Image classification

Big picture

- We know how to do this
  - with convnets
    - minor fights about architecture, but…
- You should think about these nets as
  - complicated feature construction followed by simple classifier
- Dynamics of learning are not even slightly understood
- There are some nasty ifs, ands and buts

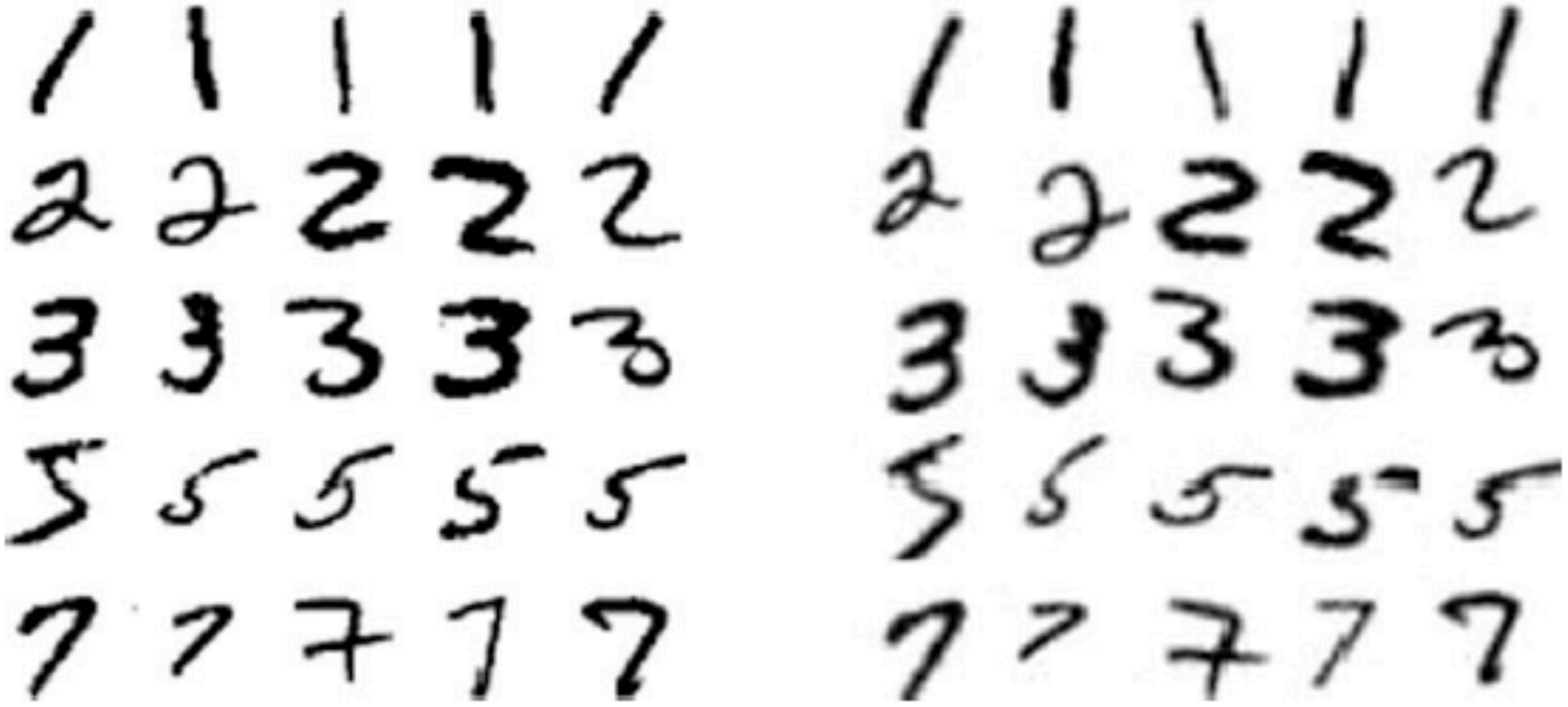


Figure 17.1: On the **left**, a selection of digits from the MNIST dataset. Notice how images of the same digit can vary, which makes classifying the image demanding. It is quite usual that pictures of "the same thing" look quite different. On the **right**, digit images from MNIST that have been somewhat 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

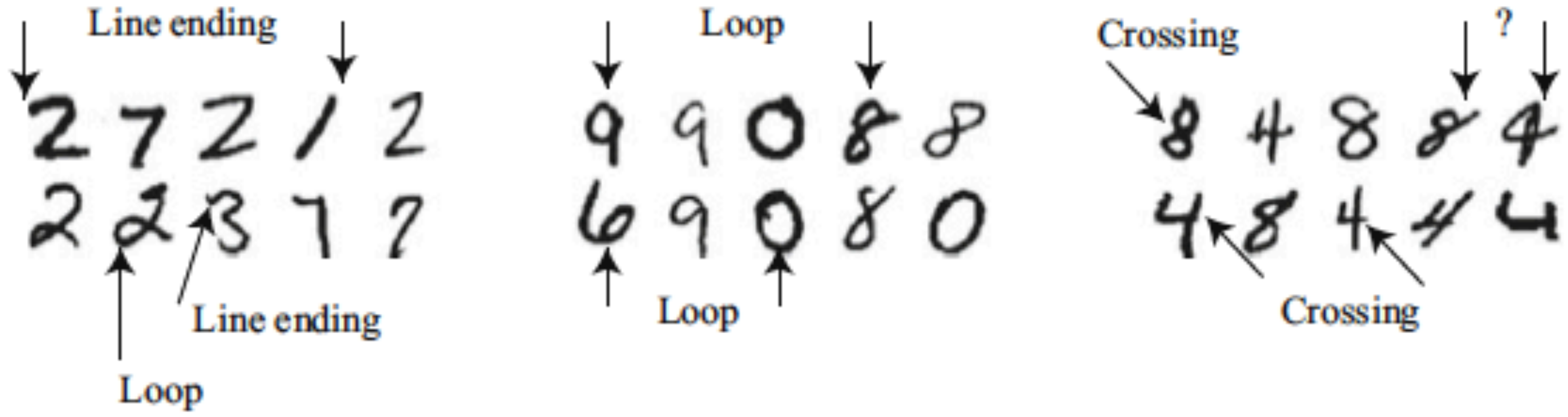


Figure 17.2: Local patterns in images are quite informative. MNIST images, shown here, are simple images, so a small set of patterns is quite helpful. The relative location of patterns is also informative. So, for example, an eight has two loops, one above the other. All this suggests a key strategy: 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

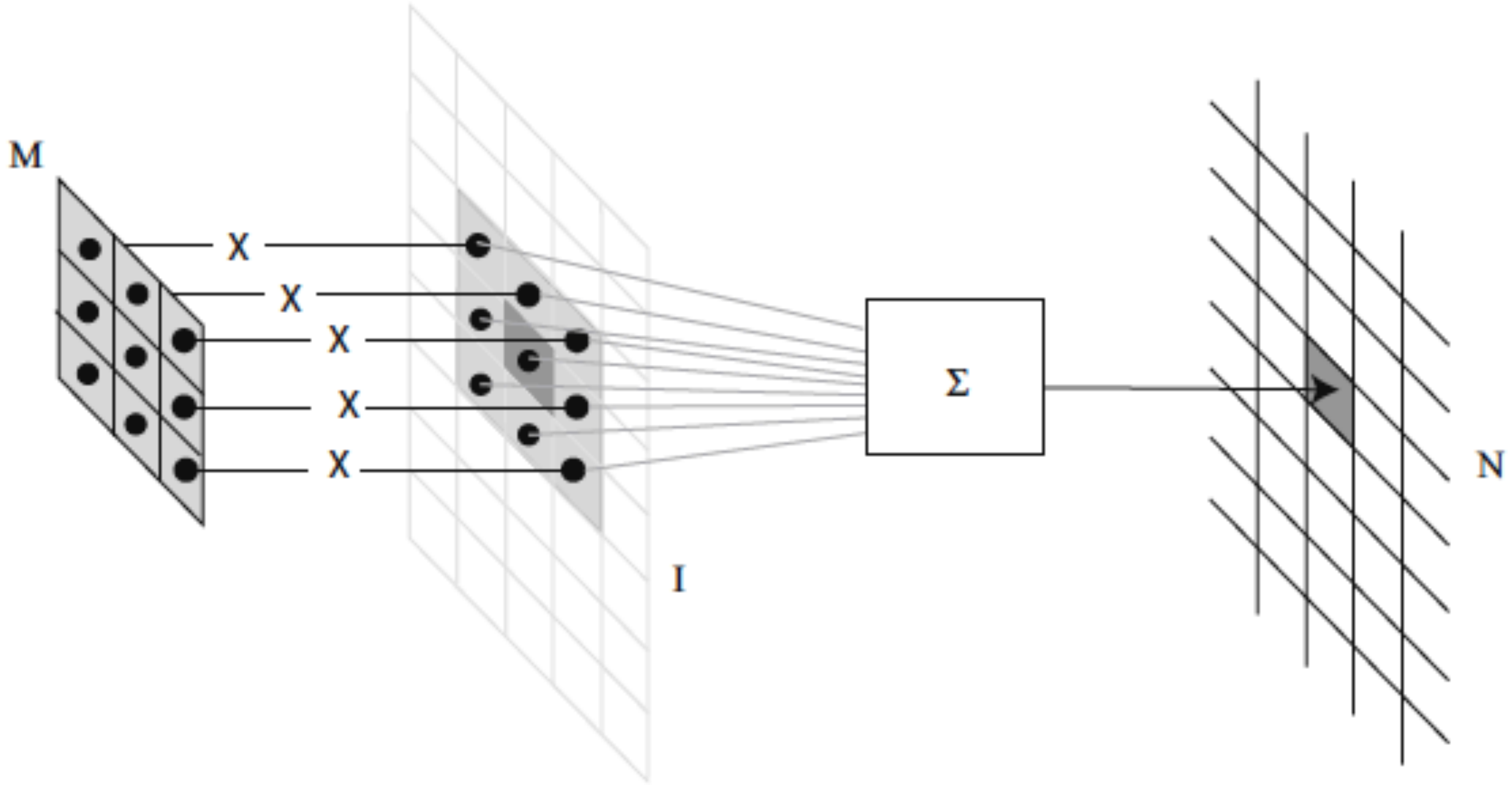
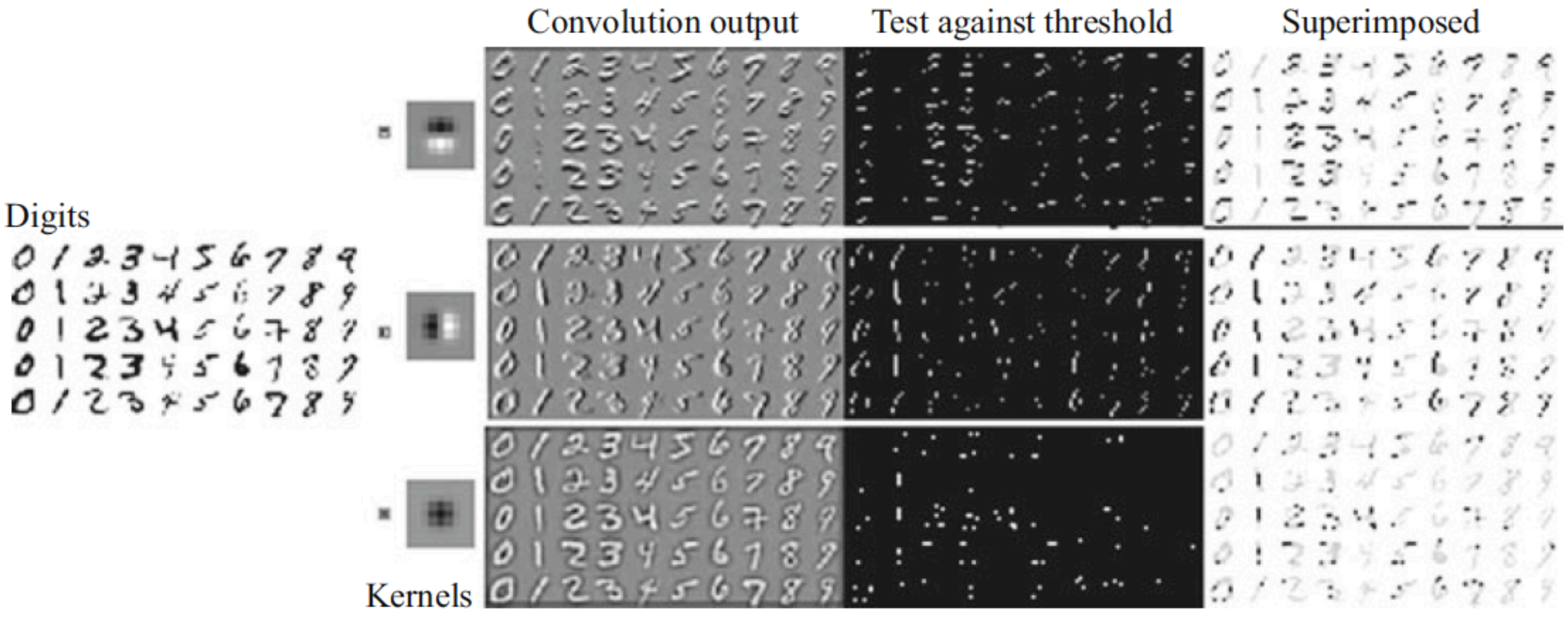


Figure 17.4: To compute the value of $\mathcal{N}$ at some location, you shift a copy of $\mathcal{M}$ to lie over that location in $\mathcal{I}$; you multiply together the non-zero elements of $\mathcal{M}$ and $\mathcal{I}$ that lie on top of one another; and you sum the results


Convolution output
Test against threshold
Superimposed
Digits
Kernels

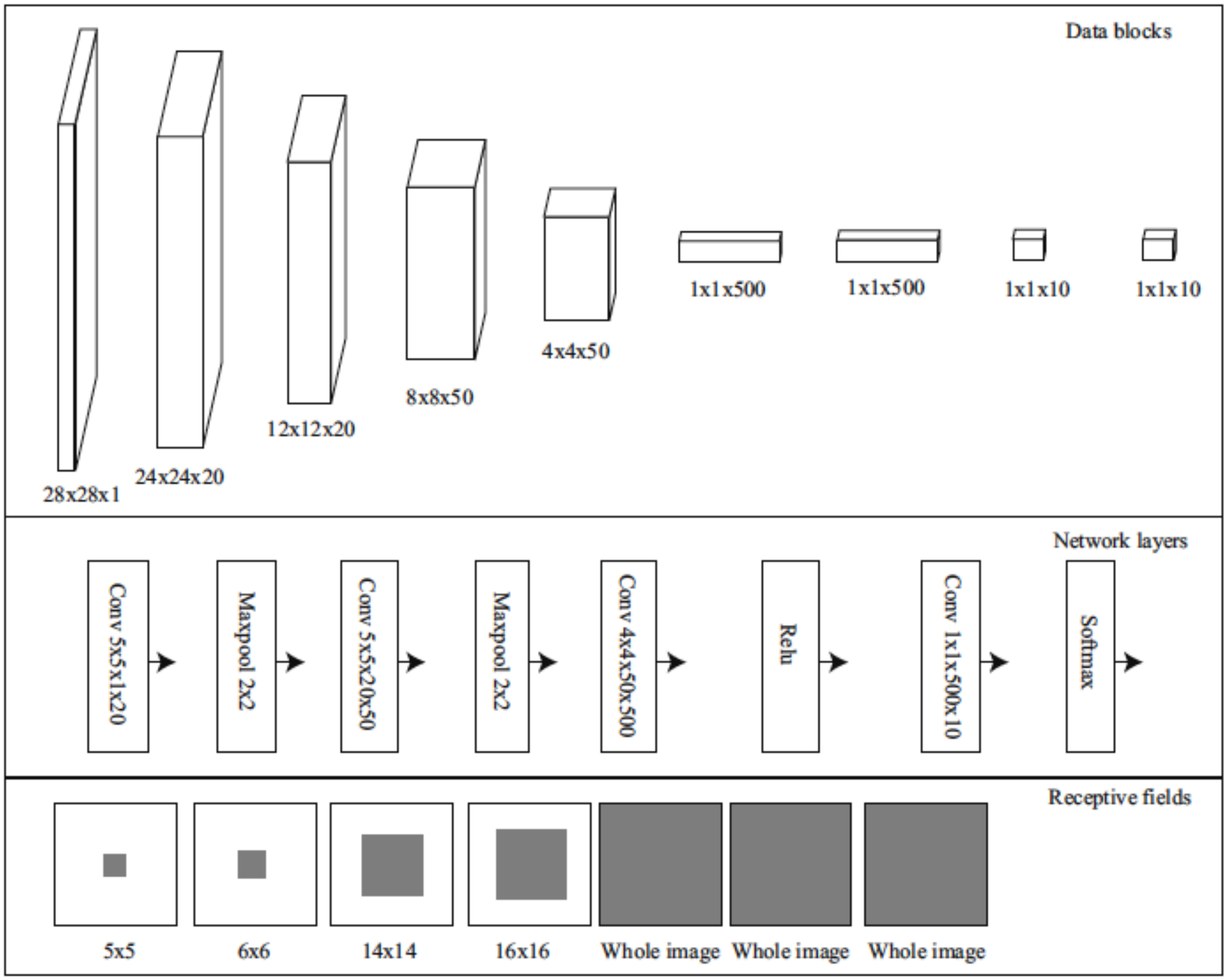


Figure 17.9: Three different representations of the simple network used to classify MNIST digits for this example. Details in the text

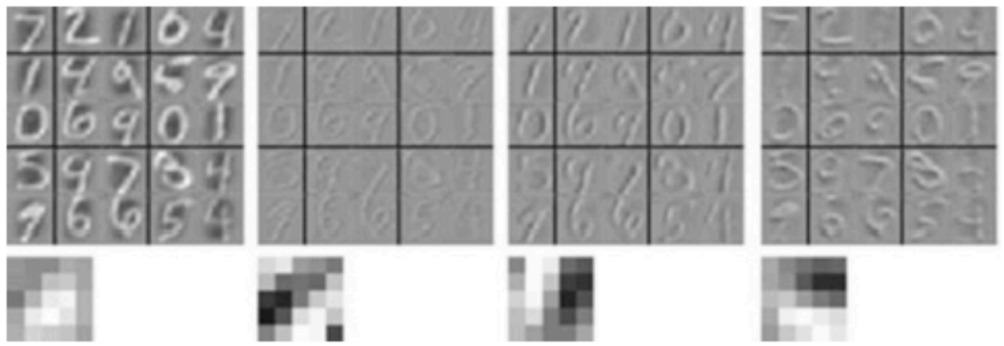


Figure 17.10: Four of the 20 kernels in the first layer of my trained version of the MNIST network. The kernels are small (5×5) and have been blown up so you can see them. The outputs for each kernel on a set of images are shown above the kernel. The output images are scaled so that the largest value 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

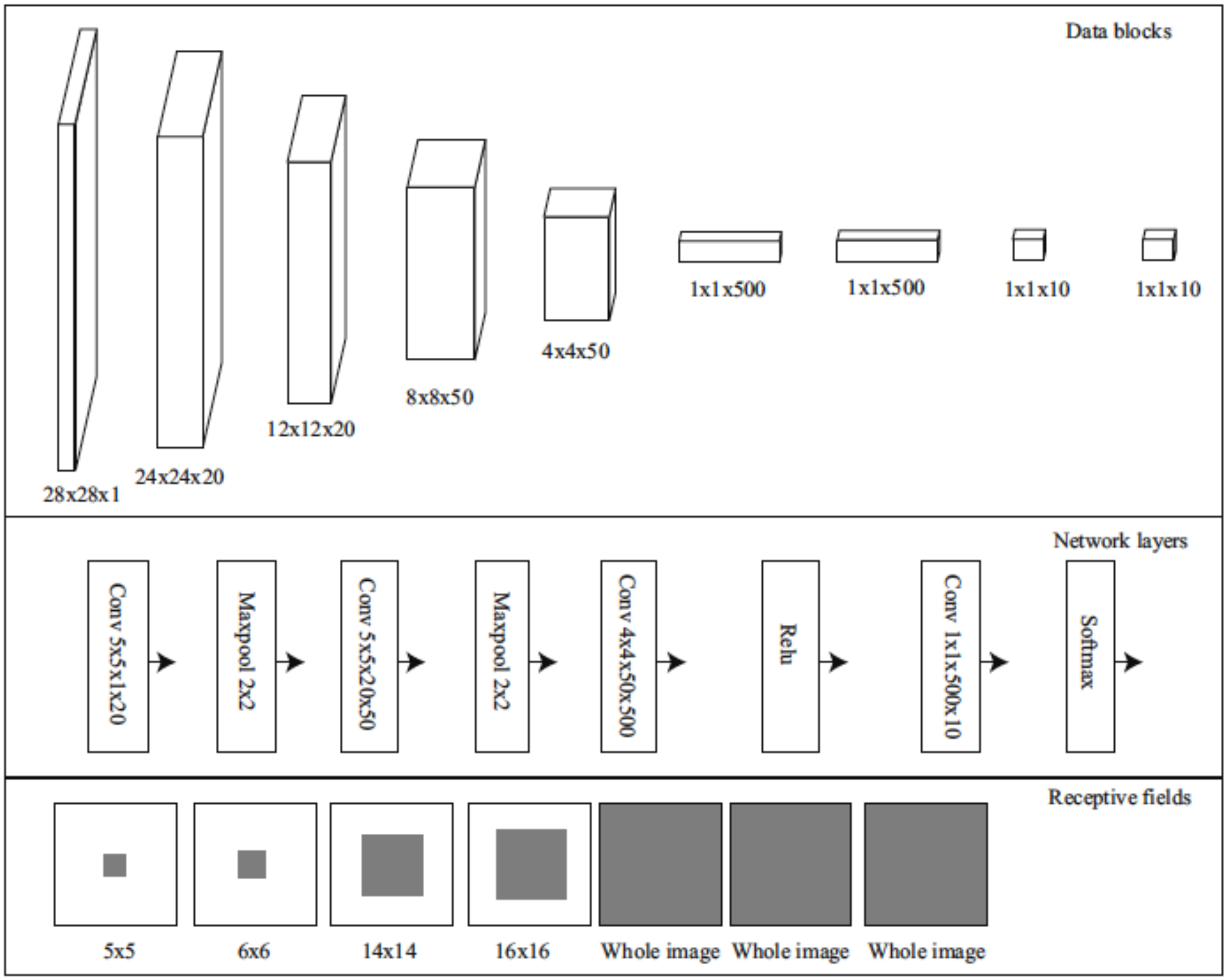


Figure 17.9: Three different representations of the simple network used to classify MNIST digits for this example. Details in the text

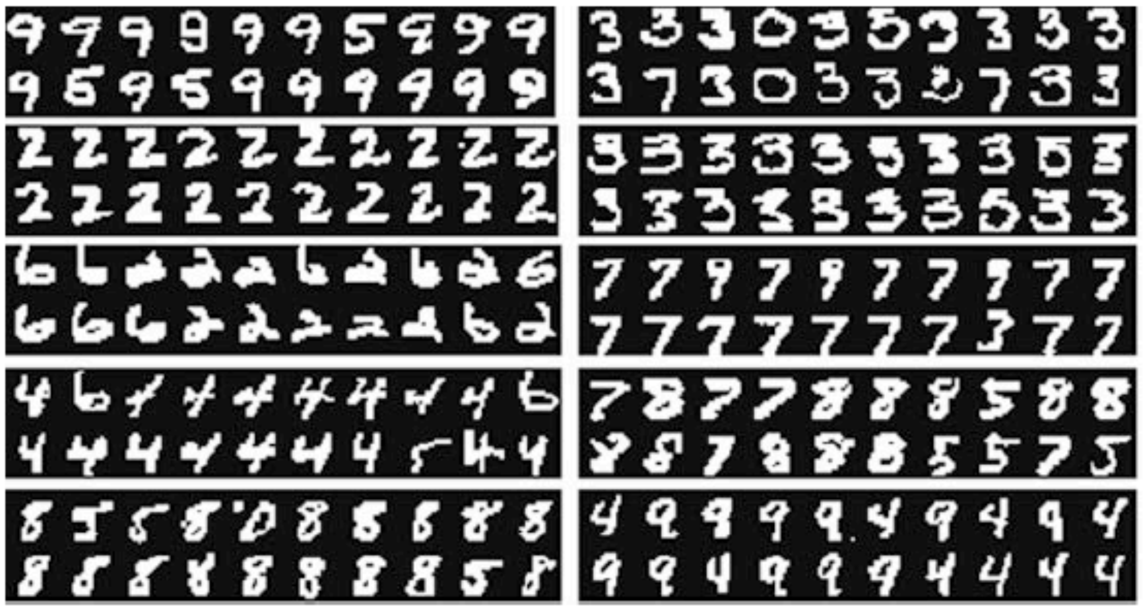
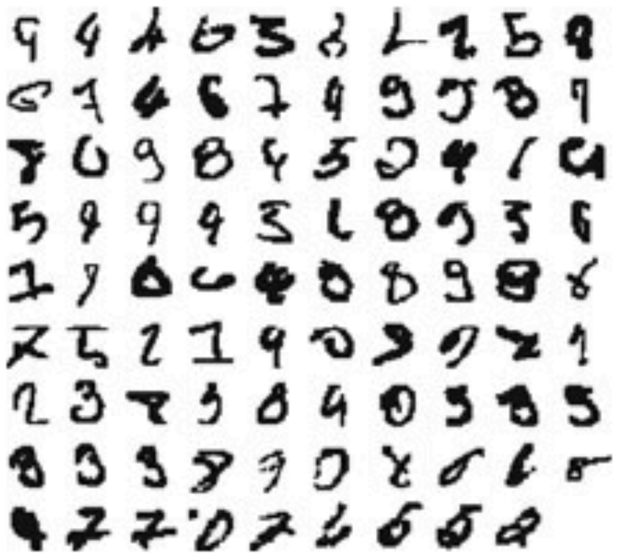


Figure 17.11: Visualizing the patterns that the final stage ReLUs respond to for the simple CIFAR example. Each block of images shows the images that get the largest output for each of 10 ReLUs (the ReLUs were chosen at random from the 500 available). Notice that these ReLU outputs don't correspond to class—these outputs go through a fully connected layer before classification—but each ReLU clearly responds to a pattern, and different ReLUs respond more strongly to different patterns



```
| 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 10,000 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

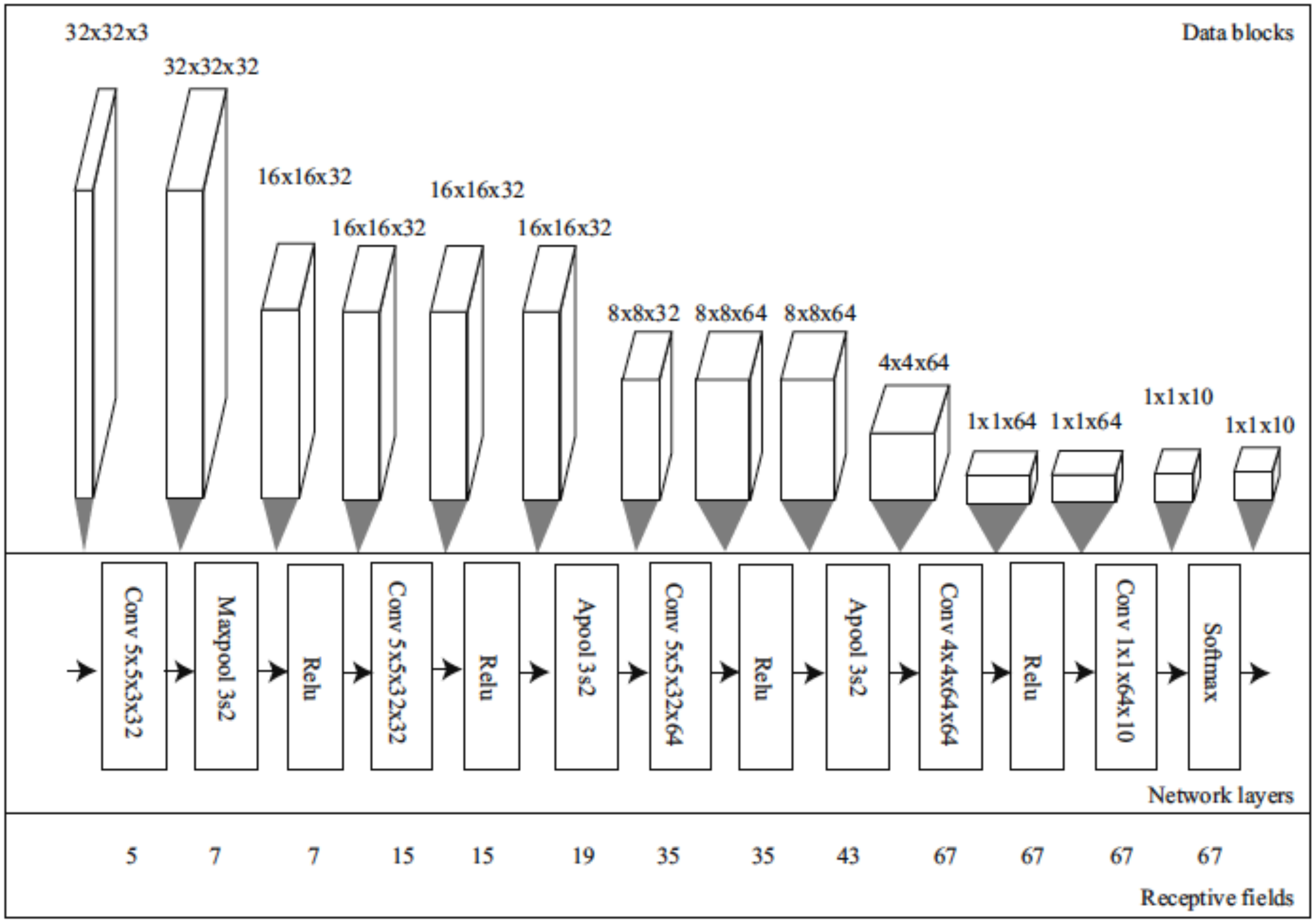


Figure 17.15: Three different representations of the simple network used to classify CIFAR-10 images for this example. Details in the text

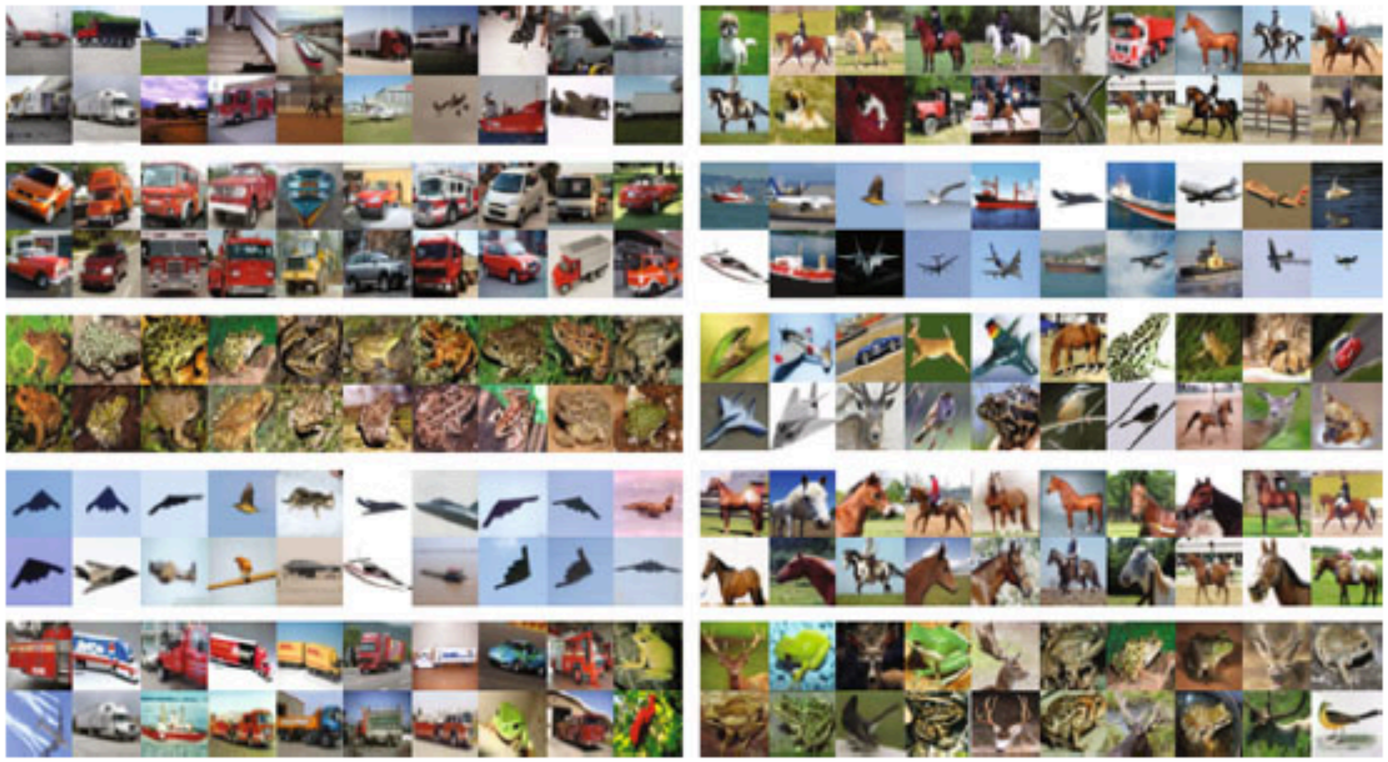


Figure 17.19: Visualizing the patterns that the final stage ReLUs respond to for the simple CIFAR example. Each block of images shows the images that get the largest output for each of 10 ReLUs (the ReLUs were chosen at random from the 64 available in the top ReLU layer). Notice that these ReLU outputs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

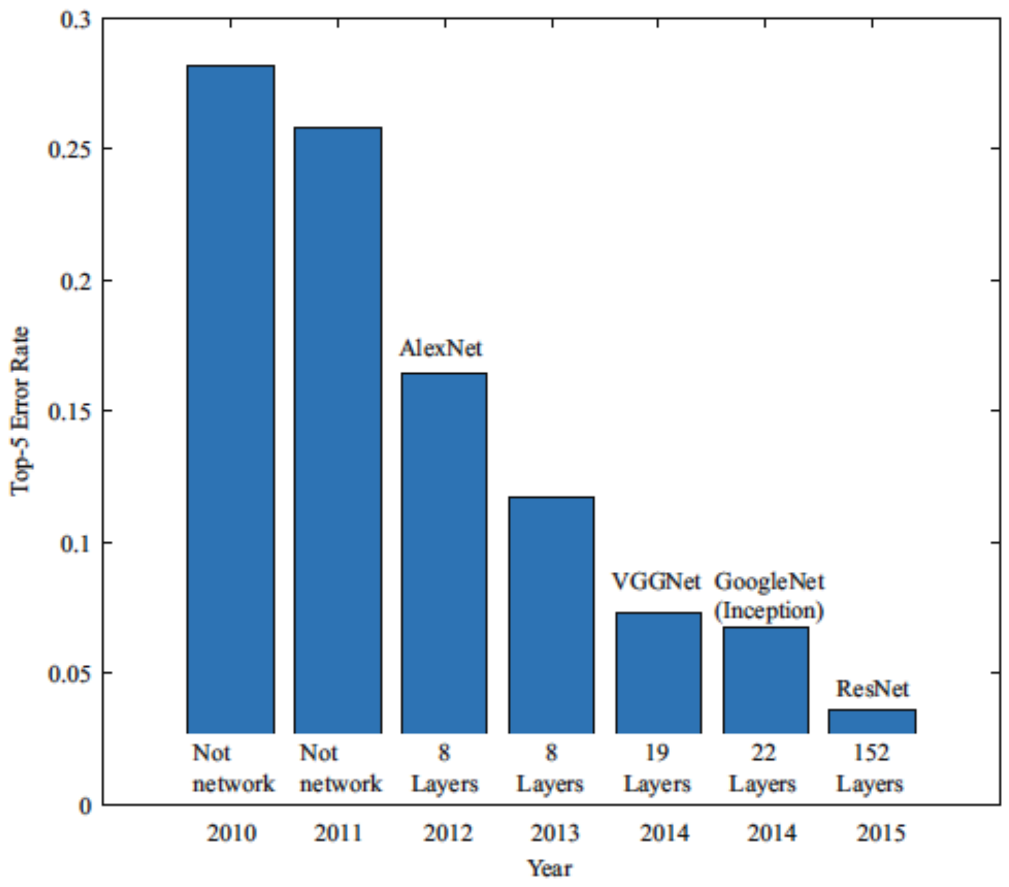


Figure 18.1: The top-5 error rate for image classification using the ImageNet dataset has collapsed from 28% to 3.6% from 2010 to 2015. There are two 2014 entries here, which makes the fall in error rate look slower. This is because each of these methods is significant, and discussed in the sections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

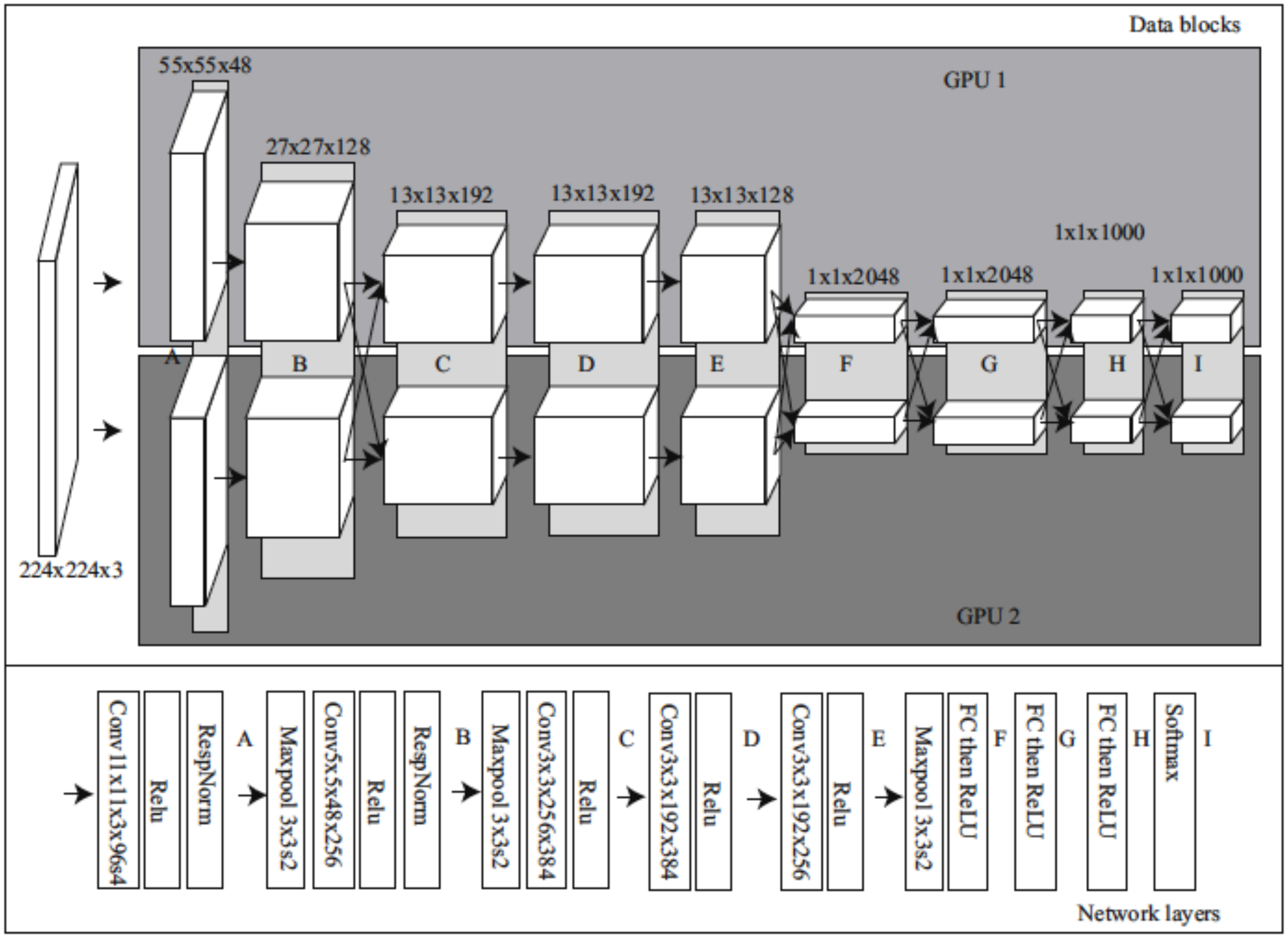


Figure 18.2: Two views of the architecture of AlexNet, the first convolutional neural network architecture to beat earlier feature constructions at image classification. There are five convolutional layers with ReLU, response normalization, and pooling layers interspersed. **Top** shows the data blocks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

Network architecture				
A	B	C	D	E
Number of layers with learnable weights				
11	13	16	16	19
Input (224 × 224 × 3 image)				
conv3-64	conv3-64	conv3-64	conv3-64	conv3-64
	conv3-64	conv3-64	conv3-64	conv3-64
maxpool2 × 2s2				
conv3-64	conv3-64	conv3-64	conv3-64	conv3-64
	conv3-64	conv3-64	conv3-64	conv3-64
maxpool2 × 2s2				
conv3-128	conv3-128	conv3-128	conv3-128	conv3-128
	conv3-128	conv3-128	conv3-128	conv3-128
maxpool2 × 2s2				
conv3-256	conv3-256	conv3-256	conv3-256	conv3-256
conv3-256	conv3-256	conv3-256	conv3-256	conv3-256
		conv1-256	**conv3-256**	conv3-256
				conv3-256
maxpool2 × 2s2				
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
		conv1-512	**conv3-512**	conv3-512
				conv3-512
maxpool2 × 2s2				
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
conv3-512	conv3-512	conv3-512	conv3-512	conv3-512
		conv1-512	**conv3-512**	conv3-512
				conv3-512
maxpool2 × 2s2				
FC-4096				
FC-4096				
FC-1000				
softmax				

Batch normalization

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

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

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

You can normalize in other ways…

- Instance, group, etc.
- See blog post on web page

Inception

- Modules
- Normalization
- 1x1 Convolution
  - You should think of this as a dimension reduction process

Inception

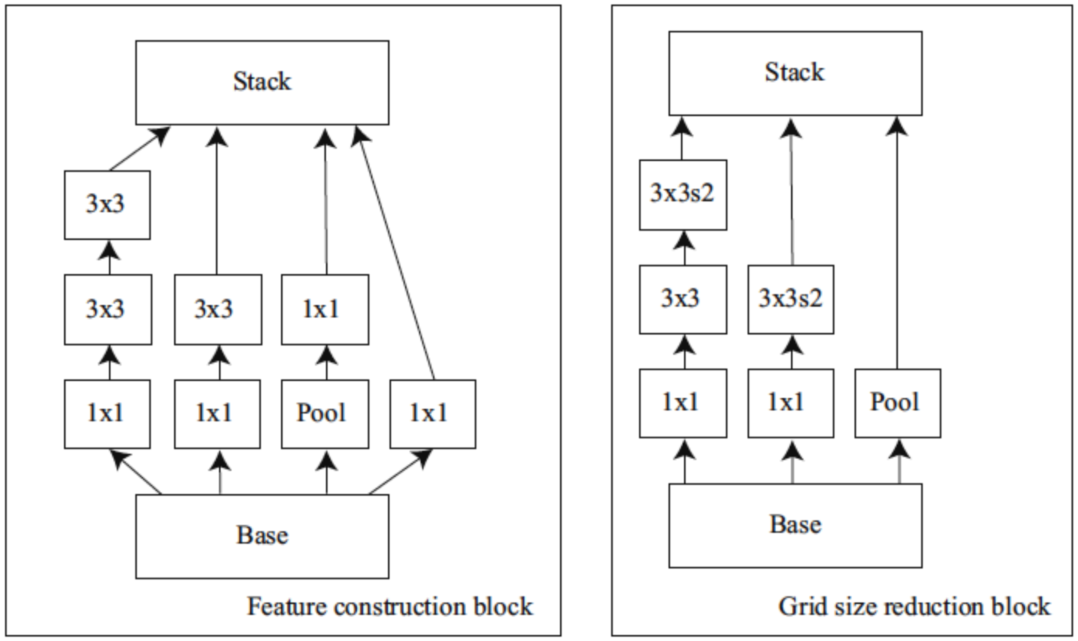


Figure 18.4: On the **left** an inception module for computing features. On the **right**, a module that reduces the size of the grid. The feature module features with: 5×5 support (far left stream); 3×3 support (left stream); 1×1 support after pooling (right stream); and 1×1 support without pooling.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

Residual networks

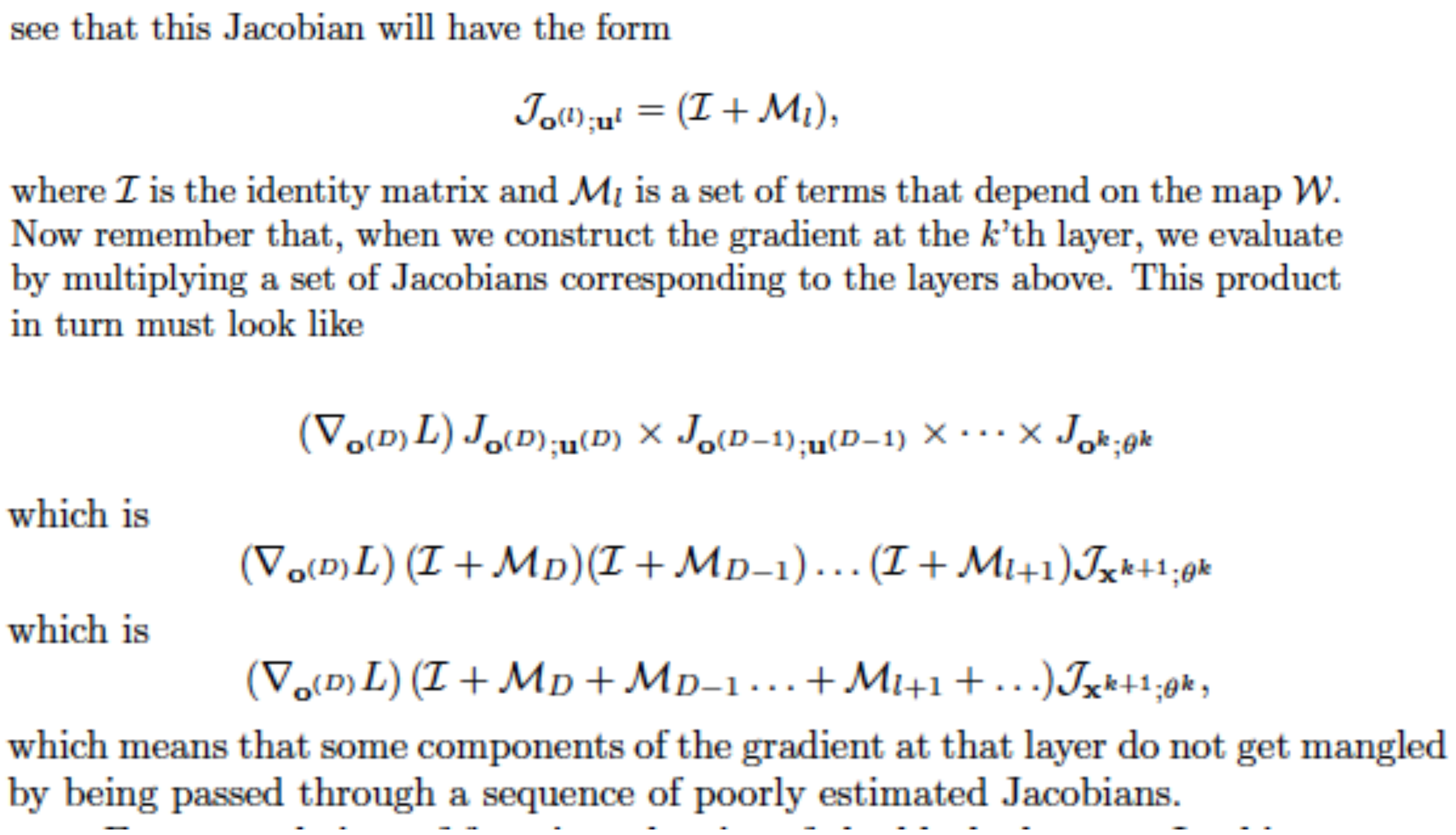
Our usual process takes a data block $\mathcal{X}^{(l)}$, forms a function of that block $\mathcal{W}(\mathcal{X}^{(l)})$, then applies a ReLU to the result. To date, the function involves applying either a fully connected layer or a convolution, then adding bias terms. Writing $F(\cdot)$ for a ReLU,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

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

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

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

Residual networks



see that this Jacobian will have the form

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

where $\mathcal{I}$ is the identity matrix and $\mathcal{M}_l$ is a set of terms that depend on the map $\mathcal{W}$. Now remember that, when we construct the gradient at the k'th layer, we evaluate by multiplying a set of Jacobians corresponding to the layers above. This product in turn must look like

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

which is

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

which is

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

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

If’s ands and buts

- Adversarial examples
  - AAAARGH!
- Weird correlation properties between networks
  - mostly all make the same errors, often in disturbing detail
- What do you do if you want to:
  - improve accuracy
  - change representation
  - talk sense about generalization
  - classify with very little data