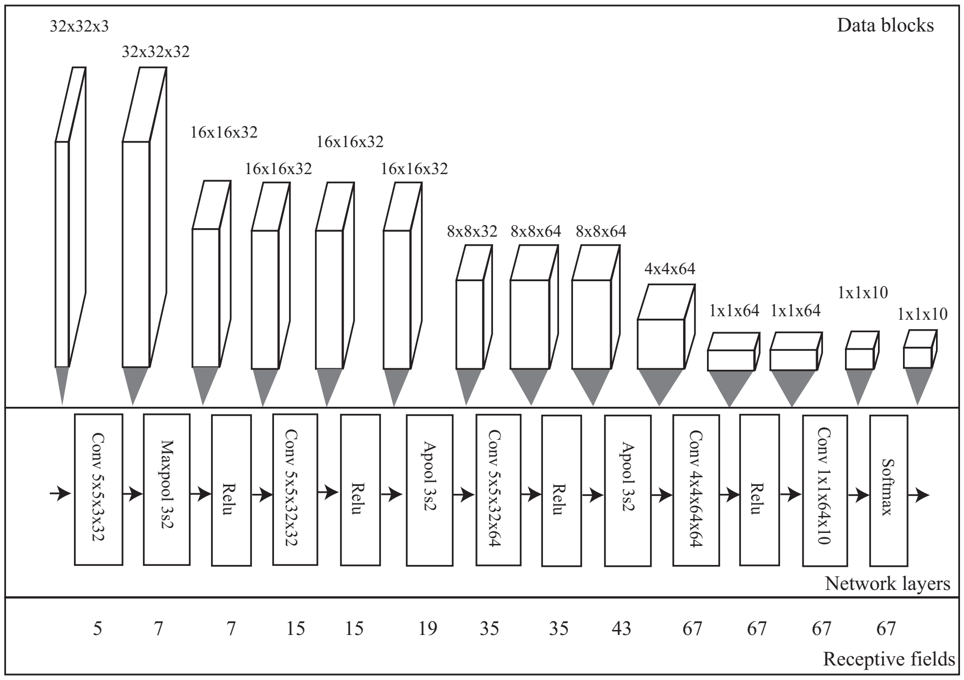


FIGURE 17.15: *Three different representations of the simple network used to classify CIFAR-10 images for this example. Details in the text.*

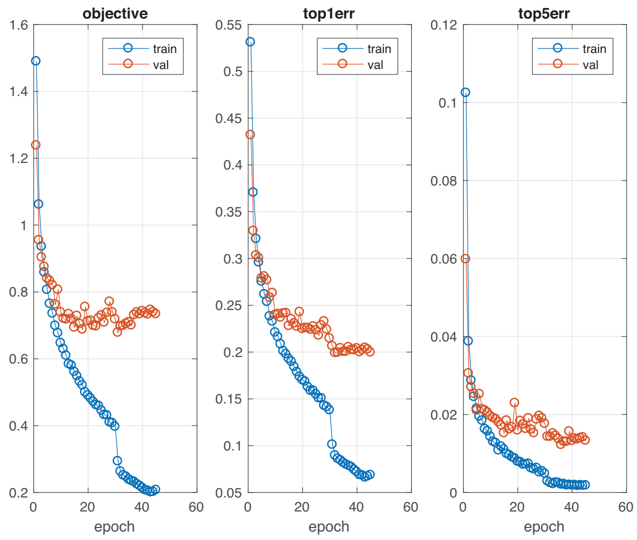


FIGURE 17.16: *This figure shows the results of training the network of Figure 17.15 on the CIFAR-10 training set. Loss, top-1 error and top-5 error for training and validation sets, plotted as a function of epoch for the network of the text. The loss (recorded here as "objective") is the log-loss. Note: the low validation error; the gap between train and validation error; and the very low top-5 error. The validation error is actually quite high for this dataset — you can find a league table at* http://rodrigob.github.io/are_we_there_yet/build/classification_datasets_results.html.

have a stride of 2, so they overlap. The block is padded for this pooling layer, by attaching a single column at the right and a single row at the bottom to get a $33 \times 33 \times 32$ block. With this padding and stride, the pooling takes $33 \times 33 \times 32$ block and produces a $16 \times 16 \times 32$ block (you should check this with a pencil and paper drawing; it's right). The receptive field for values in this block is 7×7 (you should check this with a pencil and paper drawing; it's right, too).

The layer labelled "Apool 3s2" is an average pooling layer which computes an average in a 3×3 window, again with a stride of 2. The block is padded before this layer in the same way the block before the max pooling layer was padded. Eventually, we wind up with a 64 dimensional feature vector describing the image, and the convolutional layer and softmax that follow are logistic regression applied to that feature vector.

Just like MNIST, much of the information in a CIFAR-10 image is redundant. It's now somewhat harder to see the redundancies, but Figure 17.14 should make you suspect that some classes have different backgrounds than others. Figure 17.14 shows the class mean for each class. There are a variety of options for normalizing these images (more below). For this example, I whitened pixel values for each pixel in the image grid independently (procedure 17.1, which is widely used). Whitened images tend to be very hard for humans to interpret. However, the normalization

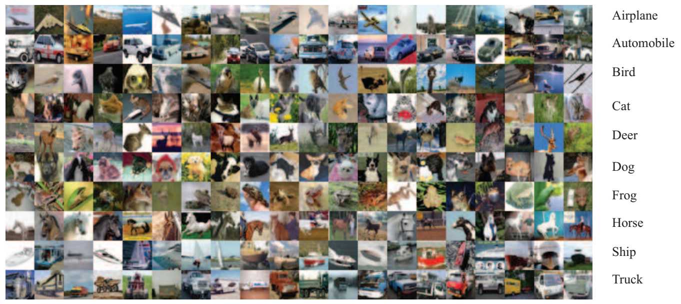


FIGURE 17.17: *Some of the approximately 2000 test examples misclassified by the network trained in the text. Each row corresponds to a category. The images in that row belong to that category, but are classified as belonging to some other category. At least some of these images look like "uncommon" views of the object or "strange" instances – it's plausible that the network misclassifies images when the view is uncommon or the object is a strange instance of the category.*

involved deals with changes in overall image brightness and moderate shifts in color rather well, and can significantly improve classification.

Procedure: 17.1 *Simple image whitening*

At training time: Start with N training images $\mathcal{I}^{(i)}$. We assume these are 3D blocks of data. Write $I^{(i)}_{uvw}$ for the u, v, w'th location in the i'th image. Compute $\mathcal{M}$ and $\mathcal{S}$, where the u, v, w'th location in each is given by

$$M_{uvw} = \frac{\sum_i I^{(i)}_{uvw}}{N}$$

$$S_{uvw} = \sqrt{\frac{\sum_i (I^{(i)}_{uvw} - M^{(i)}_{uvw})^2}{N}}$$

Choose some small number ϵ to avoid dividing by zero. Now the i'th whitened image, $\mathcal{W}^{(i)}$, has for its u, v, w'th location

$$W^{(i)}_{uvw} = (I^{(i)}_{uvw} - M_{uvw})/(S_{uvw} + \epsilon)$$

Use these whitened images to train.
At test time: For a test image $\mathcal{T}$, compute $\mathcal{W}$ which has for its u, v, w'th location

$$W_{uvw} = (T_{uvw} - M_{uvw})/(S_{uvw} + \epsilon)$$

and classify that.

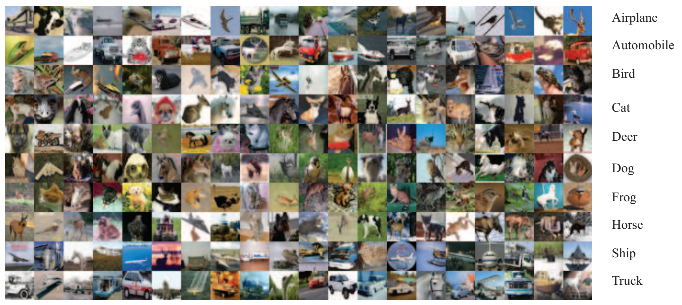


FIGURE 17.18: *Some of the approximately 2000 test examples misclassified by the network trained in the text. Each row corresponds to a category. The images in that row are classified as belonging to that category, but actually belong to another. At least some of these images look like "confusing" views — for example, you can find birds that do look like aircraft, and aircraft that do look like birds.*

I trained this network for 20 epochs using tutorial code circulated with Matconvnet. Mini-batches are pre-selected so that each training data item is touched once per epoch, so an epoch represents a single pass through the data. It is common in image classification to report loss, top-1 error and top-5 error. Top-1 error is the frequency that the correct class has the highest posterior. Top-5 error is the frequency that the correct class appears in the five classes with largest posterior. This can be useful when the top-1 error is large, because you may observe improvements in top-5 error even when the top-1 error doesn't change. Figure 17.16 shows the loss, top-1 error and top-5 error for training and validation sets plotted as a function of epoch. This classifier misclassifies about 2000 of the test examples, so it is hard to show all errors. Figure 17.17 shows examples from each class that are misclassified as belonging to some other class. Figure 17.18 shows examples that are that are misclassified into each class.

The phenomenon that ReLU's are pattern detectors is quite reliable. Figure 17.19 shows the 20 images that give the strongest responses for each of 10 ReLU's in the final ReLU layer. These ReLU's clearly have a quite strong theory of a pattern, and different ReLU's respond most strongly to quite different patterns. More sophisticated visualizations search for images that get the strongest response from units at various stages of complex networks; it's quite reliable that these images show a form of order or structure.

17.2.3 Quirks: Adversarial Examples

Adversarial examples are a curious experimental property of neural network image classifiers. Here is what happens. Assume you have an image $\mathbf{x}$ that is correctly classified with label l. The network will produce a probability distribution over labels $P(L|\mathbf{x})$. Choose some label k that is not correct. It is possible to use modern

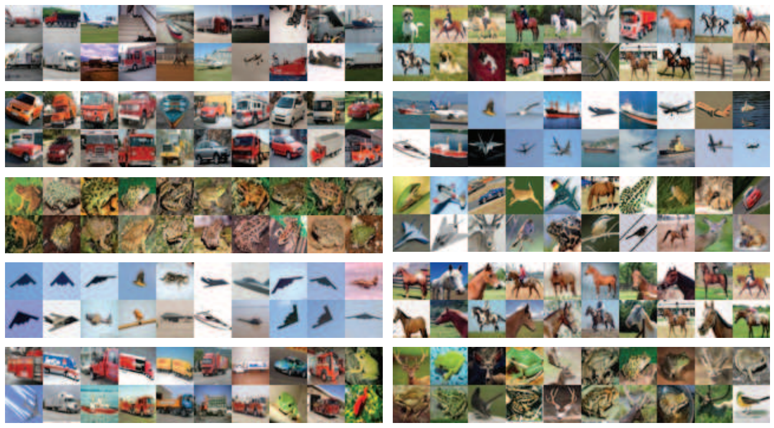


FIGURE 17.19: *Visualizing the patterns that the final stage ReLU's respond to for the simple CIFAR example. Each block of images shows the images that get the largest output for each of 10 ReLU's (the ReLU's were chosen at random from the 64 available in the top ReLU layer). Notice that these ReLU outputs don't correspond to class – these outputs go through a fully connected layer before classification – but each ReLU are clearly responds to a pattern, and different ReLU's respond more strongly to different patterns.*

optimization methods to search for a modification to the image $\delta\mathbf{x}$ such that

$$\begin{array}{rc} \delta\mathbf{x} & \text{is small} \\ & \textit{and} \\ P(k|\mathbf{x}+\delta\mathbf{x}) & \text{is large.} \end{array}$$

You might expect that $\delta\mathbf{x}$ is "large"; what is surprising is that mostly it is so tiny as to be imperceptible to a human observer. The property of being an adversarial example seems to be robust to image smoothing, simple image processing, and printing and photographing. The existence of adversarial examples raises the following, rather alarming, prospect: You could make a template that you could hold over a stop sign, and with one pass of a spraypaint can, turn that sign into something that is interpreted as a minimum speed limit sign by current computer vision systems. I haven't seen this demonstration done yet, but it appears to be entirely within the reach of modern technology, and it and activities like it offer significant prospects for mayhem.

What is startling about this behavior is that it is exhibited by networks that are very good at image classification, *assuming* that no-one has been fiddling with the images. So modern networks are very accurate on untampered pictures, but may behave very strangely in the presence of tampering. One can (rather vaguely) identify the source of the problem, which is that neural network image classifiers have far more degrees of freedom than can be pinned down by images. This observation doesn't really help, though, because it doesn't explain why they (mostly) work rather well, and it doesn't tell us what to do about adversarial examples.

There have been a variety of efforts to produce networks that are robust to adversarial examples, but evidence right now is based only on experiment (some networks behave better than others) and we are missing clear theoretical guidance.

17.3 YOU SHOULD

17.3.1 remember these definitions:

17.3.2 remember these terms:

image classification	414
object detection	414
mask	415
kernel	415
convolution	416
stride	417
padding	418
feature maps	418
blocks	418
kernel block	419
convolutional layer	419
features	421
1×1 convolution	422
receptive field	422
pooling	423
max pooling	423
average pooling	423

17.3.3 remember these facts:

Definition: Convolutional Layer	420
Making fully connected layers with convolutional layers	420
Convolutional layer + ReLU=Pattern detector	421
There are two common meanings of “convolutional layer”	421
Pattern detector responses are usually sparse	422
1×1 convolution=linear map	422

17.3.4 remember these procedures:

Simple image whitening	431

17.3.5 be able to:

- Explain what convolutional layers do.
- Compute the size of a data block resulting from applying a convolutional layer with given size and stride to a block with given padding.
- Explain what a 1×1 convolution does and why it might be useful.
- Train and run a simple image classifier in your chosen framework.
- Explain why pre-processing data might help a neural network based classifier.
- Explain what an adversarial example is.

PROGRAMMING EXERCISES

17.1. Download tutorial code for a simple MNIST classifier for your chosen programming framework, and train and run a classifier using that code. You should be able to do this exercise without access to a GPU.

17.2. Now reproduce the example of Section 17.2.1 in your chosen programming framework. The section contains enough detail about the structure of the network for you to build that network. This isn't a super good classifier; the point of the exercise is being able to translate a description of a network to an instance. Use the standard test-train split, and train with straightforward stochastic gradient descent. Choose a minibatch size that works for this example and your hardware. Again, you should be able to do this exercise without access to a GPU.

- **(a)** Does using momentum improve training?
- **(b)** Does using dropout in the first two layers result in a better performing network?
- **(c)** Modify this network architecture to improve performance. Reading ahead will suggest some tricks. What works best?

17.3. Download tutorial code for a simple CIFAR-10 classifier for your chosen programming framework, and train and run a classifier using that code. You might very well be able to do this exercise without access to a GPU.

17.4. Now reproduce the example of Section 17.2.2 in your chosen programming framework. The section contains enough detail about the structure of the network for you to build that network. This isn't a super good classifier; the point of the exercise is being able to translate a description of a network to an instance. Use the standard test-train split, and train with straightforward stochastic gradient descent. Choose a minibatch size that works for this example and your hardware. Again, you might very well be able to do this exercise without access to a GPU.

- **(a)** Does using momentum improve training?
- **(b)** Does using dropout in the first two layers result in a better performing network?
- **(c)** Modify this network architecture to improve performance. Reading ahead will suggest some tricks. What works best?