PREV
4. Extensions

NEXT
6. Recurrent Neural Networks

Chapter 5. Convolutional Neural Networks

In this chapter, we'll cover convolutional neural networks (CNNs). CNNs are the standard neural network architecture used for prediction when the input observations are images, which is the case in a wide range of neural network applications. So far in the book, we've focused exclusively on fully connected neural networks, which we implemented as a series of `Dense` layers. Thus, we'll start this chapter by reviewing some key elements of these networks and use this to motivate why we might want to use a different architecture for images. We'll then cover CNNs in a manner similar to that in which we introduced other concepts in this book: we'll first discuss how they work at a high level, then move to discussing them at a lower level, and finally show in detail how they work by coding up the convolution operation from scratch.[1] By the end of this chapter, you'll have a thorough enough understanding of how CNNs work to be able to use them both to solve problems and to learn about advanced CNN variants, such as ResNets, DenseNets, and Octave Convolutions on your own.

Neural Networks and Representation Learning

Neural networks initially receive data on observations, with each observation represented by some number *n* features. So far we've seen two examples of this in two very different domains: the first was the house prices dataset, where each observation was made up of 13 features, each of which represented a numeric characteristic about that house. The second was the MNIST dataset of handwritten digits; since the images were represented with 784 pixels (28 pixels wide by 28 pixels high), each observation was represented by 784 values indicating the lightness or darkness of each pixel.

In each case, after appropriately scaling the data, we were able to build a model that predicted the appropriate outcome for that dataset with high accuracy. Also in each case, a simple neural network model with one hidden layer performed better than a model without that hidden layer. Why is that? One reason, as I showed in the case of the house prices data, is that the neural network could learn *nonlinear* relationships between input and output. However, a more general reason is that in machine learning, we often need *linear combinations* of our original features in order to effectively predict our target. Let's say that the pixel values for an MNIST digit are x_1 through x_{784}. It could be the case, for example, contribute positively or negatively to the probability that an image is of a particular digit. Neural networks can automatically *discover* combinations of the original features that are important through their training process. That process starts by creating initially random combinations of the original features via multiplication by a random weight matrix; through training, the neural network learns to refine combinations that are helpful and discard those that aren't. This process of learning which combinations of features are important is known as *representation learning*, and it's the main reason why neural networks are successful across different domains. This is summarized in Figure 5-1.

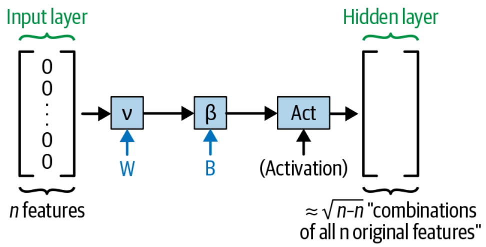

Figure 5-1. The neural networks we have seen so far start with n features and then learn somewhere between $\sqrt{n}$ and n "combinations" of these features to make predictions

Is there any reason to modify this process for image data? The fundamental insight that suggests the answer is "yes" is that *in images, the interesting "combinations of features" (pixels) tend to come from pixels that are close together in the image*. In an image, it is simply much less likely that an interesting feature will result from a combination of 9 randomly selected pixels throughout the image than from a 3 × 3 patch of adjacent pixels. We want to exploit this fundamental fact about image data: that the order of the features matters since it tells us which pixels are near each other spatially, whereas in the house prices data the order of the features doesn't matter. But how do we do it?

A Different Architecture for Image Data

The solution, at a high level, will be to create combinations of features, as before, but an order of magnitude more of them, and have each one be only a combination of the pixels from a small rectangular patch in the input image. Figure 5-2 describes this.

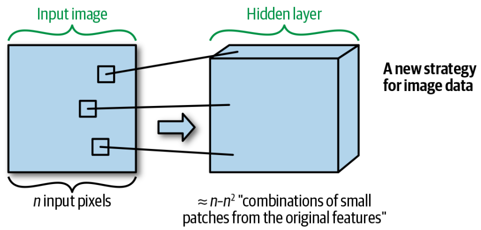

Figure 5-2. With image data, we can define each learned feature to be a function of a small patch of data, and thus define somewhere between n and n^2 output neurons

Having our neural network learn combinations of *all* of the input features—that

Nevertheless, previously it was at least extremely easy to compute new features that were combinations of all the input features: if we had *f* input features and wanted to compute *n* new features, we could simply multiply the `ndarray` containing our input features by an `f` × `n` matrix. What operation can we use to compute many combinations of the pixels from local patches of the input image? The answer is the convolution operation.

The Convolution Operation

Before we describe the convolution operation, let's make clear what we mean by "a feature that is a combination of pixels from a local patch of an image." Let's say we have a 5 × 5 input image *I*:

$$I = \begin{bmatrix} i_{11} & i_{12} & i_{13} & i_{14} & i_{15} \\ i_{21} & i_{22} & i_{23} & i_{24} & i_{25} \\ i_{31} & i_{32} & i_{33} & i_{34} & i_{35} \\ i_{41} & i_{42} & i_{43} & i_{44} & i_{45} \\ i_{51} & i_{52} & i_{53} & i_{54} & i_{55} \end{bmatrix}$$

And let's say we want to calculate a new feature that is a function of the 3 × 3 patch of pixels in the middle. Well, just as we've defined new features as linear combinations of old features in the neural networks we've seen so far, we'll define a new feature that is a function of this 3 × 3 patch, which we'll do by defining a 3 × 3 set of weights, *W*:

$$w = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix}$$

Then we'll simply take the dot product of *W* with the relevant patch from *I* to get the value of the feature in the output, which, since the section of the input image involved was centered at (3,3), we'll denote as o_{33} (the *o* stands for "output"):

$$o_{33} = w_{11} \times i_{22} + w_{12} \times i_{23} + w_{13} \times i_{24} + w_{21} \times i_{32} + w_{22} \times i_{33} + w_{23} \times i_{34} + w_{31} \times i_{42} + w_{32} \times i_{43} + w_{33} \times i_{44}$$

This value will then be treated like the other computed features we've seen in neural networks: it may have a bias added to it and then will probably be fed through an activation function, and then it will represent a "neuron" or "learned feature" that will get passed along to subsequent layers of the network. Thus we *can* define features that are functions of small patches of an input image.

How should we interpret such features? It turns out that features computed in this way have a special interpretation: they represent whether a *visual pattern defined by the weights* is present at that location of the image. The fact that 3 × 3 or 5 × 5 arrays of numbers can represent "pattern detectors" when their dot product is taken with the pixel values at each location of an image has been well known in the field of computer vision for a long time. For example, taking the dot product of the following 3 × 3 array of numbers:

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

with a given section of an input image detects whether there is an edge at that location of the image. There are similar matrices known to be able to detect whether corners exist, whether vertical or horizontal lines exist, and so on.[2]

Now suppose that we used the *same set of weights W* to detect whether the visual pattern defined by *W* existed at each location in the input image. We could imagine "sliding *W* over the input image," taking the dot product of *W* with the pixels at each location of the image, and ending up with a new image *O* of almost identical size to the original image (it may be slightly different, depending on how we handle the edges). This image *O* would be a kind of "feature map" showing the locations in the input image where the pattern defined by *W* was present. This operation is in fact what happens in convolutional neural networks:

This operation is at the core of how CNNs work. Before we can incorporate it into a full-fledged `Operation`, of the kind we've seen in the prior chapters, we have to add another dimension to it—literally.

The Multichannel Convolution Operation

To review: convolutional neural networks differ from regular neural networks in that they create an order of magnitude more features, and in that each feature is a function of just a small patch from the input image. Now we can get more specific: starting with *n* input pixels, the convolution operation just described will create *n* output features, one for each location in the input image. What actually happens in a convolutional `Layer` in a neural network goes one step further: there, we'll create *f sets* of *n* features, *each* with a corresponding (initially random) set of weights defining a visual pattern whose detection at each location in the input image will be captured in the feature map. These *f* feature maps will be created via *f* convolution operations. This is captured in Figure 5-3.

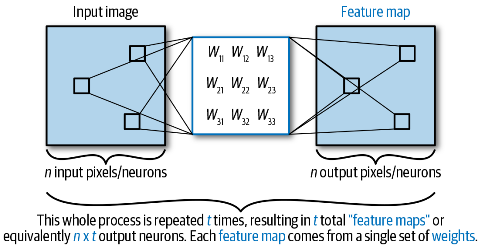
Input image
Feature map
W_{11} W_{12} W_{13}
W_{21} W_{22} W_{23}
W_{31} W_{32} W_{33}
n input pixels/neurons
n output pixels/neurons
This whole process is repeated *t* times, resulting in *t* total "feature maps" or equivalently *n* x *t* output neurons. Each feature map comes from a single set of weights.

Figure 5-3. More specifically than before, for an input image with n pixels, we define an output with f feature maps, each of which has about the same size as the original image, for a total of n × f total output neurons for the image, each of which is a function of only a small patch of the original image

Now that we've introduced a bunch of concepts, let's define them for clarity. While each "set of features" detected by a particular set of weights is called a feature map, in the context of a convolutional `Layer`, the number of feature maps is referred to as the number of *channels* of the `Layer`—this is why the operation involved with the `Layer` is called the multichannel convolution. In addition, the *f* sets of weights W_i are called the convolutional *filters*.[3]

Convolutional Layers

Now that we understand the multichannel convolution operation, we can think about how to incorporate this operation into a neural network layer. Previously, our neural network layers were relatively straightforward: they received two-dimensional `ndarray`s as input and produced two-dimensional `ndarray`s as output. Based on the description in the prior section, however, convolutional layers will have a 3D `ndarray` as output *for a single image*, with dimensions *number of channels* (same as "feature maps") × *image height* × *image width*.

This raises a question: how can we feed this `ndarray` forward into another convolutional layer to create a "deep convolutional" neural network? We've seen how to perform the convolution operation on an image with a single channel and our filters; how can we perform the multichannel convolution on an *input* with *multiple* channels, as we'll have to do when two convolutional layers are strung together? Understanding this is the key to understanding *deep* convolutional neural networks.

Consider what happens in a neural network with fully connected layers: in the

h_2 "features of features" of the original features. To create this next layer of h_2 features, we use $h_1 \times h_2$ weights to represent that each of the h_2 features is a function of each of the h_1 features in the prior layer.

As described in the prior section, an analogous process happens in the first layer of a convolutional neural network: we first transform the input image into m_1 *feature maps*, using m_1 convolutional *filters*. We should think of the output of this layer as representing whether each of the m_1 different visual patterns represented by the weights of the m_1 filters is present at each location in the input image. Just as different layers of a fully connected neural network can contain different numbers of neurons, the next layer of the convolutional neural network could contain m_2 filters. In order for the network to learn complex patterns, the interpretation of each of these should be whether each of the *"patterns of patterns"* or higher-order visual features represented by *combinations of the m_1 visual patterns from the prior layer* was present at that location of the image. This implies that if the output of the convolutional layer is a 3D `ndarray` of shape m_2 channels × image height × image width, then a given location in the image on one of the m_2 feature maps is *a linear combination of convolving m_1 different filters over that same location in each of the corresponding m_1 feature maps from the prior layer*. This will allow each location in each of the m_2 filter maps to represent a *combination* of the m_1 visual features already learned in the prior convolutional layer.

Implementation Implications

This understanding of how two multichannel convolutional layers are connected tells us how to implement the operation: just as we need $h_1 \times h_2$ weights to connect a fully connected layer with h_1 neurons to one with h_2, we need $m_1 \times m_2$ *convolutional filters* to connect a convolutional layer with m_1 channels to one with m_2. With this last detail in place, we can now specify the dimensions of the `ndarray`s that will make up the input, output, and parameters of the full, multichannel convolution operation:

1. The input will have shape:
    - Batch size
    - Input channels
    - Image height
    - Image width
2. The output will have shape:
    - Batch size
    - Output channels
    - Image height
    - Image width
3. The convolutional filters themselves will have shape:
    - Input channels
    - Output channels
    - Filter height
    - Filter width

NOTE

The order of the dimensions may vary from library to library, but these four dimensions will always be present.

We'll keep all of this in mind when we implement this convolution operation later in the chapter.

The Differences Between Convolutional and Fully Connected Layers

At the beginning of the chapter, we discussed the differences between convolutional and fully connected layers at a high level; Figure 5-4 revisits that comparison, now that we've described convolutional layers in more detail.

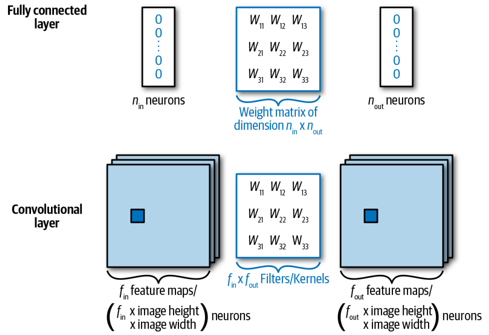
Fully connected layer
0 0 ⋮ 0 0
n_{in} neurons
W_{11} W_{12} W_{13} W_{21} W_{22} W_{23} W_{31} W_{32} W_{33}
Weight matrix of dimension n_{in} x n_{out}
0 0 ⋮ 0 0
n_{out} neurons
Convolutional layer
f_{in} feature maps/
$\left(f_{in}$ x image height x image width$\right)$ neurons
W_{11} W_{12} W_{13} W_{21} W_{22} W_{23} W_{31} W_{32} W_{33}
f_{in} x f_{out} Filters/Kernels
f_{out} feature maps/
$\left(f_{out}$ x image height x image width$\right)$ neurons

Figure 5-4. Comparison between convolutional and fully connected layers

In addition, one last difference between the two kinds of layers is the way in which the individual neurons themselves are interpreted:

- The interpretation of each neuron of a fully connected layer is that it detects whether or not *a particular combination of the features learned by the prior layer* is present in the current observation.
- The interpretation of a neuron of a convolutional layer is that it detects whether or not *a particular combination of visual patterns* learned by the prior layer is present *at the given location* of the input image.

There's one more problem we need to solve before we can incorporate such a layer into a neural network: how to use the dimensional `ndarray`s we obtain as output to make predictions.

Making Predictions with Convolutional Layers: The Flatten Layer

We've covered how convolutional layers learn features that represent whether visual patterns exist in images and store those features in layers of feature maps; how do we use these layers of feature maps to make predictions? When using fully connected neural networks to predict which of 10 classes an image belonged to in the prior chapter, we just had to ensure that the last layer had dimension 10; we could then feed these 10 numbers into the softmax cross entropy loss function to ensure they were interpreted as probabilities. Now we need to figure out what we can do in the case of our convolutional layer, where

To see the answer, recall that each neuron simply represents whether a particular combination of visual features (which, if this is a deep convolutional neural network, could be a feature of features or a feature of features of features) is present at a given location in the image. This is no different from the features that would be learned if we applied a fully connected neural network to this image: the first fully connected layer would represent features of the individual pixels, the second would represent features of these features, and so on. And in a fully connected architecture, we would simply treat each "feature of features" that the network had learned as a single neuron that would be used as input to a prediction of which class the image belonged to.

It turns out that we can do the same thing with convolutional neural networks—we treat the *m* feature maps as $m \times image_{height} \times image_{width}$ neurons and use a `Flatten` operation to squash these three dimensions (the number of channels, the image height, and the image width) down into a one-dimensional vector, after which we can use a simple matrix multiplication to make our final predictions. The intuition for why this works is that each individual neuron *fundamentally represents the same "kind of thing"* as the neurons in a fully connected layer—specifically, whether a given visual feature (or combination of features) is present at a given location in an image)—and thus we can treat them the same way in the final layer of the neural network.[4]

We'll see how to implement the `Flatten` layer later in the chapter. But before we dive into the implementation, let's discuss another kind of layer that is important in many CNN architectures, though we won't cover it in great detail in this book.

Pooling Layers

Pooling layers are another kind of layer commonly used in convolutional neural networks. They simply *downsample* each of the feature maps created by a convolution operation; for the most typically used pooling size of 2, this involves mapping each 2 × 2 section of each feature map either to the maximum value of that section, in the case of *max-pooling*, or to the average value of that section, in the case of *average-pooling*. For an *n* × *n* image, then, this would map the entire image to one of size $\frac{n}{2} \times \frac{n}{2}$. Figure 5-5 illustrates this.

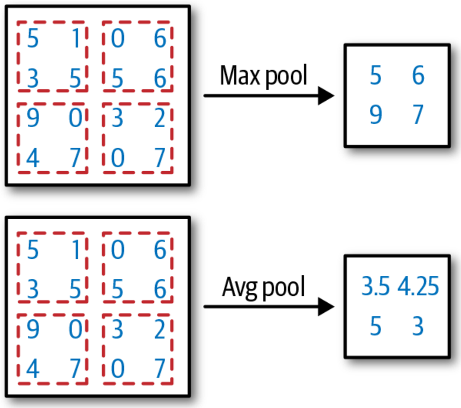

Figure 5-5. An illustration of max- and average-pooling with a 4 × 4 input; each 2 × 2 patch is mapped to either the average or the max values for that patch

The main advantage of pooling is computational: by downsampling the image to by a factor of 4; this can be further compounded if multiple pooling layers are used in the network, as they were in many architectures in the early days of CNNs. The downside of pooling, of course, is that only one-fourth as much information can be extracted from the downsampled image. However, the fact that architectures showed very strong performance on benchmarks in image recognition despite the use of pooling suggested that, even though pooling was causing the networks to "lose information" about the images by decreasing the images' resolution, the trade-offs in terms of increased computational speed were worth it. Nevertheless, pooling was considered by many to be a trick that just happened to work but should probably be done away with; as Geoffrey Hinton wrote on a Reddit AMA in 2014, "The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster." And indeed, most recent CNN architectures (such as Residual Networks, or "ResNets"[5]) use pooling minimally or not at all. Thus, in this book, we're not going to implement pooling layers, but given their importance for "putting CNNs on the map" via their use in famous architectures such as AlexNet, we mention them here for completeness.

APPLYING CNNS BEYOND IMAGES

Everything we have described so far is extremely standard for dealing with images using neural networks: the images are typically represented as a set of m_1 channels of pixels, where $m_1 = 1$ for black-and-white images, and $m_1 = 3$ for color images—and then some number m_2 of convolution operations are applied to each channel (using the $m_1 \times m_2$ filter maps as explained previously), with this pattern continuing on for several layers. This has all been covered in other treatments of convolutional neural networks; what is less commonly covered is that the idea of organizing data into "channels" and then processing that data using a CNN goes beyond just images. For example, this data representation was a key to DeepMind's series of AlphaGo programs showing that neural networks could learn to play Go. To quote the paper:[6]

> *The input to the neural network is a 19 × 19 × 17 image stack comprising 17 binary feature planes. 8 feature planes X_t consist of binary values indicating the presence of the current player's stones ($X_{t_i} = 1$ if intersection i contains a stone of the player's color at time-step t; 0 if the intersection is empty, contains an opponent stone, or if $t < 0$). A further 8 feature planes, Y_t, represent the corresponding features for the opponent's stones. The final feature plane, C, represents the color to play, and has a constant value of either 1 if black is to play or 0 if white is to play. These planes are concatenated together to give input features $s_t = X_t, Y_t, X_{t-1}, Y_{t-1}, \ldots, X_{t-7}, Y_{t-7}, C$. History features X_t, Y_t are necessary because Go is not fully observable solely from the current stones, as repetitions are forbidden; similarly, the color feature C is necessary because the komi is not observable.*

In other words, they essentially represented the board as a 19 × 19 pixel "image" with 17 channels! They used 16 of these channels to encode what had happened on the 8 prior moves that each player had taken; this was necessary so that they could encode rules that prevented repetition of earlier moves. The 17th channel was actually a 19 × 19 grid of either all 1s or all 0s, depending on whose turn it was to move.[7] CNNs and their multichannel convolution operations are mostly commonly applied to images, but the even more general idea of representing data that is arranged along some spatial dimension with multiple "channels" is applicable even beyond images.

In keeping with the theme of this book, however, to truly understand the multichannel convolution operation, you have to implement it from scratch; the next several sections will describe this process in detail.

Implementing the Multichannel Convolution Operation

It turns out that implementing this daunting operation—which involves a four-dimensional input `ndarray` and a four-dimensional parameter `ndarray`—is much clearer if we first examine the *one*-dimensional case. Building up to the full

The Forward Pass

The convolution in one dimension is conceptually identical to the convolution in two dimensions: we take in a one-dimensional input and a one-dimensional convolutional filter as inputs and then create the output by sliding the filter along the input.

Let’s suppose our input is of length 5:

$$\text{Input: } [t_1, t_2, t_3, t_4, t_5]$$

And let’s say the size of the “patterns” we want to detect is length 3:

$$\text{Filter: } [w_1, w_2, w_3]$$

DIAGRAMS AND MATH

The first element of the output would be created by convolving the first element of the input with the filter:

$$\text{Output Feature 1: } t_1 w_1 + t_2 w_2 + t_3 w_3$$

The second element of the output would be created by sliding the filter one unit to the right and convolving it with the next set values of the series:

$$\text{Output Feature 2: } t_2 w_1 + t_3 w_2 + t_4 w_3$$

Fair enough. However, when we compute the next output value, we realize that we have run out of room:

$$\text{Output Feature 3: } O_3 = t_3 w_1 + t_4 w_2 + t_5 w_3$$

We have hit the end of our input, and the resulting output has just three elements, when we started with five! How can we address this?

PADDING

To avoid the output shrinking as a result of the convolution operation, we’ll introduce a trick used throughout convolutional neural networks: we “pad” the input with zeros around the edges, enough so that the output remains the same size as the input. Otherwise, every time we convolve a filter over the input, we’ll end up with an output that is slightly smaller than the input, as seen previously.

size as the input. More generally, since we almost always use odd-numbered filter sizes, we add padding equal to the filter size divided by 2 and rounded down to the nearest integer.

Let's say we add this padding, so that instead of the input ranging from i_1 to i_5, it ranges from i_0 to i_6, where both i_0 and i_6 are 0. Then we can compute the output of the convolution as:

$$o_1 = i_0 \times w_1 + i_1 \times w_2 + i_2 \times w_3$$

And so on, up until:

$$o_5 = i_4 \times w_1 + i_5 \times w_2 + i_6 \times w_3$$

And now the output is the same size as the input. How might we code this up?

CODE

Coding up this part turns out to be pretty straightforward. Before we do, let's summarize the steps we just discussed:

1. We ultimately want to produce an output that is the same size as the input.
2. To do this without "shrinking" the output, we'll first need to pad the input.
3. Then we'll have to write some sort of loop that goes through the input and convolves each position of it with the filter.

We'll start with our input and our filter:

```
input_1d = np.array([1,2,3,4,5])
param_1d = np.array([1,1,1])
```

Here's a helper function that can pad our one-dimensional input on each end:

```
def _pad_1d(inp: ndarray,
            num: int) -> ndarray:
    z = np.array([0])
    z = np.repeat(z, num)
    return np.concatenate([z, inp, z])

_pad_1d(input_1d, 1)
```

```
array([0., 1., 2., 3., 4., 5., 0.])
```

What about the convolution itself? Observe that for each element in the output that we want to produce, we have a corresponding element in the *padded* input where we "start" the convolution operation; once we figure out where to start, we simply loop through all the elements in the filter, doing a multiplication at each element and adding the result to the total.

How do we find this "corresponding element"? Note that, simply, the value at the first element in the output gets its value starting at the first element of the padded input! This makes the `for` loop quite easy to write:

```
def conv_1d(inp: ndarray,
            param: ndarray) -> ndarray:

    # assert correct dimensions
    assert_dim(inp, 1)
    assert_dim(param, 1)

    # pad the input
    param_len = param.shape[0]
    param_mid = param_len // 2
    input_pad = _pad_1d(inp, param_mid)
    # perform the 1d convolution
    for o in range(out.shape[0]):
        for p in range(param_len):
            out[o] += param[p] * input_pad[o+p]

    # ensure shapes didn't change
    assert_same_shape(inp, out)

    return out

conv_1d_sum(input_1d, param_1d)
```

```
array([ 3.,  6.,  9., 12.,  9.])
```

That’s simple enough. Before we move on to the backward pass of this operation—the tricky part—let’s briefly discuss a hyperparameter of convolutions that we’re glossing over: stride.

A NOTE ON STRIDE

We noted earlier that pooling operations were one way to downsample images from feature maps. In many early convolutional architectures, these did indeed significantly reduce the amount of computation needed without any significant hit to accuracy; nevertheless, they’ve fallen out of favor because of their downside: they effectively downsample the image so that an image with just half the resolution is passed forward into the next layer.

A much more widely accepted way to do this is to modify the *stride* of the convolution operation. The stride is the amount that the filter is incrementally slid over the image—in the previous case, we are using a stride of 1, and as a result each filter is convolved with every element of the input, which is why the output ends up being the same size as the input. With a stride of 2, the filter would be convolved with *every other* element of the input image, so that the output would be half the size of the input; with a stride of 3, the filter would be convolved with *every third* element of the input image, and so on. This means that, for example, using a stride of 2 would result in the same output size and thus much the same reduction in computation we would get from pooling with size 2, but without as much *loss of information*: with pooling of size 2, only one-fourth of the elements in the input have *any* effect on the output, whereas with a stride of 2, *every* element of the input has *some* effect on the output. The use of a stride of greater than 1 is thus significantly more prevalent than pooling for downsampling even in the most advanced CNN architectures of today.

Nevertheless, in this book I’ll just show examples with a stride of 1—modifying these operations to allow a stride of greater than 1 is left as an exercise for the reader. Using a stride equal to 1 also makes writing the backward pass easier.

Convolutions: The Backward Pass

The backward pass is where convolutions get a bit trickier. Let’s recall what we’re trying to do: before, we produced the output of a convolution operation using the input and the parameters. We now want to compute:

- The partial derivative of the loss with respect to each element of the *input* to the convolution operation—`inp` previously
- The partial derivative of the loss with respect to each element of the *filter*—`param_1d` previously

Think of how the `ParamOperation`s we saw in Chapter 4 work: in the `backward` method, they receive an output gradient representing how much each element of the output ultimately affects the loss and then use this output gradient to compute the gradients for the input and the parameters. So we need to write a function that takes in an `output_grad` with the same shape as the input and

How can we test whether the computed gradients are correct? We'll bring back an idea from the first chapter: we know that the partial derivative of a sum with respect to any one of its inputs is 1 (if the sum $s = a + b + c$, then $\frac{\partial s}{\partial a} = \frac{\partial s}{\partial b} = \frac{\partial s}{\partial c} = 1$). So we can compute the `input_grad` and `param_grad` quantities using our `_input_grad` and `_param_grad` functions (which we'll reason through and write shortly) and an `output_grad` equal to all 1s. Then we'll check whether these gradients are correct by changing elements of the input by some quantity α and seeing whether the resulting sum changes by the gradient times α.

WHAT "SHOULD" THE GRADIENT BE?

Using the logic just described, let's calculate what an element of the gradient vector for the input *should* be:

```
def conv_1d_sum(inp: ndarray,
                param: ndarray) -> ndarray:
    out = conv_1d(inp, param)
    return np.sum(out)
```

```
# randomly choose to increase 5th element by 1
input_1d_2 = np.array([1,2,3,4,6])
param_1d = np.array([1,1,1])

print(conv_1d_sum(input_1d, param_1d))
print(conv_1d_sum(input_1d_2, param_1d))
```

```
39.0
41.0
```

So, the gradient of the fifth element of the input *should* be $41 - 39 = 2$.

Now let's try to reason through how we should compute such a gradient without simply computing the difference between these two sums. Here is where things get interesting.

COMPUTING THE GRADIENT OF A 1D CONVOLUTION

We see that increasing this element of the input increased the output by 2. Taking a close look at the output shows exactly how it does this:

$$
\begin{aligned}
\text{Output:[}\quad & t_0w_1+t_1w_2+t_2w_3,=O_1\\
& t_1w_1+t_2w_2+t_3w_3,=O_2\\
& t_2w_1+t_3w_2+t_4w_3,=O_3\\
& t_3w_1+t_4w_2+t_5w_3,=O_4\\
& t_4w_1+t_5w_2+t_6w_3]=O_5
\end{aligned}
$$

This particular element of the input is denoted t_5. It appears in the output in two places:

- As part of o_4, it is multiplied by w_3.
- As part of o_5, it is multiplied by w_2.

multiplied by w_1.

Therefore, the amount that t_5 ultimately affects the loss, which we can denote as $\frac{\partial L}{\partial t_5}$, will be:

$$\frac{\partial L}{\partial t_5} = \frac{\partial L}{\partial o_4} \times w_3 + \frac{\partial L}{\partial o_5} \times w_2 + \frac{\partial L}{\partial o_6} \times w_1$$

Of course, in this simple example, when the loss is just the sum, $\frac{\partial L}{\partial o_i} = 1$ for all elements, in the output (except for the "padding" elements for which this quantity is 0). This sum is very easy to compute: it is simply $w_2 + w_3$, which is indeed 2 since $w_2 = w_3 = 1$.

WHAT'S THE GENERAL PATTERN?

Now let's look for the general pattern for a generic input element. This turns out to be an exercise in keeping track of indices. Since we're translating math into code here, let's use o_i^{grad} to denote the ith element of the output gradient (since we'll ultimately be accessing it via `output_grad[i]`). Then:

$$\frac{\partial L}{\partial t_5} = o_4^{grad} \times w_3 + o_5^{grad} \times w_2 + o_6^{grad} \times w_1$$

Looking closely at this output, we can reason similarly that:

$$\frac{\partial L}{\partial t_3} = o_2^{grad} \times w_3 + o_3^{grad} \times w_2 + o_4^{grad} \times w_1$$

and:

$$\frac{\partial L}{\partial t_4} = o_3^{grad} \times w_3 + o_4^{grad} \times w_2 + o_5^{grad} \times w_1$$

There's clearly a pattern here, and translating it into code is a bit tricky, especially since the indices on the output increase at the same time the indices on the weights decrease. Nevertheless, the way to express this turns out to be via the following double `for` loop:

```
# param: in our case an ndarray of shape (1,3)
# param_len: the integer 3
# inp: in our case an ndarray of shape (1,5)
# input_grad: always an ndarray the same shape as "inp"
# output_pad: in our case an ndarray of shape (1,7)
for o in range(inp.shape[0]):
    for p in range(param.shape[0]):
        input_grad[o] += output_pad[o+param_len-p-1] * param[p]
```

This does the appropriate incrementing of the indices of the weights, while decreasing the weights on the output at the same time.

Though it may not be obvious now, reasoning through this and getting it is out to be the trickiest part of calculating the gradients for convolution operations. Adding more complexity to this, such as batch sizes, convolutions with two-dimensional inputs, or inputs with multiple channels, is simply a matter of adding more `for` loops to the preceding lines, as we'll see in the next few sections.

COMPUTING THE PARAMETER GRADIENT

We can reason similarly about how increasing an element of the filter should increase the output. First, let's increase (arbitrarily) the first element of the filter by one unit and observe the resulting impact on the sum:

```
input_1d = np.array([1,2,3,4,5])
# randomly choose to increase first element by 1
param_1d_2 = np.array([2,1,1])

print(conv_1d_sum(input_1d, param_1d))
print(conv_1d_sum(input_1d, param_1d_2))
```

```
39.0
49.0
```

So we should find that $\frac{\partial L}{\partial w_1} = 10$.

Just as we did for the input, by closely examining the output and seeing which elements of the filter affect it, as well as padding the input to more clearly see the pattern, we see that:

$$w_1^{grad} = t_0 \times o_1^{grad} + t_1 \times o_2^{grad} + t_2 \times o_3^{grad} + t_3 \times o_4^{grad} + t_4 \times o_5^{grad}$$

And since, for the sum, all of the o_i^{grad} elements are just 1, and t_0 is 0, we have:

$$w_1^{grad} = t_1 + t_2 + t_3 + t_4 = 1 + 2 + 3 + 4 = 10$$

This confirms the calculation from earlier.

CODING THIS UP

Coding this up turns out to be easier than writing the code for the input gradient, since this time "the indices are moving in the same direction." Within the same nested `for` loop, the code is:

```
# param: in our case an ndarray of shape (1,3)
# param_grad: an ndarray the same shape as param
# inp: in our case an ndarray of shape (1,5)
# input_pad: an ndarray the same shape as (1,7)
# output_grad: in our case an ndarray of shape (1,5)
for o in range(inp.shape[0]):
    for p in range(param.shape[0]):
        param_grad[p] += input_pad[o+p] * output_grad[o]
```

Finally, we can combine these two computations and write a function to compute both the input gradient and the filter gradient with the following steps:

1. Take the input and filter as arguments.
2. Compute the output.
3. Pad the input and the output gradient (to get, say, `input_pad` and `output_pad`).
4. As shown earlier, use the padded output gradient and the filter to compute the gradient.
5. Similarly, use the output gradient (not padded) and the padded input to compute the filter gradient.

I show the full function that wraps around the preceding code blocks in the book's GitHub repo.

That concludes our explanation of how to implement convolutions in 1D! As we'll see in the next several sections, extending this reasoning to work on two-dimensional inputs, batches of two-dimensional inputs, or even multichannel batches of two-dimensional inputs is (perhaps surprisingly) straightforward.

Batches, 2D Convolutions, and Multiple Channels

First, let's add the capability for these convolution functions to work with *batches* of inputs—2D inputs whose first dimension represents the batch size of the input and whose second dimension represents the length of the 1D sequence:

```
input_1d_batch = np.array([[0,1,2,3,4,5,6],
                           [1,2,3,4,5,6,7]])
```

the input and filter gradients.

1D CONVOLUTIONS WITH BATCHES: FORWARD PASS

The only difference in implementing the forward pass when the input has a second dimension representing the batch size is that we have to pad and compute the output for each observation individually (as we did previously) and then `stack` the results to get a batch of outputs. For example, `conv_1d` becomes:

```
def conv_1d_batch(inp: ndarray,
                  param: ndarray) -> ndarray:

    outs = [conv_1d(obs, param) for obs in inp]
    return np.stack(outs)
```

1D CONVOLUTIONS WITH BATCHES: BACKWARD PASS

The backward pass is similar: computing the input gradient now simply takes the `for` loop for computing the input gradient from the prior section, computes it for each observation, and `stack`s the results:

```
# "_input_grad" is the function containing the for loop from earlier:
# it takes in a 1d input, a 1d filter, and a 1d output_gradient and com
# the input grad
grads = [_input_grad(inp[i], param, out_grad[i])[1] for i in range(ba
np.stack(grads)
```

The gradient for the filter when dealing with a batch of observations is a bit different. This is because the filter is convolved with every observation in the input and is thus connected to every observation in the output. So, to compute the parameter gradient, we have to loop through all of the observations and increment the appropriate values of the parameter gradient as we do so. Still, this just involves adding an outer `for` loop to the code to compute the parameter gradient that we saw earlier:

```
# param: in our case an ndarray of shape (1,3)
# param_grad: an ndarray the same shape as param
# inp: in our case an ndarray of shape (1,5)
# input_pad: an ndarray the same shape as (1,7)
# output_grad: in our case an ndarray of shape (1,5)
for i in range(inp.shape[0]): # inp.shape[0] = 2
    for o in range(inp.shape[1]): # inp.shape[0] = 5
        for p in range(param.shape[0]): # param.shape[0] = 3
            param_grad[p] += input_pad[i][o+p] * output_grad[i][o]
```

Adding this dimension on top of the original 1D convolution was indeed simple; extending this from one- to two-dimensional inputs is similarly straightforward.

2D Convolutions

The 2D convolution is a straightforward extension of the 1D case because, fundamentally, the way the input is connected to the output via the filters in each dimension of the 2D case is identical to the 1D case. As a result, the high-level steps on both the forward and backward passes remain the same:

1. On the forward pass, we:

    - Appropriately pad the input.

    - Use the padded input and the parameters to compute the output.

2. On the backward pass, to compute the input gradient we:

- Use this padded output gradient, along with the input and the parameters, to compute both the input gradient and the parameter gradient.

3. Also on the backward pass, to compute the parameter gradient we:

    - Appropriately pad the input.

    - Loop through the elements of the padded input and increment the parameter gradient appropriately as we go along.

2D CONVOLUTIONS: CODING THE FORWARD PASS

To make this concrete, recall that for 1D convolutions the code for computing the output given the input and the parameters on the forward pass looked as follows:

```
# input_pad: a version of the input that has been padded appropriately
# the size of param

out = np.zeros_like(inp)

for o in range(out.shape[0]):
    for p in range(param_len):
        out[o] += param[p] * input_pad[o+p]
```

For 2D convolutions, we simply modify this to be:

```
# input_pad: a version of the input that has been padded appropriately
# the size of param

out = np.zeros_like(inp)

for o_w in range(img_size): # Loop through the image height
    for o_h in range(img_size): # Loop through the image width
        for p_w in range(param_size): # Loop through the parameter wi
            for p_h in range(param_size): # Loop through the paramete
                out[o_w][o_h] += param[p_w][p_h] * input_pad[o_w+p_w
```

You can see that we've simply "blown each `for` loop out" into two `for` loops.

The extension to two dimensions when we have a batch of images is also similar to the 1D case: just as we did there, we simply add a `for` loop to the outside of the loops shown here.

2D CONVOLUTIONS: CODING THE BACKWARD PASS

Sure enough, just as in the forward pass, we can use the same indexing for the backward pass as in the 1D case. Recall that in the 1D case, the code was:

```
input_grad = np.zeros_like(inp)

for o in range(inp.shape[0]):
    for p in range(param_len):
        input_grad[o] += output_pad[o+param_len-p-1] * param[p]
```

In the 2D case, the code is simply:

```
# output_pad: a version of the output that has been padded appropriatel
# on the size of param
input_grad = np.zeros_like(inp)

for i_w in range(img_width):
    for i_h in range(img_height):
        for p_w in range(param_size):
            for p_h in range(param_size):
```

Note that the indexing on the output is the same as in the 1D case but is simply taking place in two dimensions; in the 1D case, we had:

```
output_pad[i+param_size-p-1] * param[p]
```

and in the 2D case, we have:

```
output_pad[i_w+param_size-p_w-1][i_h+param_size-p_h-1] * param[p_w][p_
```

The other facts from the 1D case also apply:

- For a batch of input images, we simply perform the preceding operation for each observation and then `stack` the results.
- For the parameter gradient, we have to loop through all the images in the batch and add components from each one to the appropriate places in the parameter gradient:[8]

```
# input_pad: a version of the input that has been padded appropriately
# the size of param

param_grad = np.zeros_like(param)

for i in range(batch_size): # equal to inp.shape[0]
    for o_w in range(img_size):
        for o_h in range(img_size):
            for p_w in range(param_size):
                for p_h in range(param_size):
                    param_grad[p_w][p_h] += input_pad[i][o_w+p_w][o_
                    * output_grad[i][o_w][o_h]
```

At this point, we've almost written the code for the complete multichannel convolution operation; currently, our code convolves filters over a two-dimensional input and produces a two-dimensional output. Of course, as we described earlier, each convolutional layer not only has neurons arranged along these two dimensions but also has some number of "channels" equal to the number of feature maps that the layer creates. Addressing this last challenge is what we'll cover next.

The Last Element: Adding "Channels"

How can we modify what we've written thus far to account for cases where both the input and the output are multichannel? The answer, as it was when we added batches earlier, is simple: we add two outer `for` loops to the code we've already seen—one loop for the input channels and another for the output channels. By looping through all combinations of the input channel and the output channel, we make each output feature map a combination of all of the input feature maps, as desired.

For this to work, we will have to *always* represent our images as three-dimensional `ndarray`s, as opposed to the two-dimensional arrays we've been using; we'll represent black-and-white images with one channel and color images with three channels (one for the red values at each location in the image, one for the blue values, and one for the green values). Then, regardless of the number of channels, the operation proceeds as described earlier, with a number of feature maps being created from the image, each of which is a combination of the convolutions resulting from all of the channels in the image (or from the channels in the prior layer, if dealing with layers further on in the network).

Given all this, the full code to compute the output for a convolutional layer, given four-dimensional `ndarray`s for the input and the parameters, is:

```
def _compute_output_obs(obs: ndarray,
                        param: ndarray) -> ndarray:
    '''
    obs: [channels, img_width, img_height]
    param: [in_channels, out_channels, param_width, param_height]
    '''
    assert_dim(obs, 3)
    assert_dim(param, 4)

    param_size = param.shape[2]
    param_mid = param_size // 2
    obs_pad = _pad_2d_channel(obs, param_mid)

    in_channels = fil.shape[0]
    out_channels = fil.shape[1]
    img_size = obs.shape[1]

    out = np.zeros((out_channels,) + obs.shape[1:])
    for c_in in range(in_channels):
        for c_out in range(out_channels):
            for o_w in range(img_size):
                for o_h in range(img_size):
                    for p_w in range(param_size):
                        for p_h in range(param_size):
                            out[c_out][o_w][o_h] += \
                              param[c_in][c_out][p_w][p_h]
                              * obs_pad[c_in][o_w+p_w][o_h+p_h]
    return out

def _output(inp: ndarray,
            param: ndarray) -> ndarray:
    '''
    obs: [batch_size, channels, img_width, img_height]
    param: [in_channels, out_channels, param_width, param_height]
    '''
    outs = [_compute_output_obs(obs, param) for obs in inp]

    return np.stack(outs)
```

Note that `_pad_2d_channel` is a function that pads the input along the channel dimension.

Again, the actual code that does the computation is similar to the code in the simpler 2D case (without channels) shown before, except now we have, for example, `fil[c_out][c_in][p_w][p_h]` instead of just `fil[p_w][p_h]`, since there are two more dimensions and `c_out × c_in` more elements in the filter array.

BACKWARD PASS

The backward pass is similar and follows the same conceptual principles as the backward pass in the simple 2D case:

1. For the input gradients, we compute the gradients of each observation individually—padding the output gradient to do so—and then `stack` the gradients.

2. We also use the padded output gradient for the parameter gradient, but we loop through the observations as well and use the appropriate values from each one to update the parameter gradient.

Here's the code for computing the output gradient:

```
def _compute_grads_obs(input_obs: ndarray,
                       output_grad_obs: ndarray,
                       param: ndarray) -> ndarray:
    '''
    input_obs: [in_channels, img_width, img_height]
    output_grad_obs: [out_channels, img_width, img_height]
    img_size = input_obs.shape[1]
    in_channels = input_obs.shape[0]
    out_channels = param.shape[1]
    output_obs_pad = _pad_2d_channel(output_grad_obs, param_mid)

    for c_in in range(in_channels):
        for c_out in range(out_channels):
            for i_w in range(input_obs.shape[1]):
                for i_h in range(input_obs.shape[2]):
                    for p_w in range(param_size):
                        for p_h in range(param_size):
                            input_grad[c_in][i_w][i_h] += \
                            output_obs_pad[c_out][i_w+param_size-p_w-1][i_h+par
                            * param[c_in][c_out][p_w][p_h]
    return input_grad

def _input_grad(inp: ndarray,
                output_grad: ndarray,
                param: ndarray) -> ndarray:

    grads = [_compute_grads_obs(inp[i], output_grad[i], param) for i in range(ou

    return np.stack(grads)
```

And here's the parameter gradient:

```
def _param_grad(inp: ndarray,
                output_grad: ndarray,
                param: ndarray) -> ndarray:
    '''
    inp: [in_channels, img_width, img_height]
    output_grad_obs: [out_channels, img_width, img_height]
    param: [in_channels, out_channels, img_width, img_height]
    '''
    param_grad = np.zeros_like(param)
    param_size = param.shape[2]
    param_mid = param_size // 2
    img_size = inp.shape[2]
    in_channels = inp.shape[1]
    out_channels = output_grad.shape[1]

    inp_pad = _pad_conv_input(inp, param_mid)
    img_shape = output_grad.shape[2:]

    for i in range(inp.shape[0]):
        for c_in in range(in_channels):
            for c_out in range(out_channels):
                for o_w in range(img_shape[0]):
                    for o_h in range(img_shape[1]):
                        for p_w in range(param_size):
                            for p_h in range(param_size):
                                param_grad[c_in][c_out][p_w][p_h] +
                                inp_pad[i][c_in][o_w+p_w][o_h+p_h]
                                * output_grad[i][c_out][o_w][o_h]
    return param_grad
```

These three functions—`_output`, `_input_grad`, and `_param_grad`—are just what we need to create a `Conv2DOperation`, which will ultimately form the core of the `Conv2DLayer`s we'll use in our CNNs! There are just a few more details to work out before we can use this `Operation` in a working convolutional neural network.

Using This Operation to Train a CNN

We need to implement a few more pieces before we can have a working CNN model:

1. We have to implement the `Flatten` operation discussed earlier in the chapter; this is necessary to enable the model to make predictions.

2. We have to incorporate this `Operation` as well as the `Conv2DOpOperation` into a `Conv2D Layer`.

3. Finally, for any of this to be usable, we have to write a faster version of the `Conv2D Operation`. We'll outline this here and share the details in

The Flatten Operation

There's one other `Operation` we'll need to complete our convolutional layer: the `Flatten` operation. The output of a convolution operation is a 3D `ndarray` for each observation, of dimension `(channels, img_height, img_width)`. However, unless we are passing this data into another convolutional layer, we'll first need to transform it into a *vector* for each observation. Luckily, as described previously, since each of the individual neurons involved encodes whether a particular visual feature is present at that location in the image, we can simply "flatten" this 3D `ndarray` into a 1D vector and pass it forward without any problem. The `Flatten` operation shown here does this, accounting for the fact that in convolutional layers, as with any other layer, the first dimension of our `ndarray` is always the batch size:

```
class Flatten(Operation):
    def __init__(self):
        super().__init__()

    def _output(self) -> ndarray:
        return self.input.reshape(self.input.shape[0], -1)

    def _input_grad(self, output_grad: ndarray) -> ndarray:
        return output_grad.reshape(self.input.shape)
```

That's the last `Operation` we'll need; let's wrap these `Operation`s up in a `Layer`.

The Full Conv2D Layer

The full convolutional layer, then, would look something like this:

```
class Conv2D(Layer):

    def __init__(self,
                 out_channels: int,
                 param_size: int,
                 activation: Operation = Sigmoid(),
                 flatten: bool = False) -> None:
        super().__init__()
        self.out_channels = out_channels
        self.param_size = param_size
        self.activation = activation
        self.flatten = flatten

def _setup_layer(self, input_: ndarray) -> ndarray:

    self.params = []
    conv_param = np.random.randn(self.out_channels,
                                 input_.shape[1],  # input channels
                                 self.param_size,
                                 self.param_size)
    self.params.append(conv_param)

    self.operations = []
    self.operations.append(Conv2D(conv_param))
    self.operations.append(self.activation)

    if self.flatten:
        self.operations.append(Flatten())

    return None
```

The `Flatten` operation is optionally added on at the end, depending on whether we want the output of this layer to be passed forward into another convolutional layer or passed into another fully connected layer for predictions.

A NOTE ON SPEED, AND AN ALTERNATIVE IMPLEMENTATION

purpose of writing the convolution operation from scratch was to solidify our understanding of how CNNs work. Still, it is possible to write convolutions in a completely different way; instead of breaking down that process like we have in this chapter, we can break it down into the following steps:

1. From the input, extract `image_height × image_width × num_channels` patches of size `filter_height × filter_width` from the test set.
2. For each of these patches, perform a dot product of the patch with the appropriate filter connecting the input channels to the output channels.
3. Stack and reshape the results of all of these dot products to form the output.

With a bit of cleverness, we can express almost all of the operations described previously in terms of a batch matrix multiplication, implemented using NumPy's `np.matmul` function. The details of how to do this are described in Appendix A and are implemented on the book's website, but suffice it to say that this allows us to write relatively small convolutional neural networks that can train in a reasonable amount of time. This lets us actually run experiments to see how well convolutional neural networks work!

Experiments

Even using the convolution operation defined by reshaping and the `matmul` functions, it takes about 10 minutes to train this model for one epoch with just one convolutional layer, so we restrict ourselves to demonstrating a model with just one convolutional layer, with 32 channels (a number chosen somewhat arbitrarily):

```
model = NeuralNetwork(
    layers=[Conv2D(out_channels=32,
                   param_size=5,
                   dropout=0.8,
                   weight_init="glorot",
                   flatten=True,
                  activation=Tanh()),
            Dense(neurons=10,
                  activation=Linear())],
            loss = SoftmaxCrossEntropy(),
seed=20190402)
```

Note that this model has just 32 × 5 × 5 = 800 parameters in the first layer, but these parameters are used to create 32 × 28 × 28 = 25,088 neurons, or "learned features." By contrast, a fully connected layer with hidden size 32 would have 784 × 32 = 25,088 *parameters*, and just 32 neurons.

Some simple trial and error—training this model for just a few hundred batches with different learning rates and observing the resulting validation losses—shows that a learning rate of 0.01 works better than a learning rate of 0.1 now that we have a convolutional layer as our first layer, rather than a fully connected layer. Training this network for one epoch with optimizer `SGDMomentum(lr = 0.01, momentum=0.9)` gives:

```
Validation accuracy after 100 batches is 79.65%
Validation accuracy after 200 batches is 86.25%
Validation accuracy after 300 batches is 85.47%
Validation accuracy after 400 batches is 87.27%
Validation accuracy after 500 batches is 88.93%
Validation accuracy after 600 batches is 88.25%
Validation accuracy after 700 batches is 89.91%
Validation accuracy after 800 batches is 89.59%
Validation accuracy after 900 batches is 89.96%
Validation loss after 1 epochs is 3.453

Model validation accuracy after 1 epoch is 90.50%
```

the training set![9]

Conclusion

In this chapter, you've learned about convolutional neural networks. You started out learning at a high level what they are and about their similarities and differences from fully connected neural networks, and then you went all the way down to seeing how they work at the lowest level, implementing the core multichannel convolution operation from scratch in Python.

Starting at a high level, convolutional layers create roughly an order of magnitude more neurons than the fully connected layers we've seen so far, with each neuron being a combination of just a few features from the prior layer, rather than each neuron being a combination of *all* of the features from the prior layer as they are in fully connected layers. A level below that, we saw that these neurons are in fact grouped into "feature maps," each of which represents whether a particular visual feature—or a particular combination of visual features, in the case of deep convolutional neural networks—is present at a given location in an image. Collectively we refer to these feature maps as the convolutional `Layer`'s "channels."

Despite all these differences from the `Operation`s we saw involved with the `Dense` layer, the convolution operation fits into the same template as other `ParamOperation`s we've seen:

- It has an `_output` method that computes the output given its input and the parameter.
- It has `_input_grad` and `_param_grad` methods that, given an `output_grad` of the same shape as the `Operation`'s `output`, compute gradients of the same shape as the input and parameters, respectively.

The difference is just that the `_input`, `output`, and `param`s are now four-dimensional `ndarray`s, whereas they were two-dimensional in the case of fully connected layers.

This knowledge should serve as an extremely solid foundation for any future learning about or application of the convolutional neural networks you undertake. Next, we'll cover another common kind of advanced neural network architecture: recurrent neural networks, designed for dealing with data that appears in sequences, rather than simply the nonsequential batches we've dealt with in the cases of houses and images. Onward!

[1] The code we'll write, while clearly expressing how convolutions work, will be extremely inefficient. In "Gradient of the Loss with Respect to the Bias Terms", I provide a more efficient implementation of the batch, multichannel convolution operation we'll describe in this chapter using NumPy.

[2] See the Wikipedia page for "Kernel (image processing)" for more examples.

[3] These are also referred to as *kernels*.

[4] This is why it is important to understand the output of convolution operations both as creating a number of filter maps—say, m—as well as creating $m \times image_{height} \times image_{width}$ individual neurons. As is the case throughout neural networks, holding multiple levels of interpretation in one's mind all at one time, and seeing the connection between them, is key.

[5] See the original ResNet paper (http://tiny.cc/dlfs_resnet_paper), "Deep Residual Learning for Image Recognition," by Kaiming He et al.

[7] A year later, DeepMind published results using a similar representation with chess—only this time, to encode the more complex ruleset of chess, the input had 119 channels! See DeepMind (David Silver et al.), “A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play”.

[8] See the full implementations of these on the book’s website.

[9] The full code can be found in the section for this chapter on the book’s GitHub repo.

Support / Sign Out

PREV
4. Extensions

NEXT
6. Recurrent Neural Networks