CHAPTER 5

Convolutional Neural Networks

Neurons in Human Vision

The human sense of vision is unbelievably advanced. Within fractions of seconds, we can identify objects within our field of view, without thought or hesitation. Not only can we name objects we are looking at, we can also perceive their depth, perfectly distinguish their contours, and separate the objects from their backgrounds. Somehow our eyes take in raw voxels of color data, but our brain transforms that information into more meaningful primitives—lines, curves, and shapes—that might indicate, for example, that we're looking at a house cat.[1]

Foundational to the human sense of vision is the neuron. Specialized neurons are responsible for capturing light information in the human eye.[2] This light information is then preprocessed, transported to the visual cortex of the brain, and then finally analyzed to completion. Neurons are single-handedly responsible for all of these functions. As a result, intuitively, it would make a lot of sense to extend our neural network models to build better computer vision systems. In this chapter, we will use our understanding of human vision to build effective deep learning models for image problems. But before we jump in, let's take a look at more traditional approaches to image analysis and why they fall short.

1 Hubel, David H., and Torsten N. Wiesel. "Receptive fields and functional architecture of monkey striate cortex." *The Journal of Physiology* 195.1 (1968): 215-243.

2 Cohen, Adolph I. "Rods and Cones." *Physiology of Photoreceptor Organs*. Springer Berlin Heidelberg, 1972. 63-110.

The Shortcomings of Feature Selection

Let's begin by considering a simple computer vision problem. I give you a randomly selected image, such as the one in Figure 5-1. Your task is to tell me if there is a human face in this picture. This is exactly the problem that Paul Viola and Michael Jones tackled in their seminal paper published in 2001.[3]

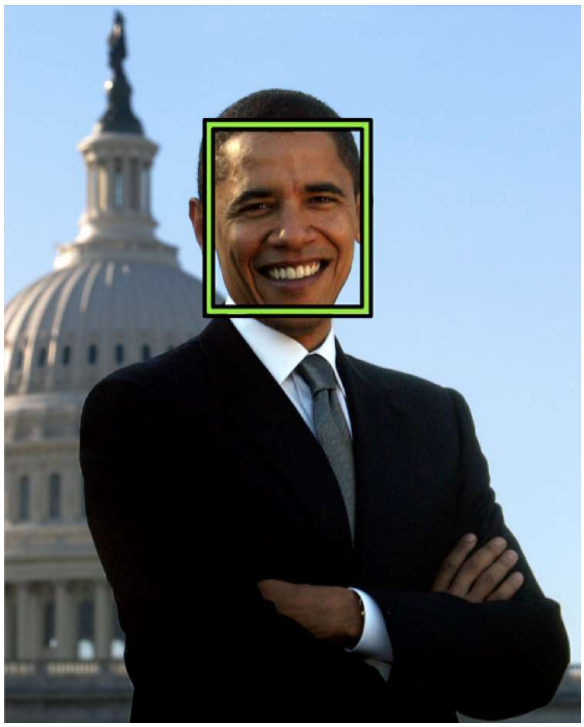

Figure 5-1. A hypothetical face-recognition algorithm should detect a face in this photograph of former President Barack Obama

For a human like you or me, this task is completely trivial. For a computer, however, this is a very difficult problem. How do we teach a computer that an image contains a face? We could try to train a traditional machine learning algorithm (like the one we described in the Chapter 1) by giving it the raw pixel values of the image and hoping it can find an appropriate classifier. Turns out this doesn't work very well at all because the signal-to-noise ratio is much too low for any useful learning to occur. We need an alternative.

The compromise that was eventually reached was essentially a trade-off between the traditional computer program, where the human defined all of the logic, and a pure

3 Viola, Paul, and Michael Jones. "Rapid Object Detection using a Boosted Cascade of Simple Features." Computer Vision and Pattern Recognition, 2001. CVPR 2001. *Proceedings of the 2001 IEEE Computer Society Conference* on. Vol. 1. IEEE, 2001.

machine learning approach, where the computer did all of the heavy lifting. In this compromise, a human would choose the features (perhaps hundreds or thousands) that he or she believed were important in making a classification decision. In doing so, the human would be producing a lower-dimensional representation of the same learning problem. The machine learning algorithm would then use these new *feature vectors* to make classification decisions. Because the *feature extraction* process improves the signal-to-noise ratio (assuming the appropriate features are picked), this approach had quite a bit of success compared to the state of the art at the time.

Viola and Jones had the insight that faces had certain patterns of light and dark patches that they could exploit. For example, there is a difference in light intensity between the eye region and the upper cheeks. There is also a difference in light intensity between the nose bridge and the two eyes on either side. These detectors are shown in Figure 5-2.

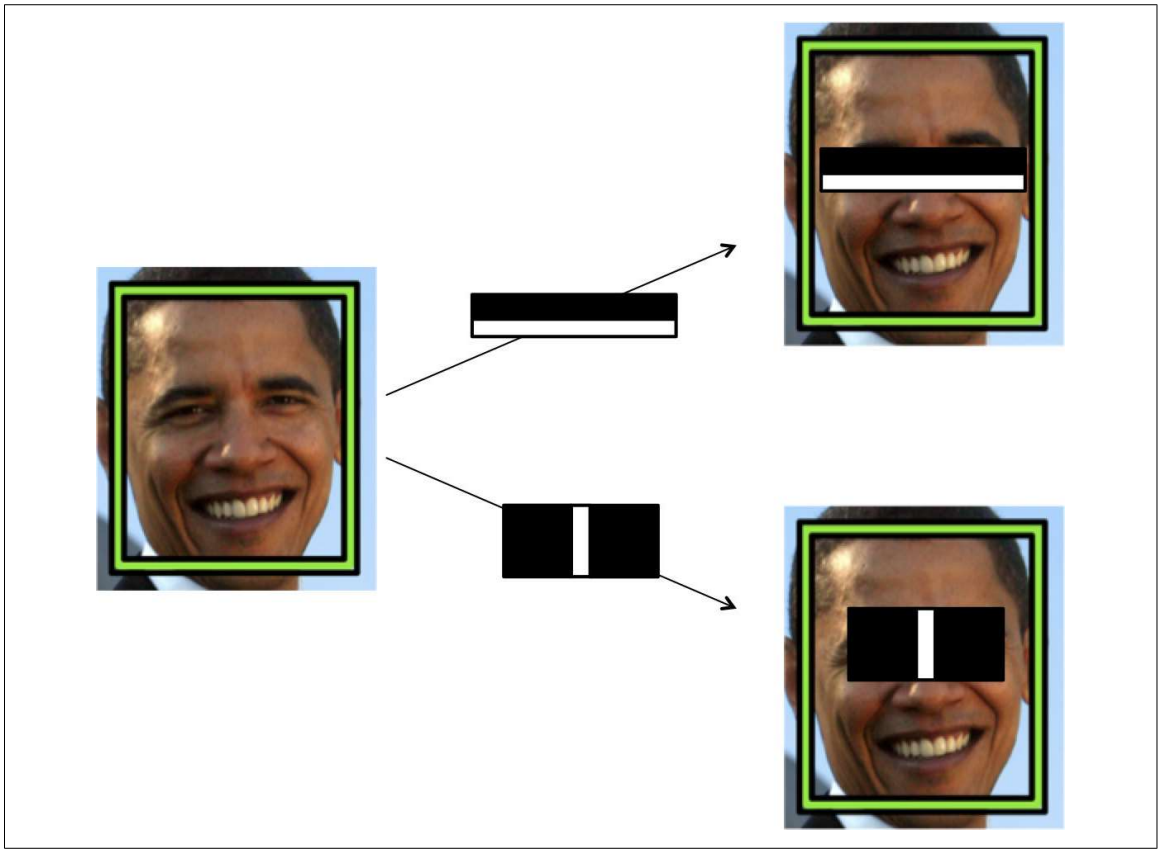

Figure 5-2. An illustration of Viola-Jones intensity detectors

By themselves, each of these features is not very effective at identifying a face. But when used together (through a classic machine learning algorithm known as boosting, described in the original manuscript), their combined effectiveness drastically increases. On a dataset of 130 images and 507 faces, the algorithm achieves a 91.4% detection rate with 50 false positives. The performance was unparalleled at the time,

but there are fundamental limitations of the algorithm. If a face is partially covered with shade, the light intensity comparisons no longer work. Moreover, if the algorithm is looking at a face on a crumpled flier or the face of a cartoon character, it would most likely fail.

The problem is the algorithm hasn't really learned that much about what it means to "see" a face. Beyond differences in light intensity, our brain uses a vast number of visual cues to realize that our field of view contains a human face, including contours, relative positioning of facial features, and color. And even if there are slight discrepancies in one of our visual cues (for example, if parts of the face are blocked from view or if shade modifies light intensities), our visual cortex can still reliably identify faces.

In order to use traditional machine learning techniques to teach a computer to "see," we need to provide our program with a lot more features to make accurate decisions. Before the advent of deep learning, huge teams of computer vision researchers would take years to debate about the usefulness of different features. As the recognition problems became more and more intricate, researchers had a difficult time coping with the increase in complexity.

To illustrate the power of deep learning, consider the ImageNet challenge, one of the most prestigious benchmarks in computer vision (sometimes even referred to as the Olympics of computer vision).[4] Every year, researchers attempt to classify images into one of 200 possible classes given a training dataset of approximately 450,000 images. The algorithm is given five guesses to get the right answer before it moves onto the next image in the test dataset. The goal of the competition is to push the state of the art in computer vision to rival the accuracy of human vision itself (approximately 95–96%). In 2011, the winner of the ImageNet benchmark had an error rate of 25.7%, making a mistake on one out of every four images.[5] Definitely a huge improvement over random guessing, but not good enough for any sort of commercial application. Then in 2012, Alex Krizhevsky from Geoffrey Hinton's lab at the University of Toronto did the unthinkable. Pioneering a deep learning architecture known as a *convolutional neural network* for the first time on a challenge of this size and complexity, he blew the competition out of the water. The runner up in the competition scored a commendable 26.1% error rate. But AlexNet, over the course of just a few months of work, completely crushed 50 years of traditional computer vision research with an

4 Deng, Jia, et al. "ImageNet: A Large-Scale Hierarchical Image Database." *Computer Vision and Pattern Recognition*, 2009. CVPR 2009. IEEE Conference. IEEE, 2009.

5 Perronnin, Florent, Jorge Sénchez, and Yan Liu Xerox. "Large-scale image categorization with explicit data embedding." *Computer Vision and Pattern Recognition* (CVPR), 2010 IEEE Conference. IEEE, 2010.

error rate of approximately 16%.[6] It would be no understatement to say that AlexNet single-handedly put deep learning on the map for computer vision, and completely revolutionized the field.

Vanilla Deep Neural Networks Don't Scale

The fundamental goal in applying deep learning to computer vision is to remove the cumbersome, and ultimately limiting, feature selection process. As we discussed in Chapter 1, deep neural networks are perfect for this process because each layer of a neural network is responsible for learning and building up features to represent the input data that it receives. A naive approach might be for us to use a vanilla deep neural network using the network layer primitive we designed in Chapter 3 for the MNIST dataset to achieve the image classification task.

If we attempt to tackle the image classification problem in this way, however, we'll quickly face a pretty daunting challenge, visually demonstrated in Figure 5-3. In MNIST, our images were only 28 x 28 pixels and were black and white. As a result, a neuron in a fully connected hidden layer would have 784 incoming weights. This seems pretty tractable for the MNIST task, and our vanilla neural net performed quite well. This technique, however, does not scale well as our images grow larger. For example, for a full-color 200 x 200 pixel image, our input layer would have 200 x 200 x 3 = 120,000 weights. And we're going to want to have lots of these neurons over multiple layers, so these parameters add up quite quickly! Clearly, this full connectivity is not only wasteful, but also means that we're much more likely to overfit to the training dataset.

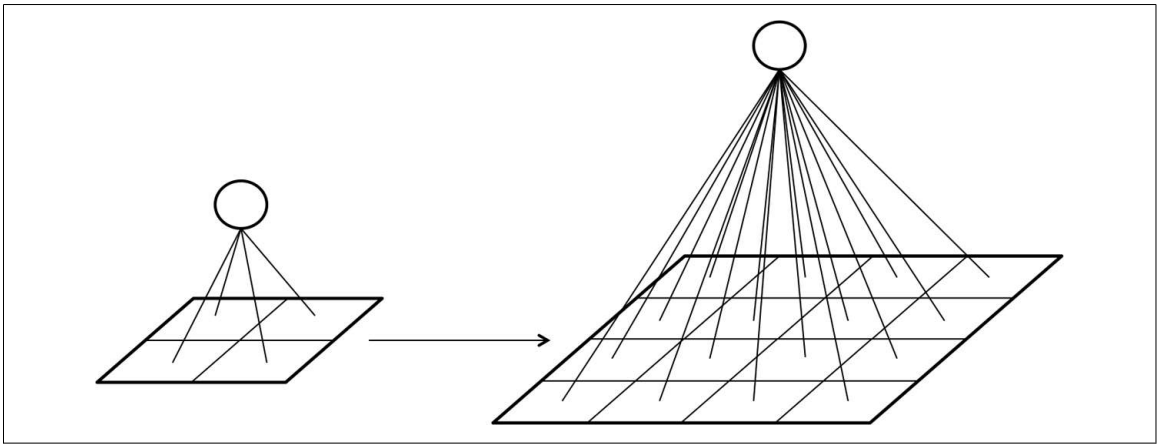

Figure 5-3. The density of connections between layers increases intractably as the size of the image increases

6 Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*. 2012.

The convolutional network takes advantage of the fact that we're analyzing images, and sensibly constrains the architecture of the deep network so that we drastically reduce the number of parameters in our model. Inspired by how human vision works, layers of a convolutional network have neurons arranged in three dimensions, so layers have a width, height, and depth, as shown in Figure 5-4.[7] As we'll see, the neurons in a convolutional layer are only connected to a small, local region of the preceding layer, so we avoid the wastefulness of fully-connected neurons. A convolutional layer's function can be expressed simply: it processes a three-dimensional volume of information to produce a new three-dimensional volume of information. We'll take a closer look at how this works in the next section.

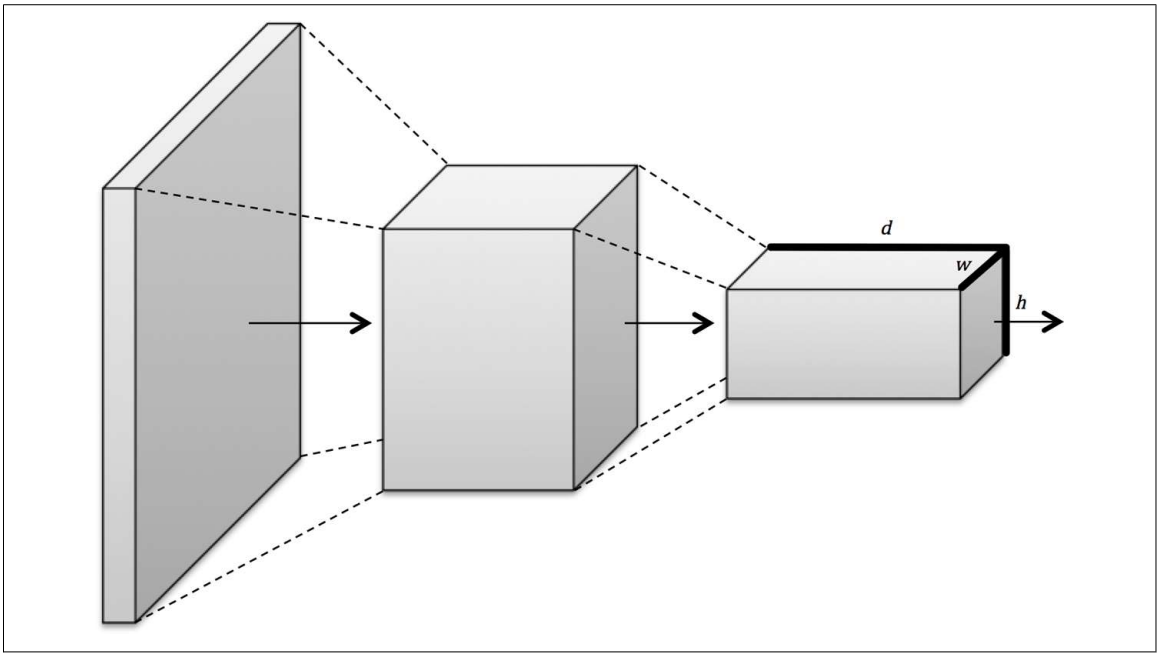

Figure 5-4. Convolutional layers arrange neurons in three dimensions, so layers have width, height, and depth

Filters and Feature Maps

In order to motivate the primitives of the convolutional layer, let's build an intuition for how the human brain pieces together raw visual information into an understanding of the world around us. One of the most influential studies in this space came from David Hubel and Torsten Wiesel, who discovered that parts of the visual cortex are responsible for detecting edges. In 1959, they inserted electrodes into the brain of a cat and projected black-and-white patterns on the screen. They found that some

7 LeCun, Yann, et al. "Handwritten Digit Recognition with a Back-Propagation Network." *Advances in Neural Information Processing Systems*. 1990.

neurons fired only when there were vertical lines, others when there were horizontal lines, and still others when the lines were at particular angles.[8]

Further work determined that the visual cortex was organized in layers. Each layer is responsible for building on the features detected in the previous layers—from lines, to contours, to shapes, to entire objects. Furthermore, within a layer of the visual cortex, the same feature detectors were replicated over the whole area in order to detect features in all parts of an image. These ideas significantly impacted the design of convolutional neural nets.

The first concept that arose was that of a *filter*, and it turns out that here, Viola and Jones were actually pretty close. A filter is essentially a feature detector, and to understand how it works, let's consider the toy image in Figure 5-5.

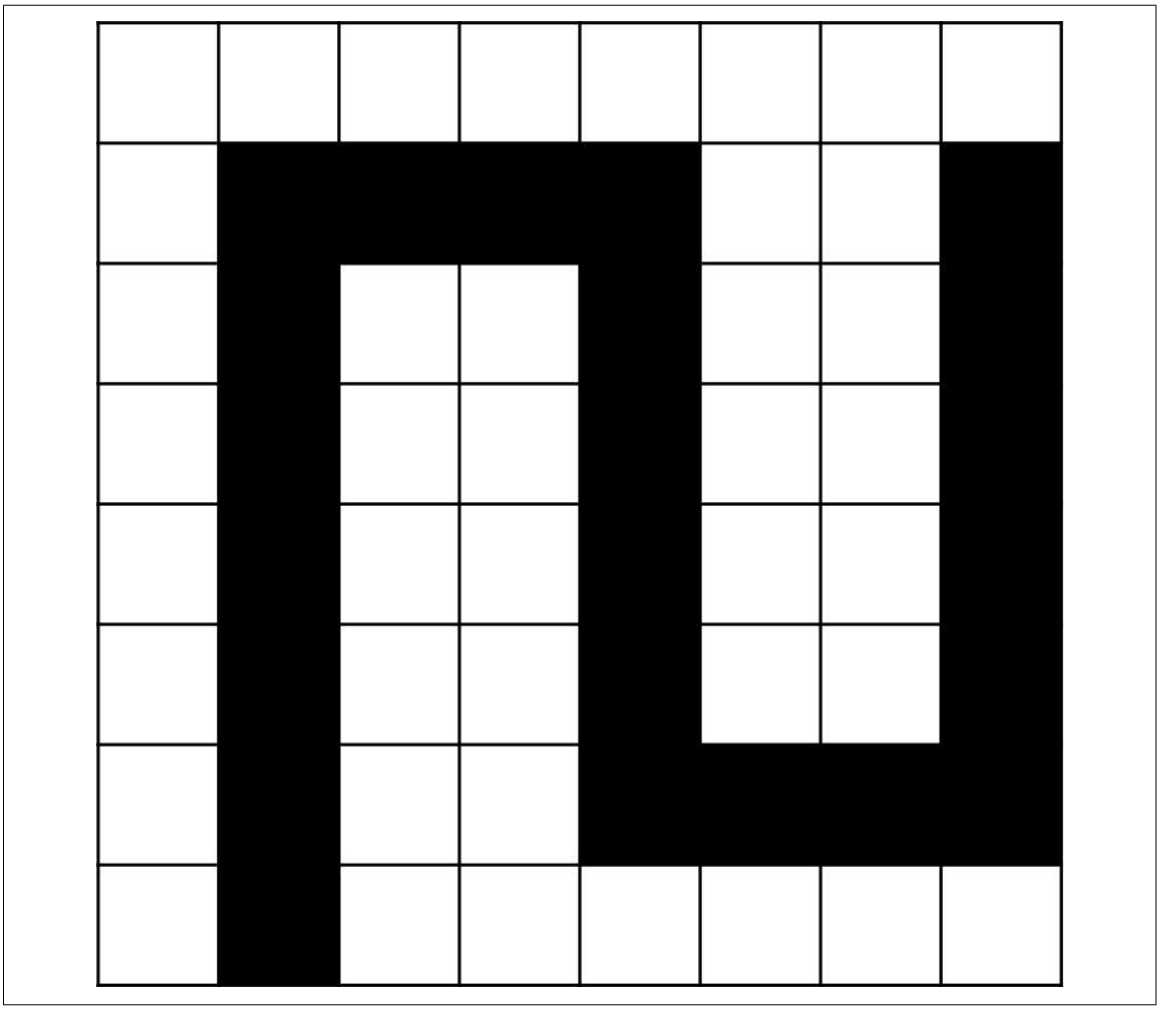

Figure 5-5. We'll analyze this simple black-and-white image as a toy example

8 Hubel, David H., and Torsten N. Wiesel. "Receptive fields of single neurones in the cat's striate cortex." *The Journal of Physiology* 148.3 (1959): 574-591.

Let's say that we want to detect vertical and horizontal lines in the image. One approach would be to use an appropriate feature detector, as shown in Figure 5-6. For example, to detect vertical lines, we would use the feature detector on the top, slide it across the entirety of the image, and at every step check if we have a match. We keep track of our answers in the matrix in the top right. If there's a match, we shade the appropriate box black. If there isn't, we leave it white. This result is our *feature map*, and it indicates where we've found the feature we're looking for in the original image. We can do the same for the horizontal line detector (bottom), resulting in the feature map in the bottom-right corner.

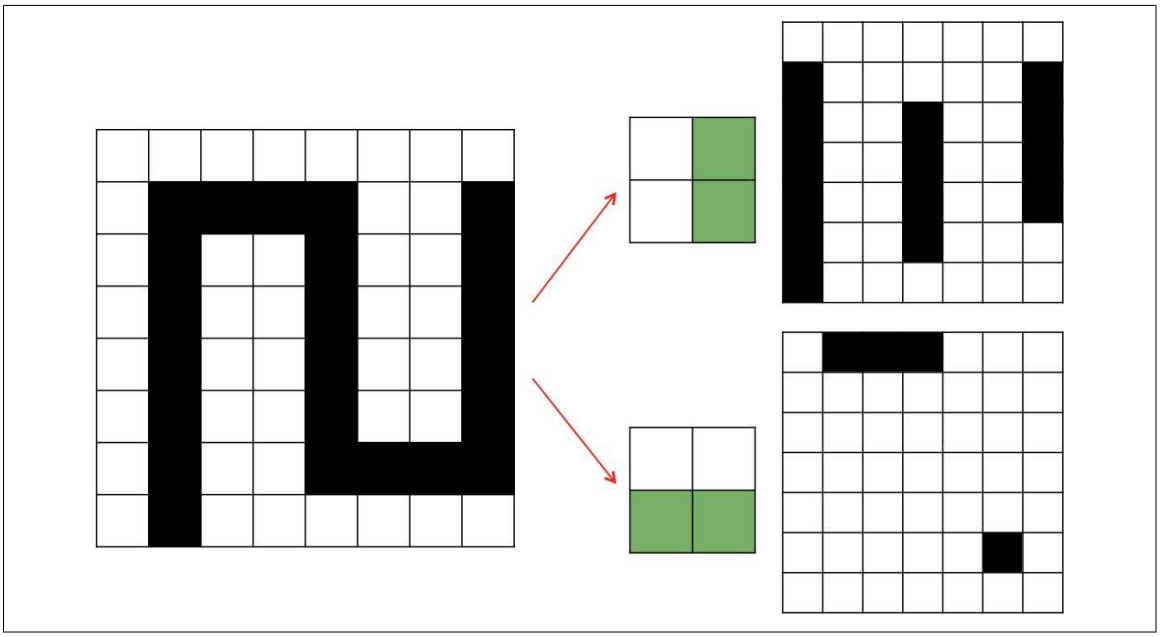

Figure 5-6. Applying filters that detect vertical and horizontal lines on our toy example

This operation is called a convolution. We take a filter and we multiply it over the entire area of an input image. Using the following scheme, let's try to express this operation as neurons in a network. In this scheme, layers of neurons in a feed-forward neural net represent either the original image or a feature map. Filters represent combinations of connections (one such combination is highlighted in Figure 5-7) that get replicated across the entirety of the input. In Figure 5-7, connections of the same color are restricted to always have the same weight. We can achieve this by initializing all the connections in a group with identical weights and by always averaging the weight updates of a group before applying them at the end of each iteration of backpropagation. The output layer is the feature map generated by this filter. A neuron in the feature map is activated if the filter contributing to its activity detected an appropriate feature at the corresponding position in the previous layer.

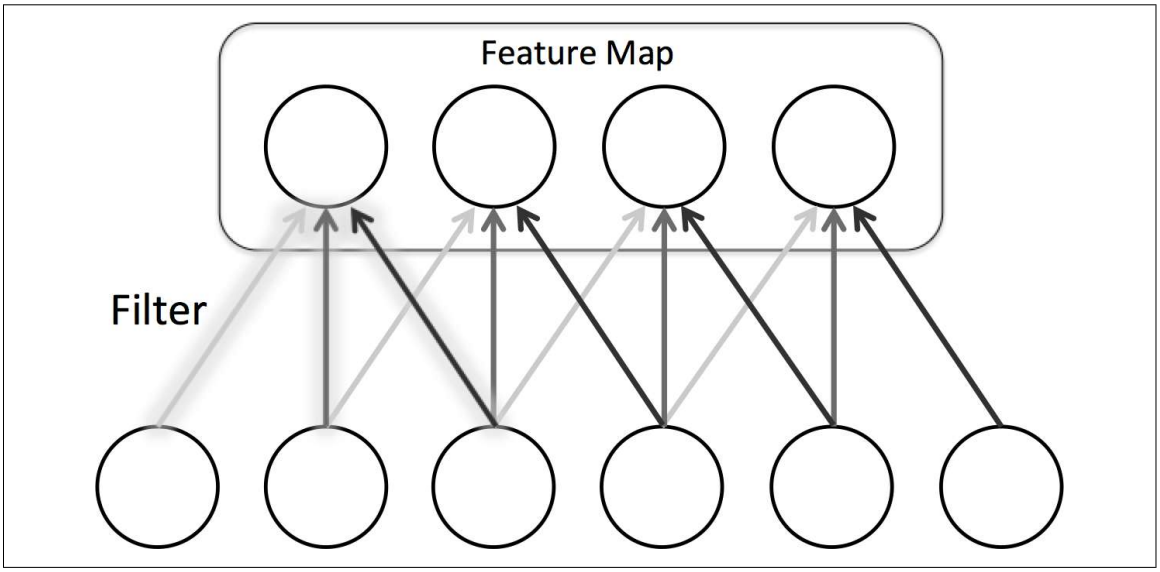

Figure 5-7. Representing filters and feature maps as neurons in a convolutional layer

Let's denote the k^{th} feature map in layer m as m^k. Moreover, let's denote the corresponding filter by the values of its weights W. Then assuming the neurons in the feature map have bias b^k (note that the bias is kept identical for all of the neurons in a feature map), we can mathematically express the feature map as follows:

$$m_{ij}^k = f\left((W * x)_{ij} + b^k\right)$$

This mathematical description is simple and succinct, but it doesn't completely describe filters as they are used in convolutional neural networks. Specifically, filters don't just operate on a single feature map. They operate on the entire volume of feature maps that have been generated at a particular layer. For example, consider a situation in which we would like to detect a face at a particular layer of a convolutional net. And we have accumulated three feature maps, one for eyes, one for noses, and one for mouths. We know that a particular location contains a face if the corresponding locations in the primitive feature maps contain the appropriate features (two eyes, a nose, and a mouth). In other words, to make decisions about the existence of a face, we must combine evidence over multiple feature maps. This is equally necessary for an input image that is of full color. These images have pixels represented as RGB values, and so we require three slices in the input volume (one slice for each color). As a result, feature maps must be able to operate over volumes, not just areas. This is shown below in Figure 5-8. Each cell in the input volume is a neuron. A local portion is multiplied with a filter (corresponding to weights in the convolutional layer) to produce a neuron in a filter map in the following volumetric layer of neurons.

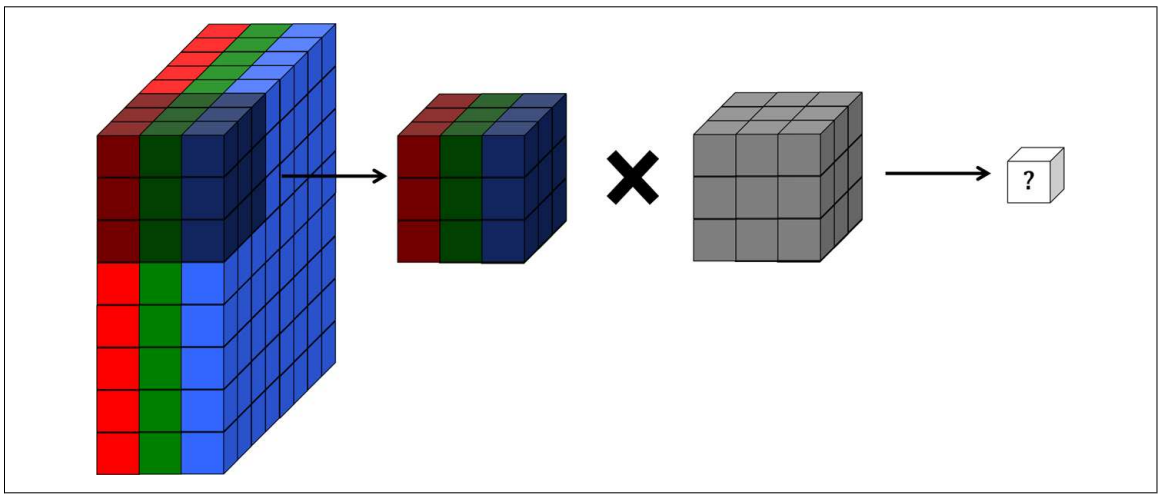

Figure 5-8. Representing a full-color RGB image as a volume and applying a volumetric convolutional filter

As we discussed in the previous section, a convolutional layer (which consists of a set of filters) converts one volume of values into another volume of values. The depth of the filter corresponds to the depth of the input volume. This is so that the filter can combine information from all the features that have been learned. The depth of the output volume of a convolutional layer is equivalent to the number of filters in that layer, because each filter produces its own slice. We visualize these relationships in Figure 5-9.

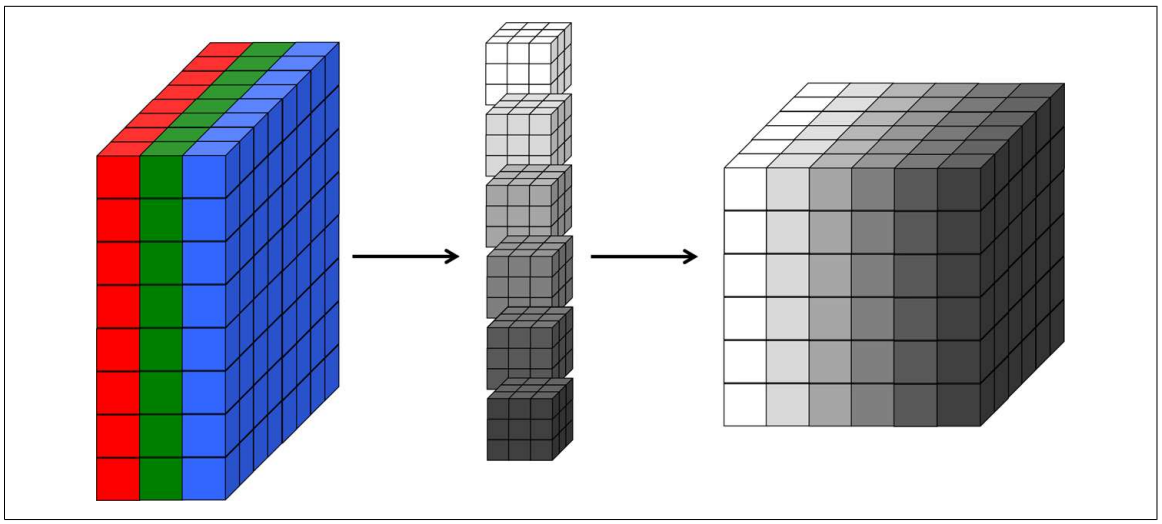

Figure 5-9. A three-dimensional visualization of a convolutional layer, where each filter corresponds to a slice in the resulting output volume

In the next section, we will use these concepts and fill in some of the gaps to create a full description of a convolutional layer.

Full Description of the Convolutional Layer

Let's use the concepts we've developed so far to complete the description of the convolutional layer. First, a convolutional layer takes in an input volume. This input volume has the following characteristics:

- Its *width* w_{in}
- Its *height* h_{in}
- Its *depth* d_{in}
- Its *zero padding* p

This volume is processed by a total of k filters, which represent the weights and connections in the convolutional network. These filters have a number of hyperparameters, which are described as follows:

- Their *spatial extent* e, which is equal to the filter's height and width.
- Their *stride* s, or the distance between consecutive applications of the filter on the input volume. If we use a stride of 1, we get the full convolution described in the previous section. We illustrate this in Figure 5-10.
- The bias b (a parameter learned like the values in the filter) which is added to each component of the convolution.

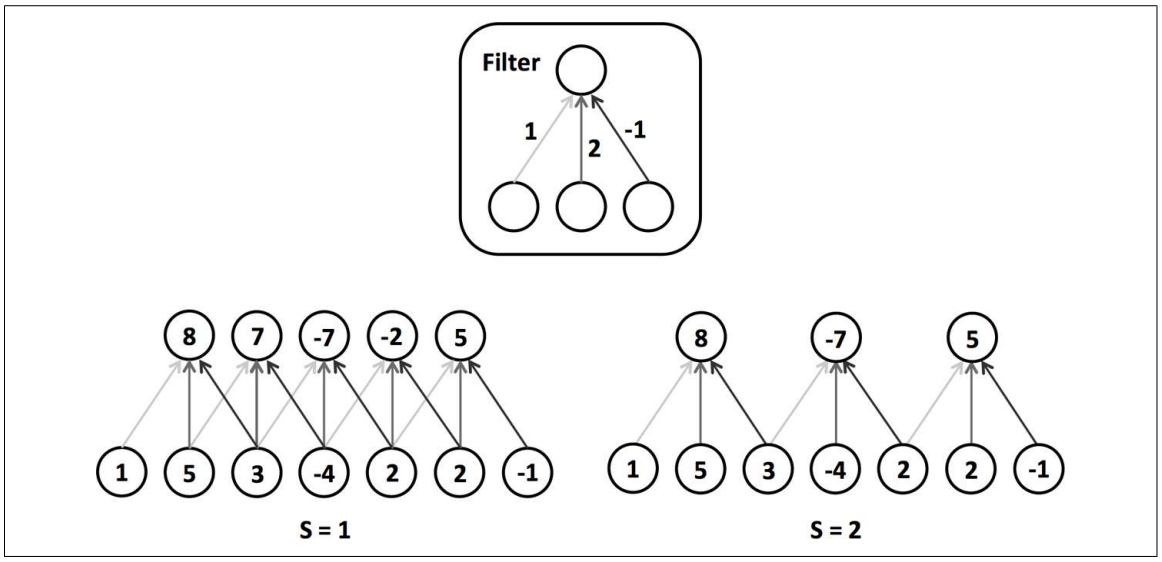

Figure 5-10. An illustration of a filter's stride hyperparameter

This results in an output volume with the following characteristics:

- Its function f, which is applied to the incoming logit of each neuron in the output volume to determine its final value
- Its *width* $w_{out} = \left\lceil \frac{w_{in} - e + 2p}{s} \right\rceil + 1$

- Its *height* $h_{out} = \left\lceil \frac{h_{in} - e + 2p}{s} \right\rceil + 1$
- Its *depth* $d_{out} = k$

The m^{th} "depth slice" of the output volume, where $1 \leq m \leq k$, corresponds to the function f applied to the sum of the m^{th} filter convoluted over the input volume and the bias b^m. Moreover, this means that per filter, we have $d_{in}e^2$ parameters. In total, that means the layer has $kd_{in}e^2$ parameters and k biases. To demonstrate this in action, we provide an example of a convolutional layer in Figure 5-11 and Figure 5-12 with a 5 x 5 x 3 input volume with zero padding $p = 1$. We'll use two 3 x 3 x 3 filters (spatial extent) with a stride $s = 2$. We'll use a linear function to produce the output volume, which will be of size 3 x 3 x 2.

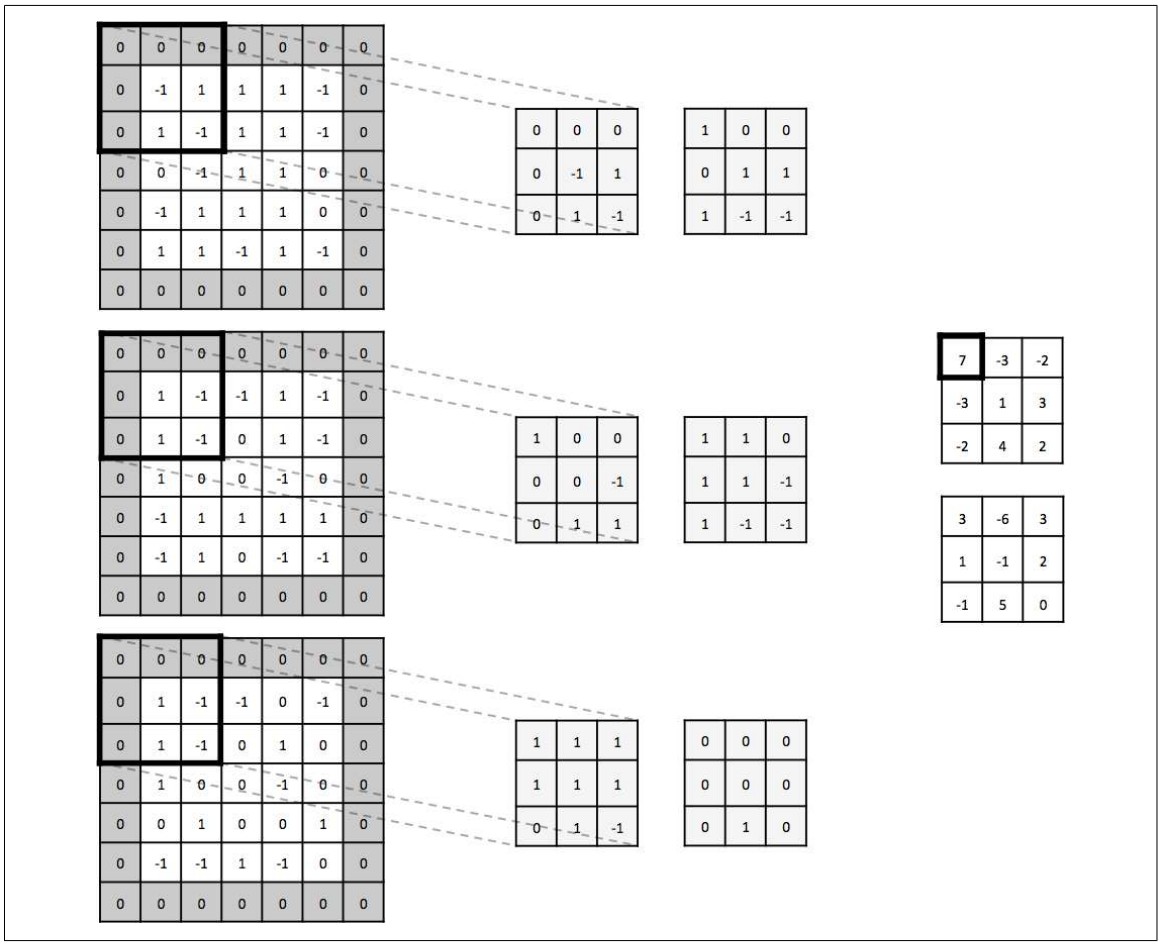

Figure 5-11. This is a convolutional layer with an input volume that has width 5, height 5, depth 3, and zero padding 1. There are 2 filters, with spatial extent 3 and applied with a stride of 2. It results in an output volume with width 3, height 3, and depth 2. We apply the first convolutional filter to the upper-leftmost 3 x 3 piece of the input volume to generate the upper-leftmost entry of the first depth slice.

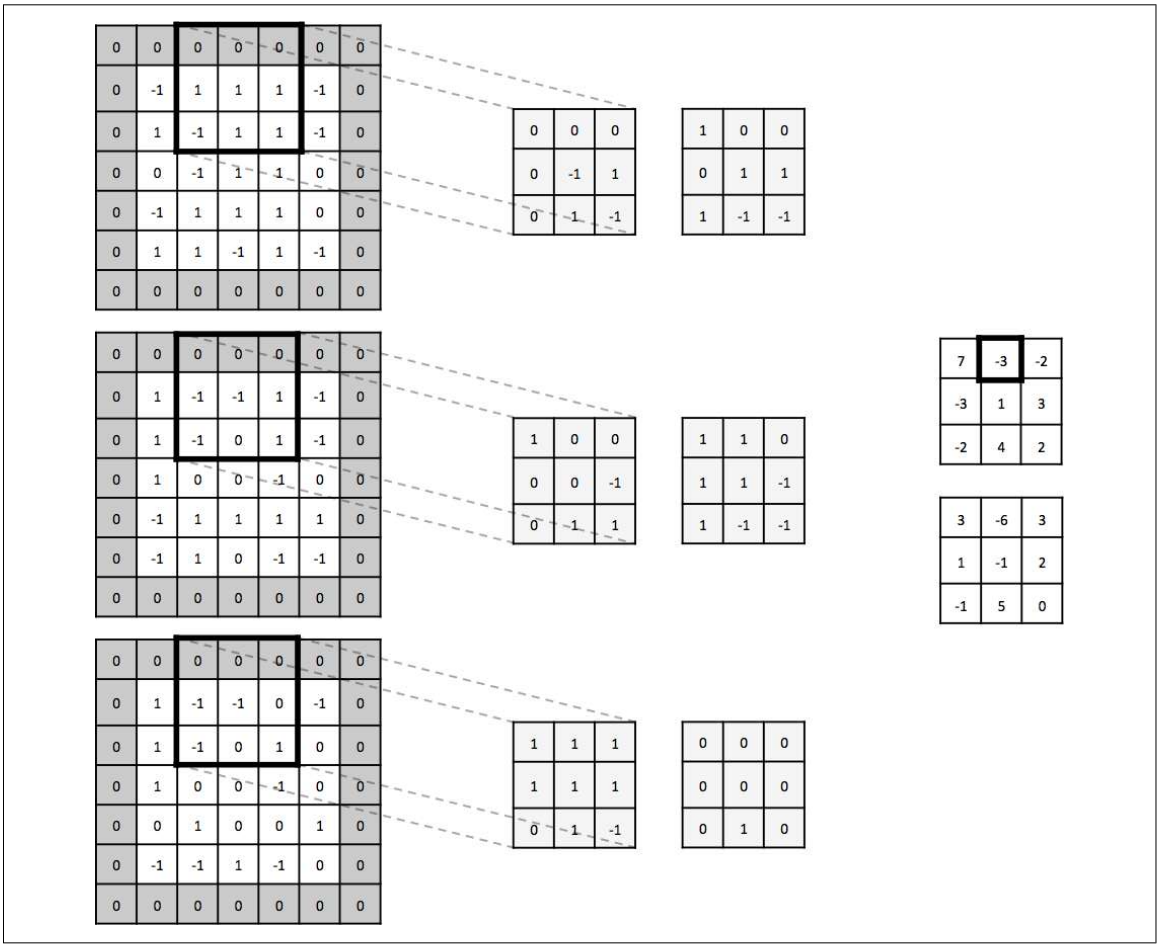

Figure 5-12. Using the same setup as Figure 5-11, we generate the next value in the first depth slice of the output volume.

Generally, it's wise to keep filter sizes small (size 3 x 3 or 5 x 5). Less commonly, larger sizes are used (7 x 7) but only in the first convolutional layer. Having more small filters is an easy way to achieve high representational power while also incurring a smaller number of parameters. It's also suggested to use a stride of 1 to capture all useful information in the feature maps, and a zero padding that keeps the output volume's height and width equivalent to the input volume's height and width.

TensorFlow provides us with a convenient operation to easily perform a convolution on a minibatch of input volumes (note that we must apply our choice of function f ourselves and it is not performed by the operation itself):[9]

```
tf.nn.conv2d(input, filter, strides, padding,
  use_cudnn_on_gpu=True,
              name=None)
```

9 *https://www.tensorflow.org/api_docs/python/tf/nn/conv2d*

Here, `input` is a four-dimensional tensor of size $N \times h_{in} \times w_{in} \times d_{in}$, where N is the number of examples in our minibatch. The `filter` argument is also a four-dimensional tensor representing all of the filters applied in the convolution. It is of size $e \times e \times d_{in} \times k$. The resulting tensor emitted by this operation has the same structure as `input`. Setting the `padding` argument to `"SAME"` also selects the zero padding so that height and width are preserved by the convolutional layer.

Max Pooling

To aggressively reduce dimensionality of feature maps and sharpen the located features, we sometimes insert a *max pooling* layer after a convolutional layer.[10] The essential idea behind max pooling is to break up each feature map into equally sized tiles. Then we create a condensed feature map. Specifically, we create a cell for each tile, compute the maximum value in the tile, and propagate this maximum value into the corresponding cell of the condensed feature map. This process is illustrated in Figure 5-13.

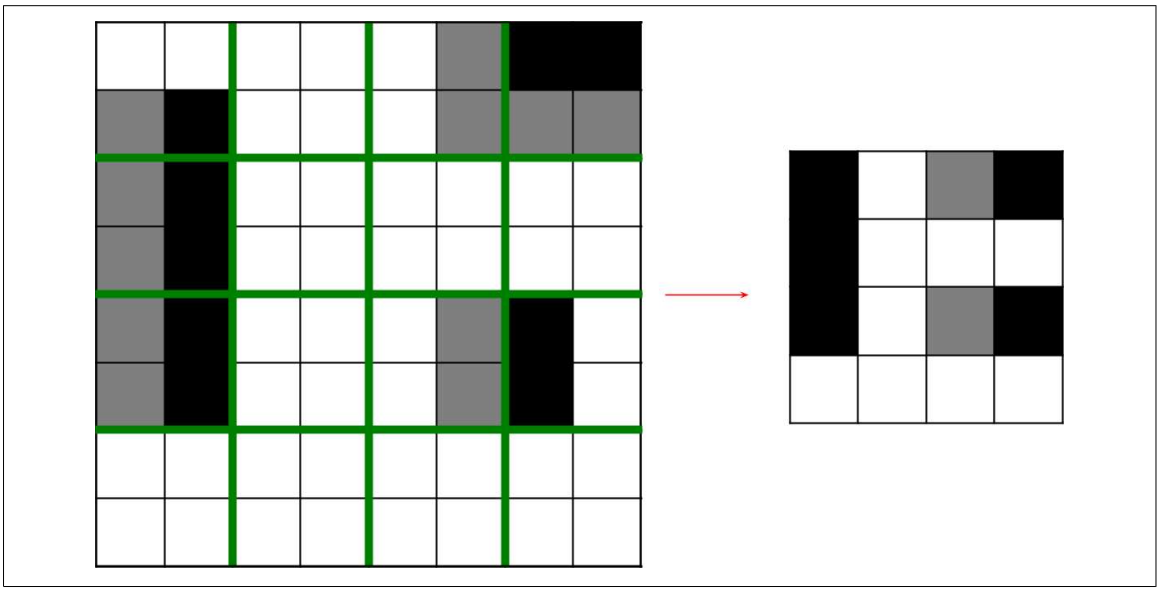

Figure 5-13. An illustration of how max pooling significantly reduces parameters as we move up the network

More rigorously, we can describe a pooling layer with two parameters:

- Its spatial extent e
- Its stride s

10 *https://www.tensorflow.org/api_docs/python/tf/nn/max_pool*

It's important to note that only two major variations of the pooling layer are used. The first is the nonoverlapping pooling layer with $e = 2, s = 2$. The second is the overlapping pooling layer with $e = 3, s = 2$. The resulting dimensions of each feature map are as follows:

- Its width $w_{out} = \left\lceil \frac{w_{in} - e}{s} \right\rceil + 1$
- Its height $h_{out} = \left\lceil \frac{h_{in} - e}{s} \right\rceil + 1$

One interesting property of max pooling is that it is *locally invariant*. This means that even if the inputs shift around a little bit, the output of the max pooling layer stays constant. This has important implications for visual algorithms. Local invariance is a very useful property if we care more about whether some feature is present than exactly where it is. However, enforcing large amounts of local invariance can destroy our network's ability to carry important information. As a result, we usually keep the spatial extent of our pooling layers quite small.

Some recent work along this line has come out of the University of Warwick from Graham[11], who proposes a concept called *fractional max pooling*. In fractional max pooling, a pseudorandom number generator is used to generate tilings with noninteger lengths for pooling. Here, fractional max pooling functions as a strong regularizer, helping prevent overfitting in convolutional networks.

Full Architectural Description of Convolution Networks

Now that we've described the building blocks of convolutional networks, we start putting them together. Figure 5-14 depicts several architectures that might be of practical use.

11 Graham, Benjamin. "Fractional Max-Pooling." *arXiv Preprint arXiv*:1412.6071 (2014).

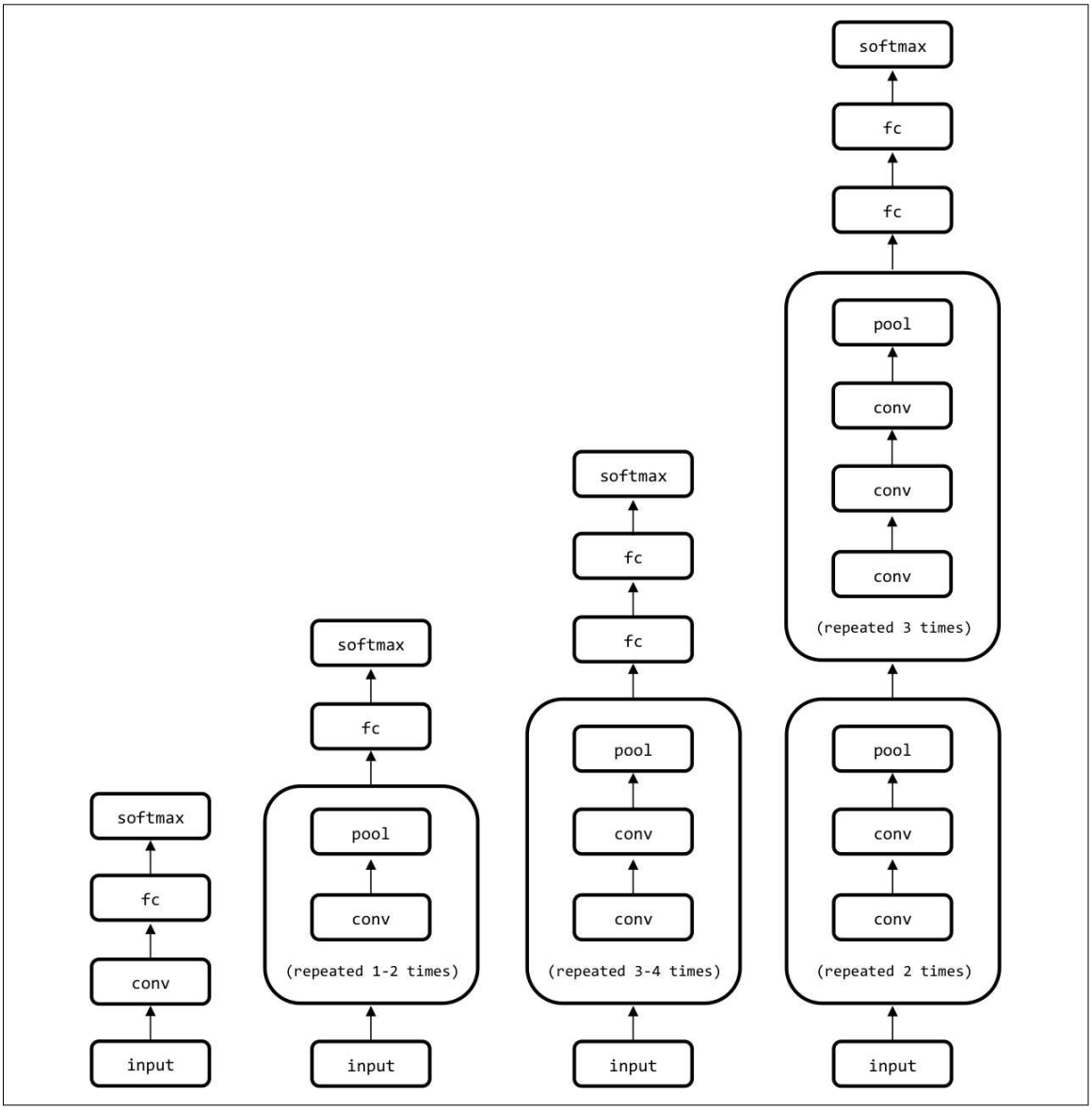

Figure 5-14. Various convolutional network architectures of various complexities. The architecture of VGGNet, a deep convolutional network built for ImageNet, is shown in the rightmost network.

One theme we notice as we build deeper networks is that we reduce the number of pooling layers and instead stack multiple convolutional layers in tandem. This is generally helpful because pooling operations are inherently destructive. Stacking several convolutional layers before each pooling layer allows us to achieve richer representations.

As a practical note, deep convolutional networks can take up a significant amount of space, and most casual practitioners are usually bottlenecked by the memory capacity on their GPU. The VGGNet architecture, for example, takes approximately 90 MB of memory on the forward pass per image and more than 180 MB of memory on the

backward pass to update the parameters.[12] Many deep networks make a compromise by using strides and spatial extents in the first convolutional layer that reduce the amount of information that needs to propagated up the network.

Closing the Loop on MNIST with Convolutional Networks

Now that we have a better understanding of how to build networks that effectively analyze images, we'll revisit the MNIST challenge we've tackled over the past several chapters. Here, we'll use a convolutional network to learn how to recognize handwritten digits. Our feed-forward network was able to achieve a 98.2% accuracy. Our goal will be to push the envelope on this result.

To tackle this challenge, we'll build a convolutional network with a pretty standard architecture (modeled after the second network in Figure 5-14): two pooling and two convolutional interleaved, followed by a fully connected layer (with dropout, $p = 0.5$) and a terminal softmax. To make building the network easy, we write a couple of helper methods in addition to our `layer` generator from the feed-forward network:

```
def conv2d(input, weight_shape, bias_shape):
    in = weight_shape[0] * weight_shape[1] * weight_shape[2]
    weight_init = tf.random_normal_initializer(stddev=
                                               (2.0/in)**0.5)
    W = tf.get_variable("W", weight_shape,
                        initializer=weight_init)
    bias_init = tf.constant_initializer(value=0)
    b = tf.get_variable("b", bias_shape, initializer=bias_init)
    conv_out = tf.nn.conv2d(input, W, strides=[1, 1, 1, 1],
                            padding='SAME')
    return tf.nn.relu(tf.nn.bias_add(conv_out, b))

def max_pool(input, k=2):
    return tf.nn.max_pool(input, ksize=[1, k, k, 1],
                          strides=[1, k, k, 1], padding='SAME')
```

The first helper method generates a convolutional layer with a particular shape. We set the stride to be to be 1 and the padding to keep the width and height constant between input and output tensors. We also initialize the weights using the same heuristic we used in the feed-forward network. In this case, however, the number of incoming weights into a neuron spans the filter's height and width and the input tensor's depth.

12 Simonyan, Karen, and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *arXiv Preprint arXiv*:1409.1556 (2014).

The second helper method generates a max pooling layer with non-overlapping windows of size k. The default, as recommended, is k=2, and we'll use this default in our MNIST convolutional network.

With these helper methods, we can now build a new inference constructor:

```
def inference(x, keep_prob):

    x = tf.reshape(x, shape=[-1, 28, 28, 1])
    with tf.variable_scope("conv_1"):
        conv_1 = conv2d(x, [5, 5, 1, 32], [32])
        pool_1 = max_pool(conv_1)

    with tf.variable_scope("conv_2"):
        conv_2 = conv2d(pool_1, [5, 5, 32, 64], [64])
        pool_2 = max_pool(conv_2)

    with tf.variable_scope("fc"):
        pool_2_flat = tf.reshape(pool_2, [-1, 7 * 7 * 64])
        fc_1 = layer(pool_2_flat, [7*7*64, 1024], [1024])

        # apply dropout
        fc_1_drop = tf.nn.dropout(fc_1, keep_prob)

    with tf.variable_scope("output"):
        output = layer(fc_1_drop, [1024, 10], [10])

    return output
```

The code here is quite easy to follow. We first take the flattened versions of the input pixel values and reshape them into a tensor of the $N \times 28 \times 28 \times 1$, where N is the number of examples in a minibatch, 28 is the width and height of each image, and 1 is the depth (because the images are black and white; if the images were in RGB color, the depth would instead be 3 to represent each color map). We then build a convolutional layer with 32 filters that have spatial extent 5. This results in taking an input volume of depth 1 and emitting a output tensor of depth 32. This is then passed through a max pooling layer which compresses the information. We then build a second convolutional layer with 64 filters, again with spatial extent 5, taking an input tensor of depth 32 and emitting an output tensor of depth 64. This, again, is passed through a max pooling layer to compress information.

We then prepare to pass the output of the max pooling layer into a fully connected layer. To do this, we flatten the tensor. We can do this by computing the full size of each "subtensor" in the minibatch. We have 64 filters, which corresponds to the depth of 64. We now have to determine the height and width after passing through two max pooling layers. Using the formulas we found in the previous section, it's easy to confirm that each feature map has a height and width of 7. Confirming this is left as an exercise for the reader.

After the reshaping operation, we use a fully connected layer to compress the flattened representation into a hidden state of size 1,024. We use a dropout probability in this layer of 0.5 during training and 1 during model evaluation (standard procedure for employing dropout). Finally, we send this hidden state into a softmax output layer with 10 bins (the softmax is, as usual, performed in the `loss` constructor for better performance).

Finally, we train our network using the Adam optimizer. After several epochs over the dataset, we achieve an accuracy of 99.4%, which isn't state of the art (approximately 99.7 to 99.8%), but is very respectable.

Image Preprocessing Pipelines Enable More Robust Models

So far we've been dealing with rather tame datasets. Why is MNIST a tame dataset? Well, fundamentally, MNIST has already been preprocessed so that all the images in the dataset resemble each other. The handwritten digits are perfectly cropped in just the same way; there are no color aberrations because MNIST is black and white; and so on. Natural images, however, are an entirely different beast.

Natural images are messy, and as a result, there are a number of preprocessing operations that we can utilize in order to make training slightly easier. The first technique that is supported out of the box in TensorFlow is approximate per-image whitening. The basic idea behind whitening is to zero-center every pixel in an image by subtracting out the mean and normalizing to unit 1 variance. This helps us correct for potential differences in dynamic range between images. In TensorFlow, we can achieve this using:

```
tf.image.per_image_whitening(image)
```

We also can expand our dataset artificially by randomly cropping the image, flipping the image, modifying saturation, modifying brightness, etc:

```
tf.random_crop(value, size, seed=None, name=None)
tf.image.random_flip_up_down(image, seed=None)
tf.image.random_flip_left_right(image, seed=None)
tf.image.transpose_image(image)
tf.image.random_brightness(image, max_delta, seed=None)
tf.image.random_contrast(image, lower, upper, seed=None)
tf.image.random_saturation(image, lower, upper, seed=None)
tf.image.random_hue(image, max_delta, seed=None)
```

Applying these transformations helps us build networks that are robust to the different kinds of variations that are present in natural images, and make predictions with high fidelity in spite of potential distortions.

Accelerating Training with Batch Normalization

In 2015, researchers from Google devised an exciting way to even further accelerate the training of feed-forward and convolutional neural networks using a technique called *batch normalization*.[13] We can think of the intuition behind batch normalization like a tower of blocks, as shown in Figure 5-15.

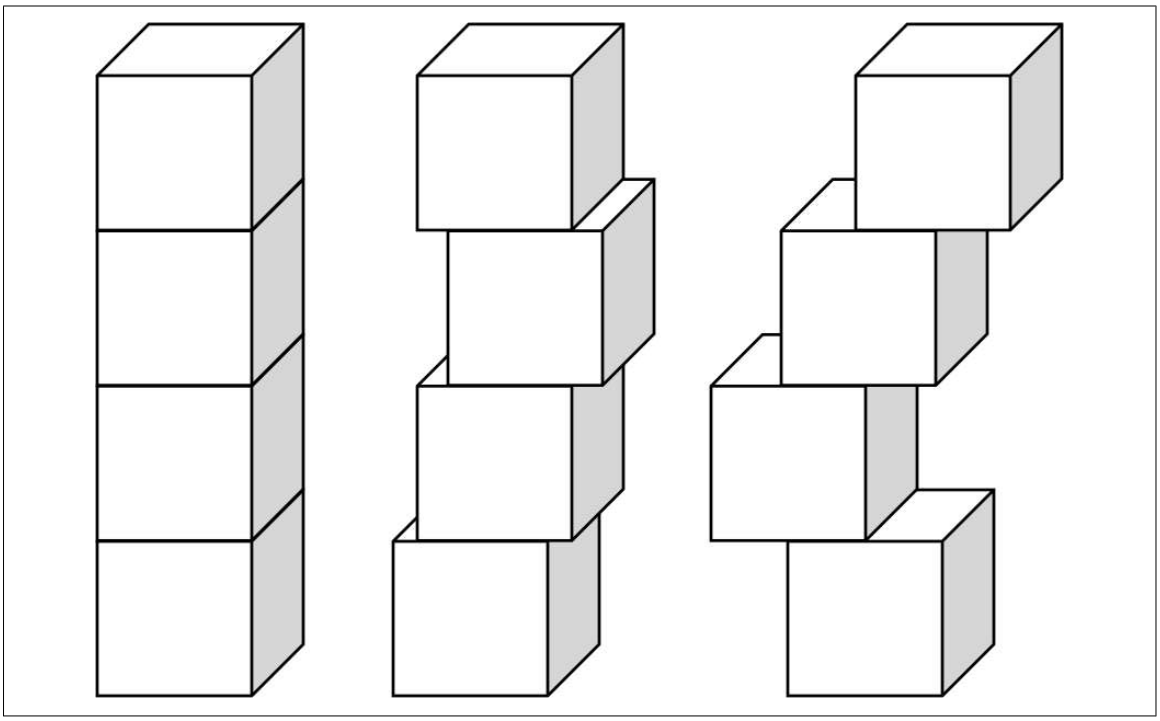

Figure 5-15. When blocks in a tower become shifted too drastically so that they no longer align, the structure can become very unstable

When a tower of blocks is stacked together neatly, the structure is stable. However, if we randomly shift the blocks, we could force the tower into configurations that are increasingly unstable. Eventually the tower falls apart.

A similar phenomenon can happen during the training of neural networks. Imagine a two-layer neural network. In the process of training the weights of the network, the output distribution of the neurons in the bottom layer begins to shift. The result of the changing distribution of outputs from the bottom layer means that the top layer not only has to learn how to make the appropriate predictions, but it also needs to somehow modify itself to accommodate the shifts in incoming distribution. This significantly slows down training, and the magnitude of the problem compounds the more layers we have in our networks.

13 S. Ioffe, C. Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *arXiv Preprint arXiv*:1502.03167. 2015.

Normalization of image inputs helps out the training process by making it more robust to variations. Batch normalization takes this a step further by normalizing inputs to every layer in our neural network. Specifically, we modify the architecture of our network to include operations that:

1. Grab the vector of logits incoming to a layer before they pass through the nonlinearity
2. Normalize each component of the vector of logits across all examples of the minibatch by subtracting the mean and dividing by the standard deviation (we keep track of the moments using an exponentially weighted moving average)
3. Given normalized inputs $\hat{\mathbf{x}}$, use an affine transform to restore representational power with two vectors of (trainable) parameters: $\gamma\hat{\mathbf{x}} + \beta$

Expressed in TensorFlow, batch normalization can be expressed as follows for a convolutional layer:

```
def conv_batch_norm(x, n_out, phase_train):
    beta_init = tf.constant_initializer(value=0.0,
                                        dtype=tf.float32)
    gamma_init = tf.constant_initializer(value=1.0,
                                         dtype=tf.float32)

    beta = tf.get_variable("beta", [n_out],
                           initializer=beta_init)
    gamma = tf.get_variable("gamma", [n_out],
                            initializer=gamma_init)

    batch_mean, batch_var = tf.nn.moments(x, [0,1,2],
                                          name='moments')
    ema = tf.train.ExponentialMovingAverage(decay=0.9)
    ema_apply_op = ema.apply([batch_mean, batch_var])
    ema_mean, ema_var = ema.average(batch_mean),
                        ema.average(batch_var)
    def mean_var_with_update():
        with tf.control_dependencies([ema_apply_op]):
            return tf.identity(batch_mean),
                   tf.identity(batch_var)
    mean, var = control_flow_ops.cond(phase_train,
        mean_var_with_update,
        lambda: (ema_mean, ema_var))

    normed = tf.nn.batch_norm_with_global_normalization(x,
             mean, var, beta, gamma, 1e-3, True)
    return normed
```

We can also express batch normalization for nonconvolutional feedforward layers, with a slight modification to how the moments are calculated, and a reshaping option for compatibility with `tf.nn.batch_norm_with_global_normalization`:

```
def layer_batch_norm(x, n_out, phase_train):
    beta_init = tf.constant_initializer(value=0.0,
                                        dtype=tf.float32)
    gamma_init = tf.constant_initializer(value=1.0,
                                         dtype=tf.float32)

    beta = tf.get_variable("beta", [n_out],
                           initializer=beta_init)
    gamma = tf.get_variable("gamma", [n_out],
                            initializer=gamma_init)

    batch_mean, batch_var = tf.nn.moments(x, [0],
                                          name='moments')
    ema = tf.train.ExponentialMovingAverage(decay=0.9)
    ema_apply_op = ema.apply([batch_mean, batch_var])
    ema_mean, ema_var = ema.average(batch_mean),
                        ema.average(batch_var)
    def mean_var_with_update():
        with tf.control_dependencies([ema_apply_op]):
            return tf.identity(batch_mean),
                   tf.identity(batch_var)
    mean, var = control_flow_ops.cond(phase_train,
        mean_var_with_update,
        lambda: (ema_mean, ema_var))

    x_r = tf.reshape(x, [-1, 1, 1, n_out])
    normed = tf.nn.batch_norm_with_global_normalization(x_r,
             mean, var, beta, gamma, 1e-3, True)
    return tf.reshape(normed, [-1, n_out])
```

In addition to speeding up training by preventing significant shifts in the distribution of inputs to each layer, batch normalization also allows us to significantly increase the learning rate. Moreover, batch normalization acts as a regularizer and removes the need for dropout and (when used) L2 regularization. Although we don't leverage it here, the authors also claim that batch regularization largely removes the need for photometric distortions, and we can expose the network to more "real" images during the training process.

Now that we've developed an enhanced toolkit for analyzing natural images with convolutional networks, we'll now build a classifier for tackling the CIFAR-10 challenge.

Building a Convolutional Network for CIFAR-10

The CIFAR-10 challenge consists of 32 x 32 color images that belong to one of 10 possible classes.[14] This is a surprisingly hard challenge because it can be difficult for even a human to figure out what is in a picture. An example is shown in Figure 5-16.

Figure 5-16. A dog from the CIFAR-100 dataset

In this section, we'll build networks both with and without batch normalization as a basis of comparison. We increase the learning rate by 10-fold for the batch normalization network to take full advantage of its benefits. We'll only display code for the batch normalization network here because building the vanilla convolutional network is very similar.

We distort random 24 × 24 crops of the input images to feed into our network for training. We use the example code provided by Google to do this. We'll jump right

14 Krizhevsky, Alex, and Geoffrey Hinton. "Learning Multiple Layers of Features from Tiny Images." (2009).

into the network architecture. To start, let's take a look at how we integrate batch normalization into the convolutional and fully connected layers. As expected, batch normalization happens to the logits before they're fed into a nonlinearity:

```
def conv2d(input, weight_shape, bias_shape, phase_train,
           visualize=False):
    incoming = weight_shape[0] * weight_shape[1]
                               * weight_shape[2]
    weight_init = tf.random_normal_initializer(stddev=
                  (2.0/incoming)**0.5)
    W = tf.get_variable("W", weight_shape,
                        initializer=weight_init)
    if visualize:
        filter_summary(W, weight_shape)
    bias_init = tf.constant_initializer(value=0)
    b = tf.get_variable("b", bias_shape, initializer=bias_init)
    logits = tf.nn.bias_add(tf.nn.conv2d(input, W,
             strides=[1, 1, 1, 1], padding='SAME'), b)
    return tf.nn.relu(conv_batch_norm(logits, weight_shape[3],
                                      phase_train))

def layer(input, weight_shape, bias_shape, phase_train):
    weight_init = tf.random_normal_initializer(stddev=
                  (2.0/weight_shape[0])**0.5)
    bias_init = tf.constant_initializer(value=0)
    W = tf.get_variable("W", weight_shape,
                        initializer=weight_init)
    b = tf.get_variable("b", bias_shape,
                        initializer=bias_init)
    logits = tf.matmul(input, W) + b
    return tf.nn.relu(layer_batch_norm(logits, weight_shape[1],
                      phase_train))
```

The rest of the architecture is straightforward. We use two convolutional layers (each followed by a max pooling layer). There are then two fully connected layers followed by a softmax. Dropout is included for reference, but in the batch normalization version, `keep_prob=1` during training:

```
def inference(x, keep_prob, phase_train):

    with tf.variable_scope("conv_1"):
        conv_1 = conv2d(x, [5, 5, 3, 64], [64], phase_train,
                        visualize=True)
        pool_1 = max_pool(conv_1)

    with tf.variable_scope("conv_2"):
        conv_2 = conv2d(pool_1, [5, 5, 64, 64], [64],
                        phase_train)
        pool_2 = max_pool(conv_2)

    with tf.variable_scope("fc_1"):
```

```
        dim = 1
        for d in pool_2.get_shape()[1:].as_list():
            dim *= d

        pool_2_flat = tf.reshape(pool_2, [-1, dim])
        fc_1 = layer(pool_2_flat, [dim, 384], [384],
                     phase_train)

        # apply dropout
        fc_1_drop = tf.nn.dropout(fc_1, keep_prob)

    with tf.variable_scope("fc_2"):

        fc_2 = layer(fc_1_drop, [384, 192], [192], phase_train)

        # apply dropout
        fc_2_drop = tf.nn.dropout(fc_2, keep_prob)

    with tf.variable_scope("output"):
        output = layer(fc_2_drop, [192, 10], [10], phase_train)

    return output
```

Finally, we use the Adam optimizer to train our convolutional networks. After some amount of time training, our networks are able to achieve an impressive 92.3% accuracy on the CIFAR-10 task without batch normalization and 96.7% accuracy with batch normalization. This result actually matches (and potentially exceeds) current state-of-the-art research on this task! In the next section, we'll take a closer look at learning and visualize how our networks perform.

Visualizing Learning in Convolutional Networks

On a high level, the simplest thing that we can do to visualize training is plot the cost function and validation errors over time as training progresses. We can clearly demonstrate the benefits of batch normalization by comparing the rates of convergence between our two networks. Plots taken in the middle of the training process are shown in Figure 5-17.

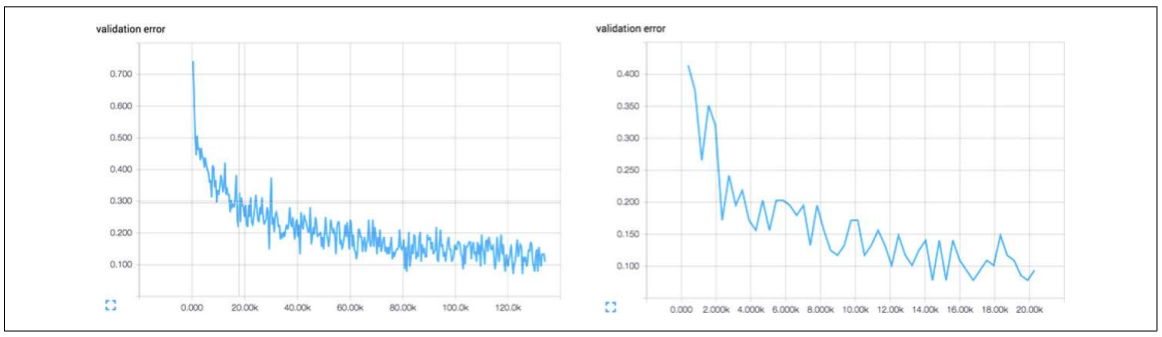

Figure 5-17. Training a convolutional network without batch normalization (left) versus with batch normalization (right). Batch normalization vastly accelerates the training process.

Without batch normalization, cracking the 90% accuracy threshold requires over 80,000 minibatches. On the other hand, with batch normalization, crossing the same threshold only requires slightly over 14,000 minibatches.

We can also inspect the filters that our convolutional network learns in order to understand what the network finds important to its classification decisions. Convolutional layers learn hierarchical representations, and so we'd hope that the first convolutional layer learns basic features (edges, simple curves, etc.), and the second convolutional layer will learn more complex features. Unfortunately, the second convolutional layer is difficult to interpret even if we decided to visualize it, so we only include the first layer filters in Figure 5-18.

Figure 5-18. A subset of the learned filters in the first convolutional layer of our network

We can make out a number of interesting features in our filters: vertical, horizontal, and diagonal edges, in addition to small dots or splotches of one color surrounded by another. We can be confident that our network is learning relevant features because the filters are not just noise.

We can also try to visualize how our network has learned to cluster various kinds of images pictorially. To illustrate this, we take a large network that has been trained on the ImageNet challenge and then grab the hidden state of the fully connected layer just before the softmax for each image. We then take this high-dimensional representation for each image and use an algorithm known as *t-Distributed Stochastic Neighbor Embedding*, or *t-SNE*, to compress it to a two-dimensional representation that we can visualize.[15] We don't cover the details of t-SNE here, but there are a number of

15 Maaten, Laurens van der, and Geoffrey Hinton. "Visualizing Data using t-SNE." *Journal of Machine Learning Research* 9.Nov (2008): 2579-2605.

publicly available software tools that will do it for us, including the script. We visualize the embeddings in Figure 5-19, and the results are quite spectacular.

Figure 5-19. The t-SNE embedding (center) surrounded by zoomed-in subsegments of the embedding (periphery). Image credit: Andrej Karpathy.[16]

At first, on a high level, it seems that images that are similarly colored are closer together. This is interesting, but what's even more striking is when we zoom into parts of the visualization, we realize that it's more than just color. We realize that all pictures of boats are in one place, all pictures of humans are in another place, and all pictures of butterflies are in yet another location in the visualization. Quite clearly, convolutional networks have spectacular learning capabilities.

16 *http://cs.stanford.edu/people/karpathy/cnnembed/*

Leveraging Convolutional Filters to Replicate Artistic Styles

Over the past couple of years, we've also developed algorithms that leverage convolutional networks in much more creative ways. One of these algorithms is called *neural style*.[17] The goal of neural style is to be able to take an arbitrary photograph and rerender it as if it were painted in the style of a famous artist. This seems like a daunting task, and it's not exactly clear how we might approach this problem if we didn't have a convolutional network. However, it turns out that clever manipulation of convolutional filters can produce spectacular results on this problem.

Let's take a pre-trained convolutional network. There are three images that we're dealing with. The first two are the source of content $\boldsymbol{p}$ and the source of style $\boldsymbol{a}$. The third image is the generated image $\boldsymbol{x}$. Our goal will be to derive an error function that we can backpropagate that, when minimized, will perfectly combine the content of the desired photograph and the style of the desired artwork.

We start with content first. If a layer in the network has k_l filters, then it produces a total of k_l feature maps. Let's call the size of each feature map m_l, the height times the width of the feature map. This means that the activations in all the feature maps of this layer can be stored in a matrix $\boldsymbol{F}^{(l)}$ of size $k_l \times m_l$. We can also represent all the activations of the photograph in a matrix $\boldsymbol{P}^{(l)}$ and all the activations of the generated image in the matrix $\boldsymbol{X}^{(l)}$. We use the `relu4_2` of the original VGGNet:

$$E_{\text{content}}(\boldsymbol{p}, \boldsymbol{x}) = \sum_{ij}(\boldsymbol{P}_{ij}^{(l)} - \boldsymbol{X}_{ij}^{(l)})^2$$

Now we can try tackling style. To do this we construct a matrix known as the *Gram matrix*, which represents correlations between feature maps in a given layer. The correlations represent the texture and feel that is common among all features, irrespective of which features we're looking at. Constructing the Gram matrix, which is of size $k_l \times k_l$, for a given image is done as follows:

$$\mathbf{G}^{(l)}_{ij} = \sum_{c=0}^{ml} \mathbf{F}^{(l)}_{ic} \mathbf{F}^{(l)}_{jc}$$

We can compute the Gram matrices for both the artwork in matrix $\boldsymbol{A}^{(l)}$ and the generated image in $\boldsymbol{G}^{(l)}$. We can then represent the error function as:

$$E_{style}(\mathbf{a}, \mathbf{x}) = \frac{1}{4k_l^2 m_l^2} \sum_{l=1}^{L} \sum_{ij} \frac{1}{L} \left(A_{ij}^{(l)} - G_{ij}^{(l)} \right)^2$$

Here, we weight each squared difference equally (dividing by the number of layers we want to include in our style reconstruction). Specifically, we use the `relu1_1`,

17 Gatys, Leon A., Alexander S. Ecker, and Matthias Bethge. "A Neural Algorithm of Artistic Style." *arXiv Preprint arXiv*:1508.06576 (2015).

`relu2_1`, `relu3_1`, `relu4_1`, and `relu5_1` layers of the original VGGNet. We omit full a discussion of the TensorFlow code (*http://bit.ly/2qAODnp*) for brevity, but the results, as shown in Figure 5-20, are again quite spectacular. We mix a photograph of the iconic MIT dome and Leonid Afremov's *Rain Princess*.

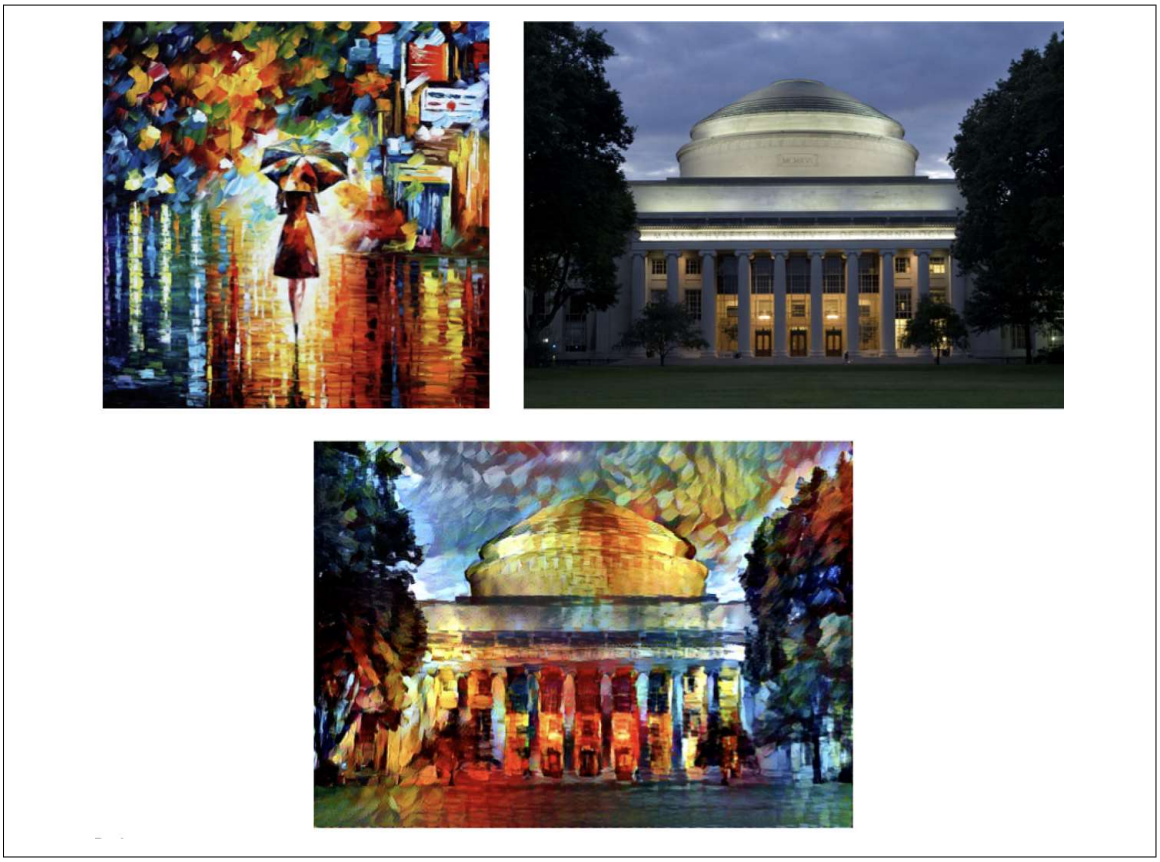

Figure 5-20. The result of mixing the Rain Princess with a photograph of the MIT Dome. Image credit: Anish Athalye.

Learning Convolutional Filters for Other Problem Domains

Although our examples in this chapter focus on image recognition, there are several other problem domains in which convolutional networks are useful. A natural extension of image analysis is video analysis. In fact, using five-dimensional tensors (including time as a dimension) and applying three-dimensional convolutions is an easy way to extend the convolutional paradigm to video.[18] Convolutional filters have

18 Karpathy, Andrej, et al. "Large-scale Video Classification with Convolutional Neural Networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014.

also been successfully used to analyze audiograms.[19] In these applications, a convolutional network slides over an audiogram input to predict phonemes on the other side.

Less intuitively, convolutional networks have also found some use in natural language processing. We'll see some examples of this in later chapters. More exotic uses of convolutional networks include teach algorithms to play board games, and analyzing biological molecules for drug discovery. We'll also discuss both of these examples in later chapters of this book.

Summary

In this chapter, we learned how to build neural networks that analyze images. We developed the concept of a convolution, and leveraged this idea to create tractable networks that can analyze both simple and more complex natural images. We built several of these convolutional networks in TensorFlow, and leveraged various image processing pipelines and batch normalization to make training our networks faster and more robust. Finally, we visualized the learning of convolutional networks and explored other interesting applications of the technology.

Images were easy to analyze because we were able to come up with effective ways to represent them as tensors. In other situations (e.g., natural language), it's less clear how one might represent our input data as tensors. To tackle this problem as a stepping stone to new deep learning models, we'll develop some key concepts in vector embeddings and representation learning in the next chapter.

19 Abdel-Hamid, Ossama, et al. "Applying Convolutional Neural Networks concepts to hybrid NN-HMM model for speech recognition." IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), Kyoto, 2012, pp. 4277-4280.