12
Convolutional Neural Network

In Chapter 11, we focused on a type of a neural network called feed-forward neural networks. But different data has different structure (*e.g.*, image, text, audio, etc.) and we need better ways of exploiting them. This can help reduce the number of parameters needed in the network, which may allow easier or more data-efficient training. In this chapter, we present a type of a neural network common in image processing called *Convolutional Neural Network* (CNN); these models use a mathematical technique called convolution in order to extract important visual features from input images.

12.1 Introduction to Convolution

Roughly speaking, *convolution* refers to a mathematical operation where two functions are "mixed" to output a new function. In machine learning, the main idea of convolution is to reuse the same set of parameters on different portions of input. This is particularly effective at exploiting the structure of images. It was originally motivated by studies of the structure of cortical cells in the $V1$ visual cortex of the human brain (Hubel and Wiesel won the Nobel Prize in 1981 for this breakthrough discovery). [1] Let's first consider an example of a $1D$ convolution.

[1] Paper: https://www.jstor.org/stable/24965293.

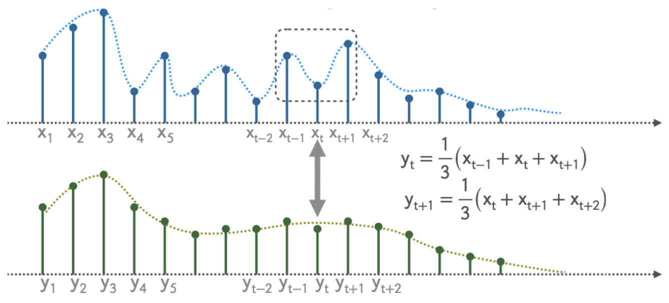

x_1 x_2 x_3 x_4 x_5 x_{t-2} x_{t-1} x_t x_{t+1} x_{t+2}
$y_t = \frac{1}{3}(x_{t-1} + x_t + x_{t+1})$
$y_{t+1} = \frac{1}{3}(x_t + x_{t+1} + x_{t+2})$
y_1 y_2 y_3 y_4 y_5 y_{t-2} y_{t-1} y_t y_{t+1} y_{t+2}

Figure 12.1: The effects of $1D$ convolution on graph of COVID-19 positive cases.

Example 12.1.1. *Consider the 3-day moving average of daily COVID-19 cases as shown in Figure 12.1. Let x_t denote the number of daily cases on day t. We can then take three consecutive values, compute their average and create a new output sequence from averages: $y_t = \frac{1}{3}(x_{t-1} + x_t + x_{t+1})$. Then if we set $w_1 = w_2 = w_3 = \frac{1}{3}$ and denote $\vec{\mathbf{w}} = (w_1, w_2, w_3)$, we can write:* [2]

$$\begin{aligned} y_t &= w_1 x_{t-1} + w_2 x_t + w_3 x_{t+1} = \vec{\mathbf{w}} \cdot (x_{t-1}, x_t, x_{t+1}) \\ y_{t+1} &= w_1 x_t + w_2 x_{t+1} + w_3 x_{t+2} = \vec{\mathbf{w}} \cdot (x_t, x_{t+1}, x_{t+2}) \\ y_{t+2} &= w_1 x_{t+1} + w_2 x_{t+2} + w_3 x_{t+3} = \vec{\mathbf{w}} \cdot (x_{t+1}, x_{t+2}, x_{t+3}) \end{aligned}$$

[2] If we set the weights to a different value, we can find a weighted moving average.

Notice that we are reusing the same weights and applying them to multiple different values of x_t to calculate y_t. It is almost like sliding a filter down the array of x_t's and applying it to every set of 3 *consecutive inputs. For this reason, we call $\vec{\mathbf{w}}$ the **convolution filter weight** of length* 3.

Example 12.1.2. *Consider an input sequence $\vec{\mathbf{x}} = (2, 1, 1, 7, -1, 2, 3, 1)$ and a convolution filter $\vec{\mathbf{w}} = (3, 2, -1)$. The first two output values will be:*

$$\begin{aligned} y_1 &= 2 \times 3 + 1 \times 2 + 1 \times (-1) = 7 \\ y_2 &= 1 \times 3 + 1 \times 2 + 7 \times (-1) = -2 \end{aligned}$$

Following a similar calculation for the other values, we see that the full output sequence is $\vec{\mathbf{y}} = (7, -2, 18, 17, -2, 11)$. Note that the length of $\vec{\mathbf{y}}$ should be $|\vec{\mathbf{x}}| - |\vec{\mathbf{w}}| + 1 = 8 - 3 + 1 = 6$.

Problem 12.1.3. *If $y_t = 2x_{t-1} - x_{t+1}$, $y_{t+1} = 2x_t - x_{t+2}$, and $y_{t+2} = 2x_{t+1} - x_{t+3}$, what is the convolution filter weight?*

12.2 Convolution in Computer Vision

In this section, we now focus on the application of convolution in computer vision. By the nature of image data, we will be primarily dealing with 2*D* convolution. Generally, 2D convolution filters are called *kernels*.

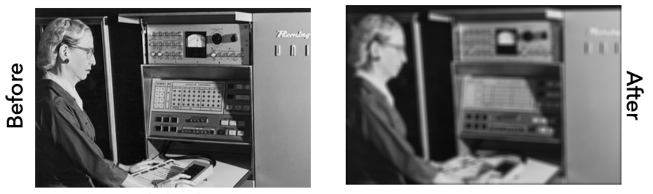


Figure 12.2: The effect of local smoothing on a sample image. (The person depicted here is Admiral Grace Murray Hopper, a computing pioneer.)

Example 12.2.1 (Local Smoothing (Blurring))**.** *An image can be blurred by constructing a filter that replaces each pixel by the average of neighboring pixels:*

$$y_{i,j} = \frac{1}{9} \sum_{b_1, b_2 \in \{-1,0,1\}} x_{i+b_1, j+b_2}$$

An example is shown in Figure 12.2.

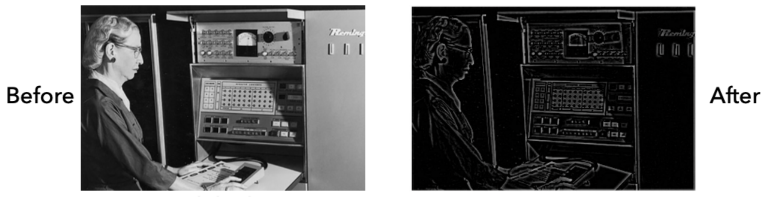


Figure 12.3: The effect of local sharpening on a sample image

Example 12.2.2 (Local Sharpening (Edge Detection)). *The edge of objects in an image can be detected by constructing a filter that replaces each pixel by its difference with the average of neighboring pixels:*

$$y_{i,j} = x_{i,j} - \frac{1}{9} \sum_{b_1, b_2 \in \{-1,0,1\}} x_{i+b_1, j+b_2}$$

An example is shown in Figure 12.3.

12.2.1 Convolution Filters for Images

Computationally, we perform 2D convolution on an image by "sliding" the filter around every possible location in the image and taking the inner product:

$$y_{i,j} = \sum_{-k \le r,s \le k} w_{r,s} x_{i+r,j+s} \tag{12.1}$$

The result is a new image and we can view each filter as a transformation which takes an image and returns an image. In the above equation, the filter size is $(2k+1) \times (2k+1)$. For example, if $k = 1$, we can consider the convolution weight filter to be

$$\begin{bmatrix} w_{-1,-1} & w_{-1,0} & w_{-1,+1} \\ w_{0,-1} & w_{0,0} & w_{0,+1} \\ w_{+1,-1} & w_{+1,0} & w_{+1,+1} \end{bmatrix}$$

The filter can only be applied to an image of size $m \times n$ at a location where the filter completely fits inside the image. Therefore, the locations in the input image where the center of the filter can be placed are $k < i \le m - k, \quad k < j \le n - k$ and the size of the output image is $(m - 2k) \times (n - 2k)$.

Example 12.2.3. *If the input and convolution filter are given as follows:*

$$\mathbf{X} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \textit{and} \quad \mathbf{W} = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

then the pixel at $(1,1)$ *of the resulting image can be calculated by applying the filter at the top left corner of the input image. That is, we take the inner product of the following parts (in red) of the two matrices*

$$\begin{bmatrix} 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix} \quad \textit{and} \quad \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

which is

$$1 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 1 \times 0 + 0 \times 1 + 0 \times 0 + 1 \times 1 = 4$$

Therefore, the $(1,1)$ *entry of the resulting image is* 4*. Similarly, the remaining pixels of the resulting image can be calculated by moving around the filter as in Figure 12.4. The output image is given as:*

$$\mathbf{Y} = \begin{bmatrix} 4 & 3 & 4 \\ 2 & 4 & 3 \\ 2 & 3 & 4 \end{bmatrix}$$

In this example, $\mathbf{X} \in \mathbb{R}^{5\times5}, \mathbf{W} \in \mathbb{R}^{3\times3}, k = 1$ *and* $\mathbf{Y} \in \mathbb{R}^{3\times3}$.

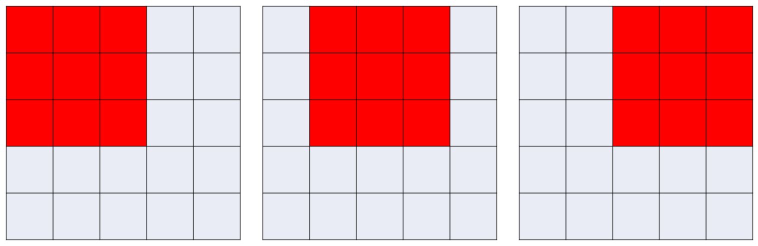


Figure 12.4: Visual representation of applying a 3×3 convolutional filter to a 5×5 image.

Problem 12.2.4. *Suppose we have a* 10×10 *image and a* 5×5 *filter. What is the size of the output image?*

Figure 12.5 shows some common filters used in image processing. Note that all these filters are hand-crafted and require domain-specific knowledge. However, in convolutional neural networks, we don't set these weights by hand and we learn all the filter weights from the data!

12.2.2 *Padding*

In standard $2D$ convolution, the size of the output image is not equal to the size of the input image because we only consider locations where the filter fits completely in the image. However, sometimes we may want their sizes to be the same. In such a case, we apply a

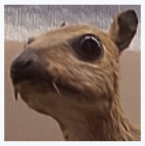
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix}1&1&1\\1&1&1\\1&1&1\end{bmatrix}$	
Gaussian blur 3 × 3 (approximation)	$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4&2\\1&2&1\end{bmatrix}$	
Gaussian blur 5 × 5 (approximation)	$\frac{1}{256}\begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&36&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}$	
Unsharp masking 5 × 5 Based on Gaussian blur with amount as 1 and threshold as 0 (with no image mask)	$\frac{-1}{256}\begin{bmatrix}1&4&6&4&1\\4&16&24&16&4\\6&24&-476&24&6\\4&16&24&16&4\\1&4&6&4&1\end{bmatrix}$	

Figure 12.5: Some common filters and corresponding weights used in image processing. Source: `https://en.wikipedia.org/wiki/Kernel_(image_processing)`

technique called *padding*. The idea is to pad pixels to all four edges of the input image (left, right, up, and down) so that the number of valid locations for the filter is the same as the number of pixels in the original image. In particular, if the filter size is $(2k+1) \times (2k+1)$, we need to pad k pixels on each side.

There are multiple ways to implement padding. *Zero padding* is when the values at all padded pixels are set to 0. *"Same" padding* is when the values at padded pixels are set to the value of the nearest pixel at the edge of the input image. In practice, zero padding is a more common form of padding (it is equivalent to adding a black frame around the image).

12.2.3 *Downsampling Input with Stride*

Another common operation in convolutional neural networks is called *stride*. Stride controls how the filter convolves around the input image. Instead of moving the filter by 1 pixel every time, we can also move the filter every 2 (or in general, s) pixels. Essentially, we are applying each of the filter weights at fewer locations of the image than before. This can be viewed as a downsampling strategy, which gives a smaller output image while greatly preserving the information from the original input.

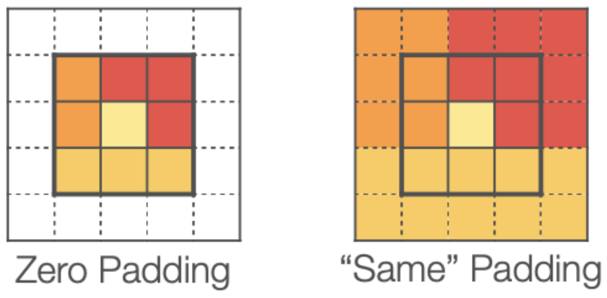


Figure 12.6: A visual comparison between two common types of padding: zero padding and "same" padding

Suppose we have an input image of size $m \times n$ and a filter of size $(2k+1) \times (2k+1)$. If padding is applied to the image, the output image size is $\lfloor (m+s-1)/s \rfloor \times \lfloor (n+s-1)/s \rfloor$. [3] If padding is not applied to the image, the output image size is $\lfloor (m+s-2k-1)/s \rfloor \times \lfloor (n+s-2k-1)/s \rfloor$; this is because convolution with stride is performed directly on the input image itself, making the effective input image size $(m-2k) \times (n-2k)$.

[3] As a sanity check, you can verify that in the special case of $s = 1$ the output image size will be the same as the input image size

Example 12.2.5. *Suppose we have an input image of size* 5×5 *and a filter of size* 3×3*. If we apply padding and take stride size* $s = 2$*, then output size is* 3×3.

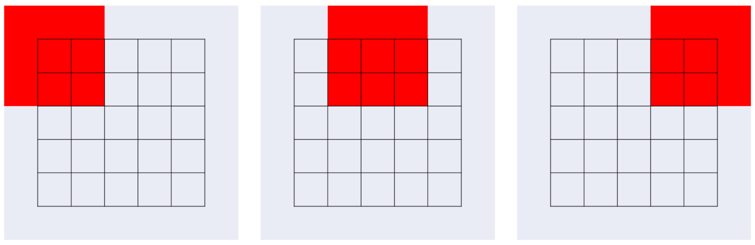


Figure 12.7: Visual representation of applying a 3×3 convolutional filter to a 5×5 image with padding and stride size 2.

12.2.4 *Nonlinear Convolution*

For each location in the image (original or padded) and a single convolution filter, we can apply a nonlinear activation function after the convolution

$$y_{i,j} = g\left(\sum_{-k \le r,s \le k} w_{r,s} x_{i+r,j+s}\right) \tag{12.2}$$

where g is some function like *ReLU*, sigmoid, tanh. The intuition is similar to what we had earlier in feedforward neural networks — if we don't add non-linear activation functions, a multi-layered convolutional neural network can be easily reduced to a linear model!

12.2.5 *Channels*

In general, we do not only use one convolution filter. We construct a network of multiple layers, and for each of the layers, we apply multiple convolutional filters. Different filters will be able to detect different features of the image (*e.g.*, one filter detects edges and one filter detects dots), and we want to apply different filters independently on the input image. The result of applying a given filter on a single input image is called a *channel*. We can stack the channels to create a $3D$ structure, as shown in Figure 12.8.

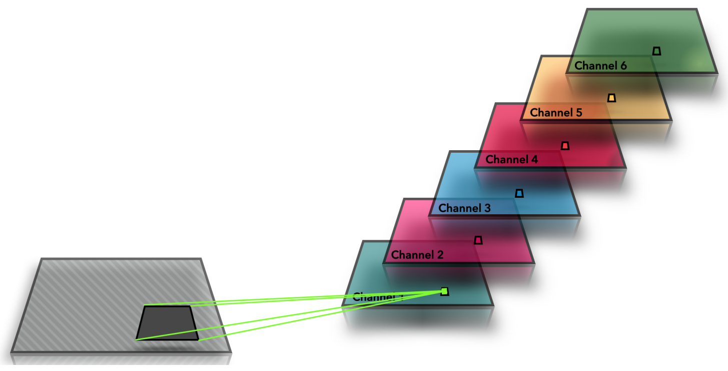


Figure 12.8: Each filter creates one channel. The output of a convolutional layer has multiple output channels.

Next, let's imagine that we want to build a deep neural network with multiple convolutional layers (state-of-the-art CNNs have 100 or even 1000 layers!). A typical convolutional layer in the middle of the network will have several input channels (equivalent to the number of output channels from the previous layer) and multiple output channels. How can we determine the number of filters needed?

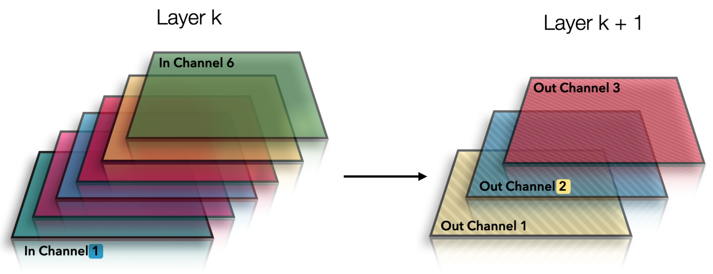


Figure 12.9: A convolutional layer which has multiple input and multiple output channels.

In this case, we want to define a filter for every possible pair of input and output channels. The output image of a particular output channel will be the summation of the output images from each of the input channels, after applying the corresponding filter. We can also add a nonlinear activation function g after taking the summation of the output images. That is, (12.2) can be rewritten for the output image in the v-th output channel as:

$$y_{i,j}^{(v)} = g\left(\sum_{u=1}^{n_{in}} \sum_{-k \le r,s \le k} w_{r,s}^{(u,v)} x_{i+r,j+s}^{(u)}\right) \tag{12.3}$$

where n_{in} is the number of input channels, $\mathbf{X}^{(u)}$ is the image at the u-th input channel, $\mathbf{Y}^{(v)}$ is the image at the v-th output channel, and $\mathbf{W}^{(u,v)}$ is the filter between the u-th input channel and the v-th output channel.

Example 12.2.6. *Assume there are 6 input channels and 3 output channels, and the filter size is* 5×5. *Then for every* 6×3 *pair of input and output channel, we have a kernel of weights of size* 5×5, *so there are a total of* $6 \times 3 \times 5 \times 5 = 450$ *weights.*

12.2.6 Pooling

Pooling is another popular way to reduce the size of the output of a convolutional layer. In contrast to stride, which applies convolution operation every s pixels, pooling partitions each image (channel) to patches of size $\Delta \times \Delta$ and performs a reduction operation on each patch. You can think of this as similar to what happens when you lower the resolution of an image. The reduction operation can either involve taking the max of all the values in the patch ("max-pooling"):

$$y_{i,j} = \max_{1 \le r,s \le \Delta} X_{(i-1)\cdot\Delta+r,(j-1)\cdot\Delta+s}$$

or taking the average of all the values in the patch ("mean-pooling"):

$$y_{i,j} = \frac{1}{\Delta^2} \sum_{r,s=1}^{\Delta} X_{(i-1)\cdot\Delta+r,(j-1)\cdot\Delta+s}$$

The pooling operation can reduce the image size by a factor of Δ^2.

If the input image is of size $m \times n$, the size of the image after pooling will be $\lfloor m/\Delta \rfloor \times \lfloor n/\Delta \rfloor$.

Example 12.2.7. *If the size of an input image to a pooling layer is* 6×6 *and* $\Delta = 2$, *then the output is of size* 3×3.

12.2.7 A Full Convolutional Neural Network

Let's put everything together and consider a full convolutional neural network. Figure 12.11 shows a typical example of a convolutional neural network. A convolutional neural network typically begins by stacking multiple convolutional layers and pooling layers. Each convolutional layer has its own kernel size and number of output channels; similarly, each pooling layer has its own kernel size. This is

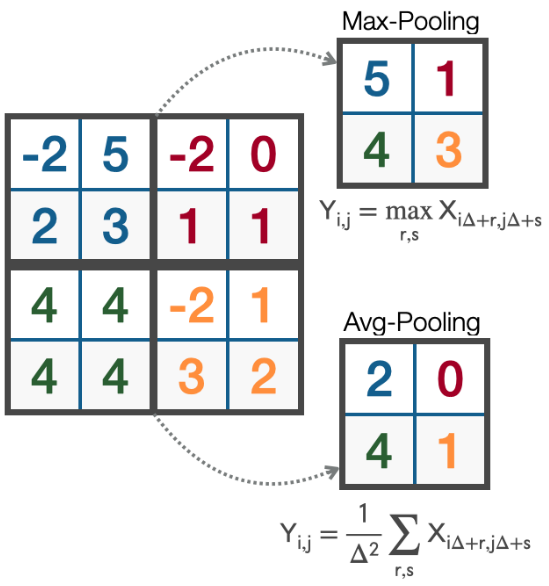

Max-Pooling
-2 5 -2 0
2 3 1 1
4 4 -2 1
4 4 3 2
5 1
4 3
$Y_{i,j} = \max_{r,s} X_{i\Delta+r, j\Delta+s}$
Avg-Pooling
2 0
4 1
$Y_{i,j} = \frac{1}{\Delta^2} \sum_{r,s} X_{i\Delta+r, j\Delta+s}$

Figure 12.10: Max-pooling vs mean-pooling.

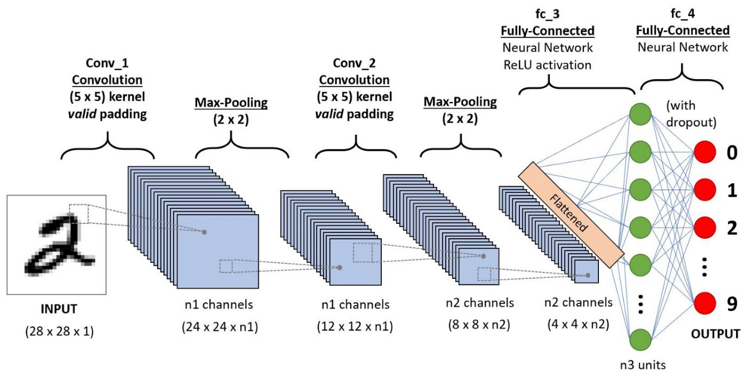


Figure 12.11: A illustration of a full convolutional neural network.

followed by several fully-connected layers at the end. Since the output images of convolutional layers are 2-dimensional, it is customary to "flatten" the images into a $1D$ vector (*i.e.*, append one row after another) before applying the fully connected layers. Intuitively, we can think of the convolutional and pooling layers as learning interesting image features (*e.g.*, stripes or dots) while the fully-connected layers map these features to output classes (*e.g.*, zebras have a lot of stripes).

All the weights in a convolutional neural network (including weights in kernels, fully-connected layers) can be learned via the backpropagation algorithm in Chapter 11. Again, modern deep learning libraries (*e.g.*, PyTorch, TensorFlow) have all the convolutional and pooling layers implemented and can compute gradients automatically!

Finally, the above convolutional neural network is still a simple network, compared to modern convolutional neural networks. Interested students can look up architectures such as AlexNet, Inception, VGG, ResNet and DenseNet.

12.2.8 Designing a Convolutional Network

While we described convolutional nets above, we did not give a good explanation of why they are well-suited to solve vision tasks. Working through the next few examples will help you understand their power. The idea is that convolution is a parameter-efficient [4] architecture that can "search for patterns" anywhere in the image. For example, suppose the net has to decide if the input image has a triangle anywhere in it. If the net were fully connected, it would have to learn how to detect triangles centered at every possible pixel location (i, j). By contrast, if a simple convolutional filter can detect a triangle, we can just replicate this filter in all patches to detect a triangle anywhere in the image.

[4] Which usually goes with sample-efficiency!

Now consider the CNN architecture in Figure 12.12. The architecture has two convolutional layers, the first with a *ReLU* activation function, and the second with a sigmoid activation function. [5] We will choose an appropriate convolutional weight and bias such that the architecture can detect a particular simple visual pattern.

[5] In both Example 12.2.8 and Example 12.2.9, the second convolutional layer can be considered a fully connected layer if we flatten image Y

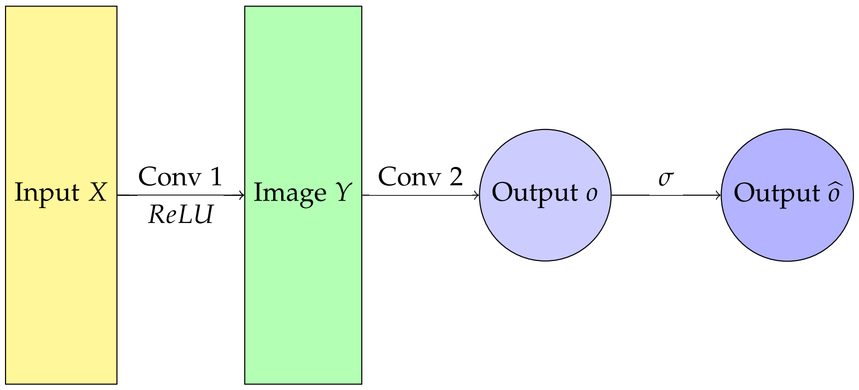


Figure 12.12: A sample CNN architecture that can be used to detect the patterns as aligned in Example 12.2.8 and Example 12.2.9.

Example 12.2.8. *The input to the network is a gray-scale image of size* 8×8 *(1 channel), and each pixel takes an integer value between* 0 *and* 255*, inclusive. If at least one pixel of the image has value exactly* 255*, the output of the CNN should have a value close to* 1 *and otherwise the output should have a value close to* 0*.*

We will now solve Example 12.2.8 by individually configuring the parameters for each convolutional layer in Figure 12.12. The first convolutional layer will have a 1×1 filter of weight 1, a bias of -254, and a ReLU activation function. The convolution will be applied with no padding, and with stride 1. That is, the (i, j)-th entry of the output image of the first convolutional layer will be

$$y_{i,j} = ReLU(x_{i,j} - 254) \qquad 1 \le i, j \le 8$$

where $x_{i,j}$ is the (i, j)-th entry of the input image. Notice that this value is zero everywhere, except if $x_{i,j} = 255$, in which case $y_{i,j}$ takes the value 1. That is,

$$y_{i,j} = \begin{cases} 1 & x_{i,j} = 255 \\ 0 & otherwise \end{cases}$$

See Figure 12.13 to see the effect of this choice of convolutional layer on a sample image. We see that we have now successfully identified the pixels in the input image that take the value 255.

$$\begin{bmatrix} 0 & 100 & 200 \\ 50 & 150 & 250 \\ 55 & 155 & 255 \end{bmatrix} \xrightarrow{\text{Conv 1}} \begin{bmatrix} -254 & -154 & -54 \\ -204 & -104 & -4 \\ -199 & -99 & 1 \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 12.13: The effect of the choice of the first convoluational layer for Example 12.2.8 on a sample image. Only a portion of the image is shown.

Next, consider the second convolutional layer with a 8×8 filter of all weights equal to 10, a bias of -5, and a sigmoid activation function. The convolution will be applied with no padding, and with stride 1. [6] The output, before the sigmoid, will be

$$o = \left(\sum_{i,j=1}^{8} 10 y_{i,j} \right) - 5$$

[6] Once the output image of the first convolutional layer is flattened to a vector of length 64, this can also be thought of as a fully-connected layer with input size 64 and output size 1.

Notice that this value is -5 if and only if there is no pixel such that $y_{i,j} = 1$ (*i.e.*, $x_{i,j} = 255$). If there is one such pixel, the output is 5; if there are two, the output is 15. The important thing is, the output is at least 5 if there is at least one pixel in the input image whose value is 255, and otherwise the output is ≤ 0. That is

$$o = \begin{cases} \ge 5 & \exists (i, j) : x_{i,j} = 255 \\ -5 & otherwise \end{cases}$$

Finally, when we apply the sigmoid function, the final output of the model will be

$$\widehat{o} = \sigma(o) = \begin{cases} \ge 0.99 & \exists (i, j) : x_{i,j} = 255 \\ 0.01 & otherwise \end{cases}$$

This is exactly what we wanted in Example 12.2.8.

Example 12.2.9. *The input to the network is a gray-scale image of size 8×8 (1 channel), and each pixel takes an integer value between 0 and 255, inclusive. If any part of the input image contains the following pattern:*

$$\begin{bmatrix} * & 255 & * \\ 255 & 255 & 255 \\ * & 255 & * \end{bmatrix} \tag{12.4}$$

the output of the CNN should have a value close to 1 and otherwise the output should have a value close to 0.

We use the same architecture as in Figure 12.12, but now with a different choice of parameters for the convolutional layers. The first convolutional layer will have a 3×3 filter with the following weights:

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

a bias of -1274, and a ReLU activation function. The convolution will be applied with no padding, and with stride 1. That is, the (i, j)-th entry of the output image of the first convolutional layer will be

$$y_{i,j} = ReLU\left(x_{i-1,j} + x_{i,j-1} + x_{i,j} + x_{i,j+1} + x_{i+1,j} - 1274\right) \quad 2 \leq i, j \leq 7$$

where $x_{i,j}$ is the (i, j)-th entry of the input image. [7] Notice that this value is zero everywhere, except if $x_{i,j} + x_{i-1,j} + x_{i,j-1} + x_{i,j+1} + x_{i+1,j} = 1275$, in which case $y_{i,j}$ takes the value 1. This can only happen if $x_{i-1,j} = x_{i,j-1} = x_{i,j} = x_{i,j+1} = x_{i+1,j} = 255$. That is, if the input image has the pattern in (12.4) centered around (i, j).

[7] Since there is no padding, the values $y_{1,1}, y_{1,8}, y_{8,1}, y_{8,8}$ are not defined.

$$y_{i,j} = \begin{cases} 1 & \textit{Pattern in (12.4) exists at } (i,j) \\ 0 & \textit{otherwise} \end{cases}$$

See Figure 12.14 to see the effect of this choice of convolutional layer on two sample images.

$$\begin{bmatrix} 0 & 255 & 0 \\ 255 & 250 & 255 \\ 0 & 255 & 0 \end{bmatrix} \xrightarrow{\text{Conv 1}} \begin{bmatrix} * & * & * \\ * & -4 & * \\ * & * & * \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} * & * & * \\ * & 0 & * \\ * & * & * \end{bmatrix}$$

$$\begin{bmatrix} 0 & 255 & 0 \\ 255 & 255 & 255 \\ 0 & 255 & 0 \end{bmatrix} \xrightarrow{\text{Conv 1}} \begin{bmatrix} * & * & * \\ * & +1 & * \\ * & * & * \end{bmatrix} \xrightarrow{ReLU} \begin{bmatrix} * & * & * \\ * & 1 & * \\ * & * & * \end{bmatrix}$$

Figure 12.14: The effect of the choice of first convoluational layer for Example 12.2.9 on two sample images. Only a portion of the images is shown.

Next, consider the second convolutional layer with a 6×6 filter of all weights equal to 10, a bias of -5, and a sigmoid activation function. The convolution will be applied with no padding, and with stride 1. The output, before the sigmoid, will be

$$o = \left(\sum_{i,j=2}^{7} 10 y_{i,j} \right) - 5$$

Notice that this value is -5 if and only if there is no pixel such that $y_{i,j} = 1$ (*i.e.*, the pattern exists at (i, j)). If there is one such pixel, the output is 5; if there are two, the output is 15. The important thing is, the output is at least 5 if there is at least one copy of the given pattern, and otherwise the output is ≤ 0. That is

$$o = \begin{cases} \geq 5 & \exists (i,j) : \textit{Pattern in } (12.4) \textit{ exists at } (i,j) \\ -5 & \textit{otherwise} \end{cases}$$

Finally, when we apply the sigmoid function, the final output of the model will be

$$\widehat{o} = \sigma(o) = \begin{cases} \geq 0.99 & \exists (i,j) : \textit{Pattern in } (12.4) \textit{ exists at } (i,j) \\ 0.01 & \textit{otherwise} \end{cases}$$

This is exactly what we wanted in Example 12.2.9.

12.3 *Backpropagation for Convolutional Nets*

A convolutional neural network is a special case of a feedforward neural network where we use convolutional layers, instead of fully-connected layers as in Chapter 11. Therefore, we can apply the same basic idea of backpropagation so that we can run the gradient descent algorithm, although the details of the calculation are slightly different.

The biggest difference is that in a fully-connected layer, each weight is used exactly once, while in a convolutional layer, each weight is applied multiple times throughout the input image. [8] This makes the computation for the gradient slightly more convoluted. But the basic idea is the same — identify all paths through which the corresponding weight affects the output of the model and add up the amount of effect for each path.

[8] This phenomenon is also known as *weight sharing*.

Figure 12.15 shows a portion of a sample neural network where weight sharing occurs. That is, the same weight w is used between the following four pairs of nodes: $(x_1, y_1), (x_2, y_2), (x_2, y_3), (x_3, y_4)$. If we wanted to find the gradient $\partial o / \partial w$, we need to consider the four paths that the weight w affects the output: $w \to y_i \to o$ where $1 \leq i \leq 4$.

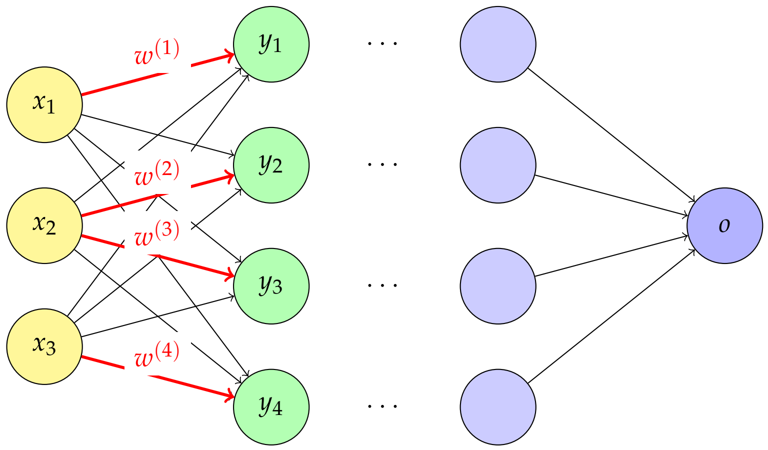


Figure 12.15: A sample neural network where weight sharing occurs. $w^{(1)}, w^{(2)}, w^{(3)}, w^{(4)}$ are the copies of the same weight w.

What we will do is consider the four copies of the weight w as separate weights that will be denoted as $w^{(i)}$ where $1 \leq i \leq 4$. Since these weights are only used in one place in the layer, we are already familiar with computing the gradients $\partial o / \partial w^{(i)}$. Then we will add (or *pool*) these values to get the gradient $\partial o / \partial w$. This works because we can think of each $w^{(i)}$ as a function of w where $w^{(i)} = w$. Then by Chain Rule, we have

$$\frac{\partial o}{\partial w} = \sum_{i=1}^{4} \frac{\partial o}{\partial w^{(i)}} \cdot \frac{\partial w^{(i)}}{\partial w} = \sum_{i=1}^{4} \frac{\partial o}{\partial w^{(i)}}$$

12.3.1 *Deriving Backpropagation Formula for Convolutional Layers*

In this subsection, we derive the backpropagation formula for a convolutional layer. (As in many other places, if your instructor did not teach it in COS 324, consider this to be advanced reading.)

Recall that in a fully-connected layer (without an activation function), which computes $\vec{\mathbf{h}}^k = \mathbf{W}^{(k)} \vec{\mathbf{h}}^{(k-1)}$, the gradient with respect to a particular weight $w_{i,j}^{(k)}$ can be simply computed as

$$\frac{\partial \ell}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial h_i^{(k)}} \cdot \frac{\partial h_i^{(k)}}{\partial w_{i,j}^{(k)}} = \frac{\partial \ell}{\partial h_i^{(k)}} \cdot h_j^{(k-1)}$$

This is because the weight $w_{i,j}^{(k)}$ is only used to compute $h_i^{(k)}$ out of all nodes in the next hidden layer. In comparison, consider a convolutional layer, which computes an output image $\mathbf{Y} \in \mathbb{R}^{n \times n}$ from an input image $\mathbf{X} \in \mathbb{R}^{m \times m}$ and filter $\mathbf{W} \in \mathbb{R}^{(2k+1) \times (2k+1)}$. Notice that the weight $w_{i,j}$ is used to compute all of the pixels in the output image. Therefore, we just need to add (or *pool*) the gradient flow from each of these paths. The gradient with respect to a particular weight

$w_{i,j}$ will be

$$\begin{aligned}\frac{\partial \ell}{\partial w_{i,j}} = &\left(\frac{\partial \ell}{\partial y_{1,1}} \cdot \frac{\partial y_{1,1}}{\partial w_{i,j}} + \frac{\partial \ell}{\partial y_{1,2}} \cdot \frac{\partial y_{1,2}}{\partial w_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{1,n}} \cdot \frac{\partial y_{1,n}}{\partial w_{i,j}}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{2,1}} \cdot \frac{\partial y_{2,1}}{\partial w_{i,j}} + \frac{\partial \ell}{\partial y_{2,2}} \cdot \frac{\partial y_{2,2}}{\partial w_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{2,n}} \cdot \frac{\partial y_{2,n}}{\partial w_{i,j}}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{n,1}} \cdot \frac{\partial y_{n,1}}{\partial w_{i,j}} + \frac{\partial \ell}{\partial y_{n,2}} \cdot \frac{\partial y_{n,2}}{\partial w_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{n,n}} \cdot \frac{\partial y_{n,n}}{\partial w_{i,j}}\right)\end{aligned}$$

Assuming there is zero padding, this can be calculated as

$$\begin{aligned}\frac{\partial \ell}{\partial w_{i,j}} = &\left(\frac{\partial \ell}{\partial y_{1,1}} \cdot x_{i+1,j+1} + \frac{\partial \ell}{\partial y_{1,2}} \cdot x_{i+1,j+2} + \ldots + \frac{\partial \ell}{\partial y_{1,n}} \cdot x_{i+1,j+n}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{2,1}} \cdot x_{i+2,j+1} + \frac{\partial \ell}{\partial y_{2,2}} \cdot x_{i+2,j+2} + \ldots + \frac{\partial \ell}{\partial y_{2,n}} \cdot x_{i+2,j+n}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{n,1}} \cdot x_{i+n,j+1} + \frac{\partial \ell}{\partial y_{n,2}} \cdot x_{i+n,j+2} + \ldots + \frac{\partial \ell}{\partial y_{n,n}} \cdot x_{i+n,j+n}\right)\end{aligned}$$

Notice that the equation above can be rewritten as

$$\frac{\partial \ell}{\partial w_{i,j}} = \sum_{1 \le r,s \le n} \frac{\partial \ell}{\partial y_{r,s}} \cdot x_{i+r,j+s} \tag{12.5}$$

That is, the Jacobian matrix $\partial \ell / \partial \mathbf{W}$ is the output when applying a convolution filter $\partial \ell / \partial \mathbf{Y}$ to the input matrix $\mathbf{X}$.

Similarly, we can try to calculate the Jacobian matrix with respect to the input matrix $\mathbf{X}$. Each input pixel $x_{i,j}$ is used to calculate the output pixels $y_{i+r,j+s}$ where $-k \le r, s \le k$. The gradient with respect to a particular input pixel will be

$$\begin{aligned}\frac{\partial \ell}{\partial x_{i,j}} = &\left(\frac{\partial \ell}{\partial y_{i-k,j-k}} \cdot \frac{\partial y_{i-k,j-k}}{\partial x_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{i-k,j+k}} \cdot \frac{\partial y_{i-k,j+k}}{\partial x_{i,j}}\right) \\ &+ \left(\frac{\partial \ell}{\partial y_{i-k+1,j-k}} \cdot \frac{\partial y_{i-k+1,j-k}}{\partial x_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{i-k+1,j+k}} \cdot \frac{\partial y_{i-k+1,j+k}}{\partial x_{i,j}}\right) \\ &+ \ldots \\ &+ \left(\frac{\partial \ell}{\partial y_{i+k,j-k}} \cdot \frac{\partial y_{i+k,j-k}}{\partial x_{i,j}} + \ldots + \frac{\partial \ell}{\partial y_{i+k,j+k}} \cdot \frac{\partial y_{i+k,j+k}}{\partial x_{i,j}}\right)\end{aligned}$$

Assuming zero padding, this is calculated as

$$\begin{aligned}\frac{\partial \ell}{\partial x_{i,j}} &= \left(\frac{\partial \ell}{\partial y_{i-k,j-k}} \cdot w_{k,k} + \ldots + \frac{\partial \ell}{\partial y_{i-k,j+k}} \cdot w_{k,-k}\right) \\ &\quad + \left(\frac{\partial \ell}{\partial y_{i-k+1,j-k}} \cdot w_{k-1,k} + \ldots + \frac{\partial \ell}{\partial y_{i-k+1,j+k}} \cdot w_{k-1,-k}\right) \\ &\quad + \ldots \\ &\quad + \left(\frac{\partial \ell}{\partial y_{i+k,j-k}} \cdot w_{-k,k} + \ldots + \frac{\partial \ell}{\partial y_{i+k,j+k}} \cdot w_{-k,-k}\right)\end{aligned}$$

which can be rewritten as

$$\frac{\partial \ell}{\partial x_{i,j}} = \sum_{-k \le r,s \le k} \frac{\partial \ell}{\partial y_{i+r,j+s}} \cdot w_{-r,-s} \tag{12.6}$$

That is, the Jacobian matrix $\partial \ell / \partial \mathbf{X}$ is the output when applying the horizontally and vertically inverted image of $\mathbf{W}$ as the convolutional filter to the input matrix $\partial \ell / \partial \mathbf{Y}$.

12.4 CNN in Python Programming

In this section, we discuss how to write Python code to implement Convolutional Neural Networks (CNN). As usual, we use the *numpy* package to speed up computation and the *torch* package to easily design and train the neural network. We also introduce the *torchvision* package:

- *torchvision*: This package focuses on computer vision applications and is integrated with the broader PyTorch framework. It provides access to pre-built models, popular datasets, and a variety of image transform capabilities. [9]

[9] Documentation is available at https://pytorch.org/vision/stable/index.html

The following code sample implements a CNN and trains it on a single image.

```python
# import necessary packages
import random
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader

# set random seeds to ensure reproducibility
torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
```

```
# load CIFAR10 data
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
train_data = torchvision.datasets.CIFAR10(root='./data', train=True,
                                      download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10(root='./data', train=False,
                                      transform=transform)

# helps iterate through the train/test data in batches
train_loader = DataLoader(dataset=train_data, batch_size=8, shuffle=True,
                                      num_workers=0)
test_loader = DataLoader(dataset=test_data, batch_size=8, shuffle=False,
                                      num_workers=0)

# define the CNN architecture
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # Conv2d takes # input channels, # output channels, kernel size
        self.conv1 = nn.Conv2d(3, 3, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(3, 16, 5)
        self.pool2 = nn.AvgPool2d(2, 2)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x

# extract one image from the dataset
images, labels = next(iter(train_loader))
image = images[0].unsqueeze(0)

# forward propagation
net = ConvNet()
output = net(image)

# choose the optimization technique to invoke
optimizer = torch.optim.SGD(net.parameters(), lr=0.01)

# backpropagation
loss = torch.norm(output - torch.ones(output.shape[1]))**2
loss.backward()
optimizer.step()
optimizer.zero_grad()
```

As usual, we start by importing packages.

```
import random
import numpy as np
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

The *DataLoader* class helps iterate through a dataset in batches.

Next, we fix all random seeds to ensure reproducibility.

```
torch.manual_seed(0)
np.random.seed(0)
random.seed(0)
```

Recall that programming languages on a classical computer can only implement pseudorandom methods, which always produce the same result for a given seed.

Then we load the CIFAR-10 dataset.

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])
train_data = torchvision.datasets.CIFAR10(root='./data', train=True,
                                      download=True, transform=transform)
test_data = torchvision.datasets.CIFAR10(root='./data', train=False,
                                      transform=transform)
```

The CIFAR-10 dataset contains simple images of a single object, and the images are labeled with the category of the objects they contain. Note that we normalize the dataset with a mean of 0.5 and standard deviation of 0.5 per color channel. Figure 12.16 shows a sampling of images from the dataset after the normalization.

Figure 12.16: Sample images from the CIFAR10 dataset.

Next we create *DataLoader* objects to help iterate through the dataset in batches. Each batch will consist of 8 images and 8 labels.

```
train_loader = DataLoader(dataset=train_data, batch_size=8, shuffle=True,
                                      num_workers=0)
test_loader = DataLoader(dataset=test_data, batch_size=8, shuffle=False,
                                      num_workers=0)
```

Then we define our CNN architecture in the *ConvNet* class.

```
class ConvNet(nn.Module):
    def __init__(self):
        super(ConvNet, self).__init__()
        # conv2d takes # input channels, # output channels, kernel size
        self.conv1 = nn.Conv2d(3, 3, 5)
        self.pool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(3, 16, 5)
        self.pool2 = nn.AvgPool2d(2, 2)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 10)

    def forward(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(x)
        x = F.relu(self.conv2(x))
        x = self.pool2(x)
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

Just like the FFNN code from the previous chapter, we define all the layers and activations we are going to use in the constructor. Note that in addition to instances of the *nn.Linear* class and the *nn.ReLU* class, we also make use of classes like *nn.Conv2d* and *nn.MaxPool2d* which are specifically designed for CNNs.

We extract one training image with the following code.

```
images, labels = next(iter(train_loader))
image = images[0].unsqueeze(0)
```

The *unsqueeze()* function adds one dimension to the training data. This is called a *batch dimension*. Normally, we would run the training in batches, and the size of the data along the batch dimension will be equal to the number of images in each batch. Here, we only use one image for the sake of exposition.

We can now run forward propagation on a sample image with the code below.

```
net = ConvNet()
output = net(image)
```

We then implement the *squared error* loss. Alternatively, we could have chosen the cross-entropy loss or any other valid loss function.

```
loss = torch.norm(output - torch.ones(output.shape[1]))**2
```

Next, we calculate the gradients of the *loss* with the following line of code

```
loss.backward()
```

and update each of the parameters according to the Gradient Descent algorithm with the following line.

```
optimizer.step()
```

Finally, we reset the values of the gradients to zero with the following code.

```
optimizer.zero_grad()
```

Recall as discussed in the previous chapter that failing to do so will cause unintended training during subsequent iterations of backpropagation. Here, we called the *zero_grad()* function at the end of one iteration of backpropagation, but it may be a good idea to call this function right before calling *backward()*, just in case there are already gradients in the buffer before program execution (*e.g.*, if someone was working with the model beforehand in the interpreter).

In this section, we only showed how to run forward propagation and backpropagation on a single data point. In general, we train the model on the entire dataset multiple times. A single pass over the entire dataset is called an *epoch*.