Chapter 11

Convolutional Neural Networks

11.1 Learning Objectives

Convolutional neural networks (CNNs or ConvNets) have been used in image recognition since the 1980s. In the past decade, thanks to the increase in computational power and the amount of training data, CNNs have achieved superhuman performance on some complex visual tasks. This chapter covers:

- Motivation of a convolution layer
- The operation of convolution
- Pool layer
- LeNet-5
- Construction and training of CNNs using PyTorch

11.2 Architecture of Convolutional Neural Networks

11.2.1 Convolution

The neural networks, discussed in previous chapters, are *fully connected* networks. Specifically, each node (or neuron) in a layer takes the outputs of all nodes in the previous layer as its input. Furthermore, all weights on the connections are not correlated in general. However, in visual applications, this approach ignores a key property of images, which is that nearby pixels are more strongly correlated than more distant pixels. CNNs exploits this property by extracting *local features* that depend only on small subregions of the image. Information from such features can then be merged in later stages of processing in order to detect higher-order features and ultimately to yield information about the image as whole. In addition, local features that exist in one region of the image are likely to exist in other regions of the image. There are three mechanisms integrated in CNNs: 1) local receptive fields; 2) weight sharing, and 3) subsampling.

For example, for images with size 32x32x3 (32 wide, 32 high, 3 color channels), a single fully connected node in a first hidden layer would have 32*32*3 = 3072 weights. This amount still seems manageable, but clearly this fully connected structure does not scale to larger images. Consider a

fully connected network for an image of size 1000x1000x3. One node in the first hidden layer requires 3,000,000 weights (connections) (ignore the bias parameter). The first hidden layer with 1000 nodes would need 3G weights. In fact, this full connectivity ignores the *"local"* property of images.

Convolutional Neural Networks take advantage of the key property of images. A CNN is nothing else but a sparsely connected neural network with weight sharing in some degree. In the representation and analysis of CNNs, it is a common practice to arrange the nodes (or data) in 3-dimensional volumes: width, height, depth. The nodes in a CNN layer will only be connected to a small region of the previous layer, and all of the nodes in a feature map share the same weight values. Consider a convolution layer in Fig.1. The input to the layer is an image, denoted as $a^{[0]}$, a 3-dimension array or tensor. The nodes in the hidden layer are arranged as a 3-dimension array too, but with a different shape in general (in this case depth is 4). Each sheet of nodes in the hidden layer is called a feature map. Thus the hidden layer has 4 feature maps. Now let's focus on a particular feature map, say the one generated by W1. Any node in this feature map is connected to a subregion of $a^{[0]}$ with weights W*1*, which is called a *filter*. The size of the connected subregion is determined by the size of the filter, and the location of the subregion is determined by relative location of the node in the feature map and the sliding stride of the filter.

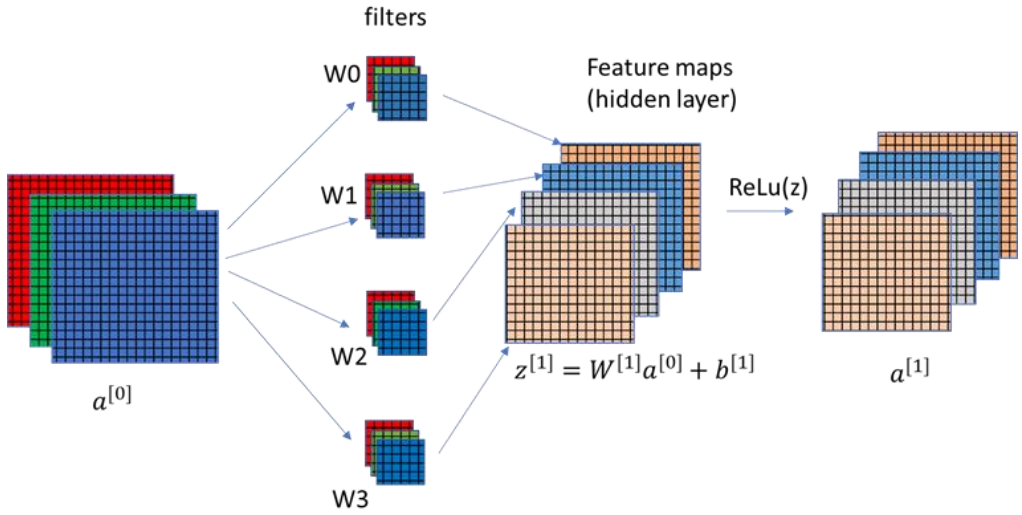
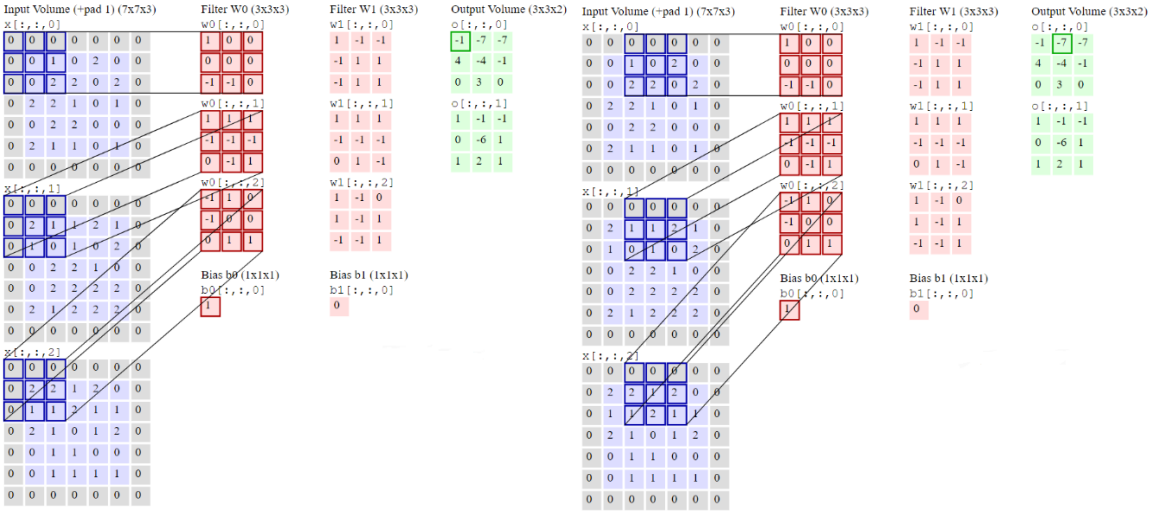


Fig.1 a convolution layer with 4 filters (channels)

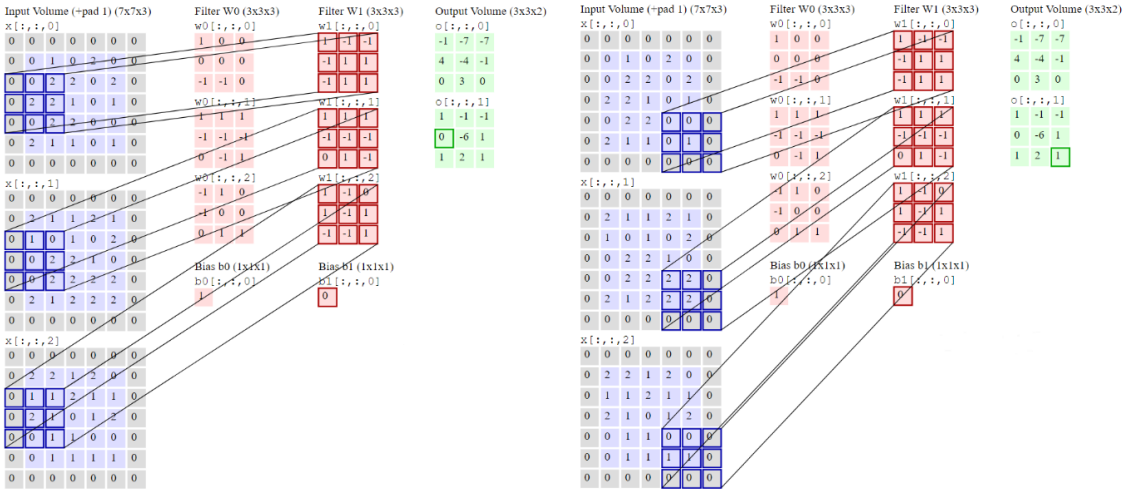
The convolution can be understood as this: 1)Each filter can be viewed as a 3-dimensional weighted mask, 2) the mask slides two-dimensionally (either horizontally or vertically) across the image at a pre-defined stride, 3) at each moment of sliding, the weighted sum of the local region of image covered by the mask is computed as one data point on the feature map. Like fully connected networks, an activation function is usually applied to the linear combination in elementwise.

To make it concrete, consider a numerical example of convolution, shown in Fig.2. There are two channels W0 and W1. The filters slide with a stride of 2. You are encouraged to verify the numerical results o[:,:,0] and o[:,:,1].



(a) W0 computes the output at (0,0) in the 1th feature map.

(b) W0 computes the output at (0,1) in the 1th feature map



(c) W1 computes the output at (1,0) in the 2th feature map

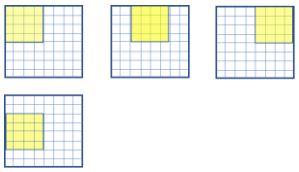
(d) W1 computes the output at (2,2) in the 2th feature map

Fig.2 A demo of convolution computation: padding=1, stride=2.

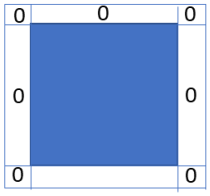
In general, to completely specify the sparse connection defined by the convolution, we need to pre-define four hyperparameters: filter size, depth, stride and zero-padding. Assume that input data (image or a certain kind of image features): $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$ (superscript for the layer index, subscript H, W and c for height, width, channel, respectively).

1) Filter size: the filter size is represented by $\left(f, f, n_c^{[l-1]}\right)$. For example, in Fig.2, f=3, $n_c^{[l-1]} = 3$.
2) Depth (or channel): denoted by $n_c^{[l]}$, the number of filters we would like to use, each learning to look for something different features in the input. For example, if the first Convolutional Layer takes as input the raw image, then different nodes along the depth dimension may activate in presence of various oriented edges, or blobs of color. We will refer to a set of nodes that are all looking at the same region of the input as a depth column. In Fig.2, $n_c^{[l]} = 2$.

3) Stride: denoted by *s*, we must specify the stride with which we slide the filter. When the stride is 1 then we move the filters one pixel at a time. When the stride is 2 (or uncommonly 3 or more, though this is rare in practice) then the filters jump 2 pixels at a time as we slide them around. This will produce smaller output volumes spatially. The figure (right) shows the case with stride of 2. In Fig.2, s=2.



4) Zero-padding: denoted by *p*, sometimes it will be convenient to pad the input volume with zeros around the border. The size of this zero-padding is a hyperparameter. The nice feature of zero padding is that it will allow us to control the spatial size of the output volumes (most commonly as we'll see soon we will use it to exactly preserve the spatial size of the input volume so that the input and output width and height are the same). The figure shows zero-padding of 1. In Fig.2, p=1.



We can compute the spatial size of the output volume as a function of the input data volume size ($n \times n \times n_c^{[l-1]}$), the filter size ($f \times f \times n_c^{[l-1]}$), the number of filters (depth) ($n_c^{[l]}$), the stride (s) with which they are applied, and the amount of zero padding used (p) on the border. A reader can easily verify that the output data volume of the Conv layer is given

$$\left(\frac{n+2p-f}{s}+1\right) \times \left(\frac{n+2p-f}{s}+1\right) \times n_c^{[l]} \quad (1)$$

As an example, the output data in Fig.2 has a size $(3 \times 3 \times 2)$ with n=5, p=1, f=3, s=2, $n_c^{[l]}$=2.

Fig.3 shows a ConvNet consisting of a series of convolution layers. The ConvNet takes an image $(39 \times 39 \times 3)$ as input. The first hidden layer, defined by the filters specified in the box, delivers the output of size $(37 \times 37 \times 10)$. The two subsequent hidden layers are similarly specified with different hyperparameters. The output layer is a fully connected layer with softmax activation.

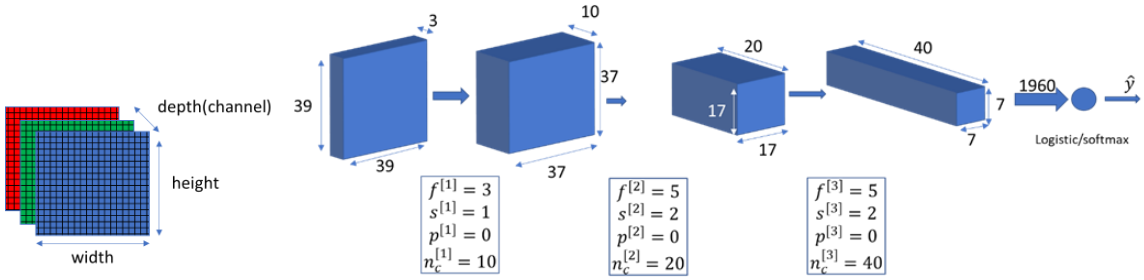


Fig.3 a simple ConvNet architecture

To describe the convolution operation by a mathematical equation, let's identify the connections between two layers and locate those connected nodes in the two layers. As we present earlier, the node located in row *i*, column *j* of the feature map *k* in a given convolution layer *l* is connected to the output of the nodes in the previous layer *l-1*, located in rows $i \times s$ to $i \times s + f - 1$ and columns $j \times s$ to $j \times s + f - 1$, across all feature maps in layer *l-1*. Fig.4 illustrates these connections, where each little square represents a node (or neuron, or data point). Note that all nodes in one feature map *k* in layer *l* share the same come-in weight matrix *W[:,:,:,k]*, and that all nodes at the same location *(i,j)* but across different feature maps at layer *l* are connected to the same

group of nodes in the previous layer but with different come-in weight matrices. The output of the convolution can be represented by

$$z(i,j,k) = b_k + \sum_{u=0}^{f-1}\sum_{v=0}^{f-1}\sum_{k'=0}^{n_c^{[l-1]}} x(i',j',k')W(u,v,k',k), \quad i' = i \times s + u, \; j' = j \times s + v \tag{2}$$

where

$x(i,j,k)$ is the output of the node at location *(i,j)* in feature map *k* in layer *l-1*.

$z(i,j,k)$ is the output of the node at location *(i,j)* in feature map *k* in layer *l*.

$W[:,:,:,k]$ is the weight matrix for feature map *k*. The element $W(u,v,k',k)$ is the weight for the connection between the node $(i \times s + u, j \times s + v, k')$ at layer *l-1* and the node (i,j,k) at layer *l*.

b_k is the bias for feature map *k*.

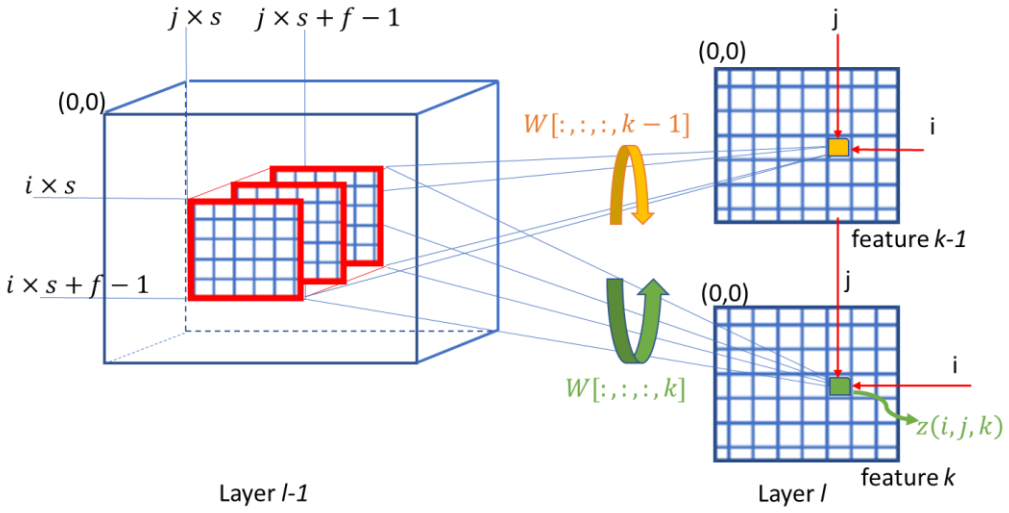


Fig. 4 the node connections between two layers in a convolution layer.

The convolution computation essentially involves dot products between the filters and local regions of the input. A common implementation pattern of the CONV layer is to take advantage of this fact and formulate the forward pass of a convolutional layer as one big matrix multiply. This is visualized by an example in Fig.5.

1) The local regions in the input image are stretched out into **columns** in an operation commonly called im2col. For example, consider a filter shown in Fig.5. Since the input is [37x37x10] and it is to be convolved with 5x5x10 filters at stride 2, we would take [5x5x10] blocks of pixels in the input and stretch each block into a column vector of size 5*5*10 = 250. Iterating this process in the input at stride of 2 gives (37-5)/2+1 = 17 locations along both width and height, leading to an output matrix X_col of im2col of size [250 x 289], where every column is a stretched out receptive field and there are 17*17 = 289 of them in total. Note that since the receptive fields overlap, every number in the input volume may be duplicated in multiple distinct columns.

2) The weights of the CONV layer are similarly stretched out into **rows**. For the example in Fig.5, since there are 20 filters of size [5x5x10], this would give a matrix W_row of size [20 x 250].
3) The result of a convolution is now equivalent to performing one large matrix multiply np.dot(W_row, X_col), which evaluates the dot product between every filter and every receptive field location. In our example, the output of this operation would be [20 x 289], giving the output of the dot product of each filter at each location. The result must finally be reshaped back to its proper output dimension [17x17x20].

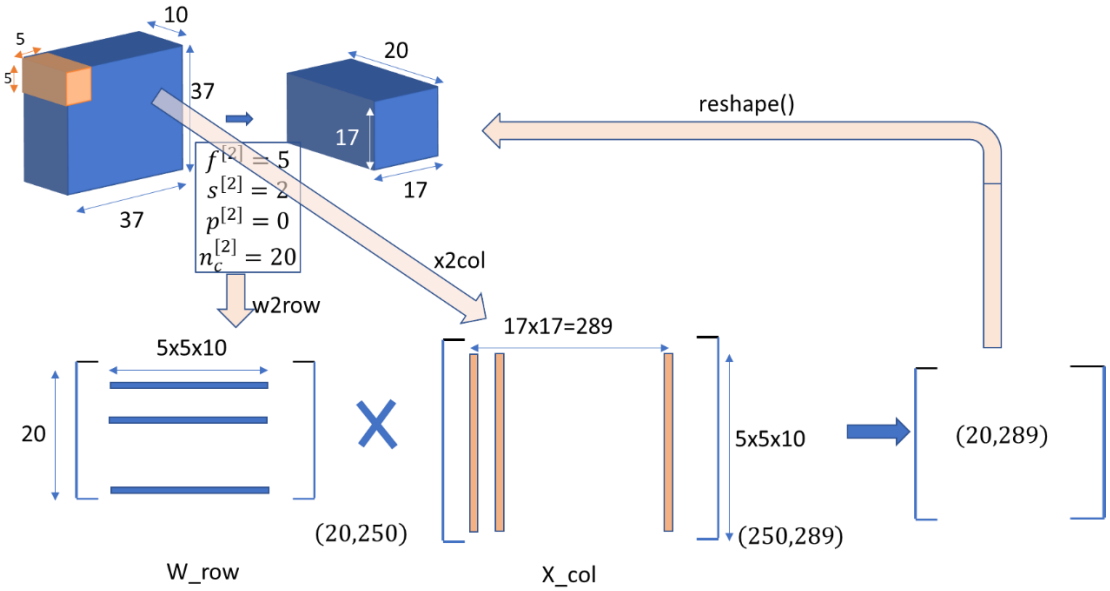


Fig.5 implementation of convolution as matrix multiplication

11.2.2 Pooling layer (POOL)

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX (or average) operation. The most common form is a pooling layer with filters of size 2x2 (f=2) applied with a stride of 2 (s=2), which downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX (or average) operation would in this case be taking a max (or average) over 4 numbers (2x2 region in some depth slice). An example is shown in Fig.6. The depth dimension remains unchanged. More generally, the pooling layer:

Input data: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$ (superscript for the layer index, subscript H, W and c for height, width, channel, respectively)
Filter size: $f^{[l]}$
Padding: $p^{[l]} = 0$ (commonly)
Stride: $s^{[l]}$

Output: $a^{[l]} \longrightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l-1]}$

$A^{[l]} \longrightarrow m \times n_H^{[l]} \times n_W^{[l]} \times n_c^{[l-1]}$ (batch: m is the number of image examples)

$$n_H^{[l]} = \frac{n_H^{[l-1]} - f^{[l]}}{s^{[l]}} + 1$$

$$n_W^{[l]} = \frac{n_W^{[l-1]} - f^{[l]}}{s^{[l]}} + 1$$

No parameter to learn.

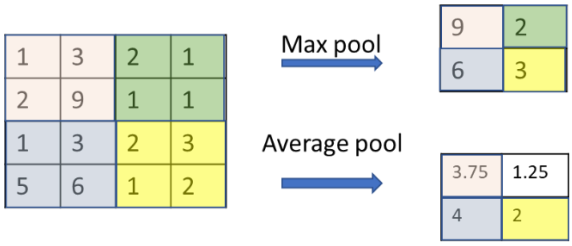


Fig. 6 Pool layer with f=2, s=2.

It is worth noting that there are only two commonly seen versions of the max pooling layer found in practice: A pooling layer with f=3,s=2 (also called overlapping pooling), and more commonly f=2,s=2. Pooling sizes with larger receptive fields are too destructive. In addition to max pooling, the pooling units can also perform other functions, such as average pooling or even L2-norm pooling. Average pooling was often used historically but has recently fallen out of favor compared to the max pooling operation, which has been shown to work better in practice.

Many people dislike the pooling operation and think that we can get away without it. For example, Striving for Simplicity: The All Convolutional Net proposes to discard the pooling layer in favor of architecture that only consists of repeated CONV layers. To reduce the size of the representation they suggest using larger stride in CONV layer once in a while. Discarding pooling layers has also been found to be important in training good generative models, such as variational autoencoders (VAEs) or generative adversarial networks (GANs).

11.2.3 Fully connected layer (FC)

After a series of Conv-layers or pooling layers, the last hidden layer (either Conv-layer or pooling layer) usually delivers a small sized data volume. Thus, the output layer is typically a fully connected layer, i.e., all the nodes in the output layer have full connections to all activations from the previous layer, as seen in regular Neural Networks.

11.2.4 CNN Example: LeNet-5

An important milestone in CNNs is a CNN, called LeNet-5, introduced by Yann LeCun, Leon Bottou, Yosuha Bengio and Patrick Haffner in a 1998 paper. The LeNet-5 was widely used to recognize hand-written digits. Yann LeCun, Leon Bottou, Yosuha Bengio and Patrick Haffner proposed a neural network architecture for handwritten and machine-printed character recognition. The architecture is illustrated by Fig. 7. In the original version of LeNet-5, the activation functions for conv1, conv2, FC3(fully connected), and FC4(fully connected) are tanh(), and pool layers use averaging pool without extra activation function. We can use ReLU() to replace tanh() activations and/or use max pooling for a variation of LeNet-5.

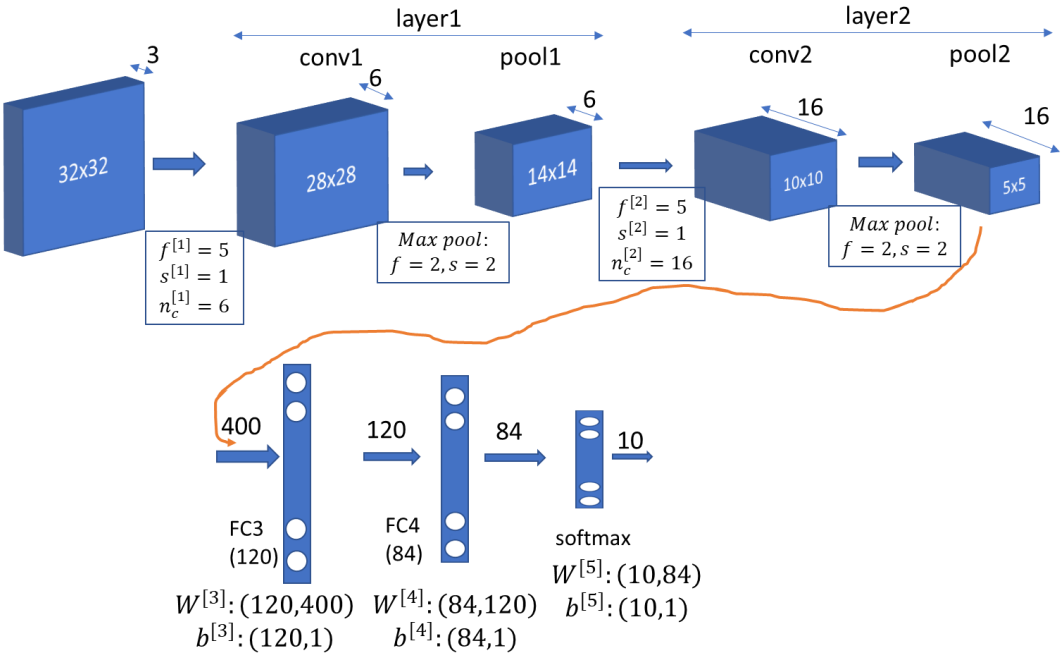


Fig. 7 Architecture of LeNet-5

11.3 Transposed Convolution

A typical convolution operation usually results in an output with a smaller or same size of feature map compared to the feature size of input, specified by equation (1). For example, the convolution of input (32x32) with filter (5x5) with stride 1 and no zero-padding delivers an output (28x28). Zero-padding is usually used to keep the output with a same size as the input. For example, the convolution of input (32x32) with a filter (3x3) with stride 1 and one zero-padding will generate an output (32x32). A large (more than one) stride will dramatically reduced the shape of the output.

The need for transposed convolutions generally arises from the desire to use a transformation going in the opposite direction of a normal convolution, i.e., from something that has the small shape to something that has the significantly larger shape. For example, we will see, the generator in GANs needs to generate an image from a random vector with a much smaller size. Transposed convolutions have been employed in more and more work (Zeiler et al., 2011; Zeiler and Fergus, 2014; Long et al., 2015; Radford et al., 2015; Visin et al., 2015; Im et al., 2016). For instance, one might use such a transformation as the decoding layer of a convolutional autoencoder or to project feature maps to a higher-dimensional space. A comprehensive description of transposed convolution can be found in (Vincent 2018).

Zeiler, M. D. and Fergus, R. (2014). Visualizing and understanding convolutional networks. In Computer vision–ECCV 2014 , pages 818–833. Springer.

Zeiler, M. D., Taylor, G. W., and Fergus, R. (2011). Adaptive deconvolutional networks for mid and high level feature learning. In Computer Vision (ICCV), 2011 IEEE International Conference on, pages 2018–2025. IEEE.

Long, J., Shelhamer, E., and Darrell, T. (2015). Fully convolutional networks for semantic segmentation. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pages 3431–3440.

Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434 .
Visin, F., Kastner, K., Courville, A. C., Bengio, Y., Matteucci, M., and Cho, K. (2015). Reseg: A recurrent neural network for object segmentation.
Im, D. J., Kim, C. D., Jiang, H., and Memisevic, R. (2016). Generating images with recurrent adversarial networks. arXiv preprint arXiv:1602.05110 .
Vincent Dumoulin and Francesco Visin, A guide to convolution arithmetic for deep learning, 2018

11.3.1 Convolution as a Matrix Multiplication

To understand the operation of transposed convolution, it is helpful to represent the general convolution as a matrix multiplication. Consider an input map feature 4x4 is convoluted with a filter/kernel 3x3 with stride 1 and no zero-padding, shown in Fig.8.

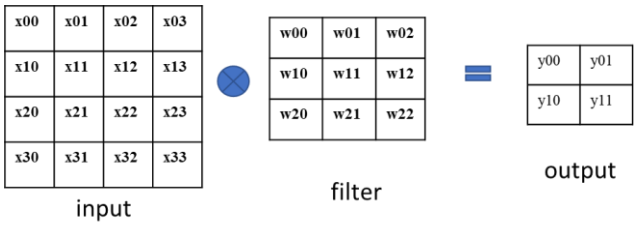


Fig.8 direct convolution

If we denote filter, input and output in a vector, respectively, as

$$W = \begin{pmatrix} w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 & 0 & 0 & 0 & 0 & 0 \\ 0 & w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 & 0 \\ 0 & 0 & 0 & 0 & 0 & w00 & w01 & w02 & 0 & w10 & w11 & w12 & 0 & w20 & w21 & w22 \end{pmatrix} \quad (3)$$

$X = \begin{pmatrix} x00 \\ x01 \\ x02 \\ x03 \\ x10 \\ x11 \\ \vdots \\ x32 \\ x33 \end{pmatrix}$ $Y = \begin{pmatrix} y00 \\ y01 \\ y10 \\ y11 \end{pmatrix}$, then the convolution can be represented by

$$Y = W \times X \quad (4)$$

The weights are represented by a sparse matrix W. This multiplication achieves the convolution operation by taking the input matrix flattened as a 16-element vector and produces a 4-element vector that is later reshaped as the 2x2 output matrix.

11.3.2 Transposed Convolution

Let's now consider an opposite mapping, i.e., mapping from a 4-dimensional space to a 16-dimensional space, while keeping the connectivity pattern of the convolution. This operation is known as a transposed convolution, also called fractionally strided convolution or deconvolution. If we take Y and X in (4) as input and output, respectively, the transposed matrix of W can be used for this mapping,

$$X = W^T \times Y \quad (5)$$

Note that the same notations in (4) and (5) take the same shapes but not necessarily the same values. Equation (5) shows the mapping from a 4-D space to a 16-D space with a W specified by (3),

implemented by a transposed convolution (shown in Fig.9). To maintain the same connectivity pattern in the equivalent convolution it is necessary to zero pad the input in such a way that the first (top-left) application of the kernel only touches the top-left pixel, i.e., the padding has to be equal to the size of the kernel minus one. It is equivalent to convolving a re-arranged 3x3 kernel over a 2x2 input padded with a 2x2 border of zeros using unit strides. Examples of kernel strides are shown in Fig.10.

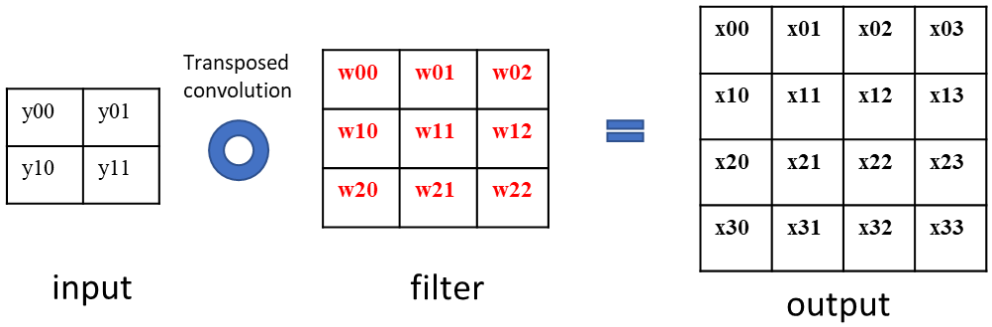


Fig.9 transposed convolution

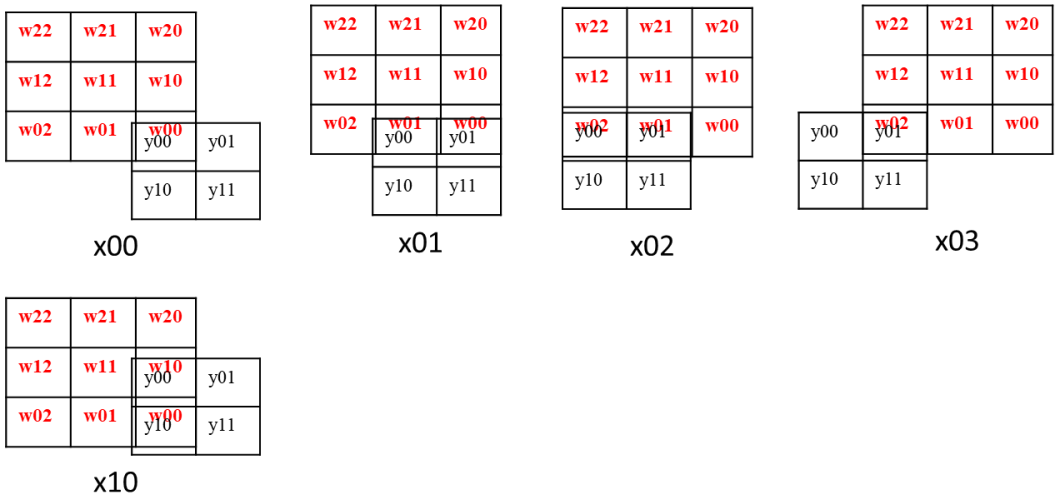


Fig.10 transposed convolution is equivalent to a direct convolution with enough zero-paddings

11.3.3 Zero padding in transposed convolution

The transposed convolution discussed in the previous section has a default setting with non-padding and unit stride. Note that the terms *padding* and *stride* refer to its corresponding direct (or opposite) convolution. Thus, the meaning of padding and stride is not straightforward.

Knowing that the transpose of a non-padded convolution is equivalent to convolving a zero padded input, it would be reasonable to suppose that the transpose of a zero padded convolution is equivalent to convolving an input padded with less zeros. Let's consider the direct convolution with zero-padding p, kernel size fxf, input nxn, and unit stride (s=1). The output size should be (n+2p-f+1) x (n+2p-f+1). In the corresponding transposed convolution, the input size is (n+2p-

f+1)x(n+2p-f+1). The transposed convolution is equivalent to a direct convolution with p' zero-paddings (as shown in Fig.10) specified by

$$(n + 2p - f + 1) + 2p' - f + 1 = n \tag{6.a}$$

$$p' = f - p - 1 \tag{6.b}$$

Thus, for a transposed convolution with input mxm, kernel fxf, *p* zero-padding and unit stride, the output size is

$$m + 2p' - f + 1 = m + f - 2p - 1 \tag{7}$$

For example, in a direct convolution, the convolution of 8x8 input (n=8) with a kernel 4x4 (f=4) at non-padding (p=0) and unit stride (s=1) will deliver an output 5x5 (n+2p-f+1=5). Thus, a corresponding transposed convolution, with the same setting (i.e., a kernel 4x4 at non-padding and unit stride), should take input 5x5, and deliver output 8x8. If one zero-padding is added, the output size 5x5 of a direct convolution requires the input size 6x6. Thus, the transposed convolution with input 5x5 and one zero-padding will deliver an output 6x6.

```
input = torch.randn(1, 3, 5, 5)   # batches =1, channels=3, input size 5x5
model=nn.ConvTranspose2d(3, 10, 4, 1, 0, bias=False)  #input channels=3, output channel=10, kernel=4x4, stride=1, padding=0.
output = model(input)
output.size()
        torch.Size([1, 10, 8, 8])

input = torch.randn(1, 3, 5, 5)   # batches =1, channels=3, input size 5x5
m=nn.ConvTranspose2d(3, 10, 4, 1, 1, bias=False)  #input channels=3, output channel=10, kernel=4x4, stride=1, padding=1.
output = model(input)
output.size()
        torch.Size([1, 10, 6, 6])
```

Two special cases for padding, namely, half (same) padding and full padding, are discussed below.

- **Same padding**: to maintain the output size as the same as the input size, half (same) padding is applied. From (7), in order for the input and output sizes to be the same, we have

  $$p = \frac{f-1}{2} \tag{8}$$

  Thus, for half (same) padding, the size of kernel should be odd.

```
input = torch.randn(1, 3, 5, 5)   # batches =1, channels=3, input size 5x5
model=nn.ConvTranspose2d(3, 10, 3, 1, 1, bias=False)  #input channels=3, output channel=10, kernel=3x3, stride=1, padding=1.
output = model(input)
output.size()
        torch.Size([1, 10, 5, 5])
```

- **Full padding**: the transpose convolution with full padding is implemented as a non-padded direct convolution (i.e. p'=0). Full padding means that a full padding (p) would be needed if the output shape is inversely mapped to the input shape with the kernel size. We have the full padding value *p* by setting *p'*=0 in (6.b),

  $$p = f - 1 \tag{9}$$

```
input = torch.randn(1, 3, 5, 5)   # batches =1, channels=3, input size 5x5
model=nn.ConvTranspose2d(3, 10, 3, 1, 2, bias=False)  #input channels=3, output channel=10, kernel=3x3, stride=1, padding=2.
output = model(input)
output.size()
        torch.Size([1, 10, 3, 3])
```

11.3.4 Stride in transposed convolution

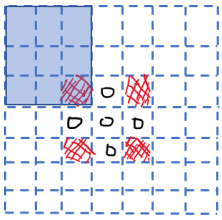
Similar to zero-padding specified for transposed convolution, stride (s) in transposed convolution refers the step size of kernel sliding in its direct convolution counterpart. It is intuitive to expect that the transposed convolution with s>1 involves an equivalent convolution with s<1. This is why transposed convolutions are also called fractionally strided convolutions. The fractional stride is implemented by inserting zeros between input elements, which makes the kernel slide around at a slower pace than with unit strides. Let *m* be the input size, *p* be the zero padding, *s* be the stride, *f* is the kernel size. We consider the following cases separately.

Case 1: *p=0* (no zero padding), but with a stride *s>1*. After inserting s-1 zeros between input elements, the input is stretched to *m+(m-1)(s-1)*. Then the transposed convolution is equivalent to the direct convolution on the stretched input with f-1 zero padding. Thus, the output size of the transposed convolution is

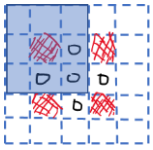
$$m + (m-1)(s-1) + f - 1 \tag{10}$$

Fig.11(a) shows the zero-padded stretched input for an example with m=2, p=0, s=2, f=3. The output size is 5.

```
input = torch.randn(1, 1, 2, 2)   # batches =1, channels=1, input size 2x2
model=nn.ConvTranspose2d(1, 1, 3, 2, 0, bias=False)  #input channels=1, output channel=1, kernel=3x3, stride=2, padding=0.
output = model(input)
output.size()
        torch.Size([1, 1, 5, 5])
```



(a) m=2, s=2, p=0, f=3



(b) m=2, s=2, p=1, f=3

Fig.11 transposed convolution with strides (s>1)

Case 2: p>0 (zero padding) and s>1 (strides). After inserting s-1 zeros between input elements, the input is stretched to *m+(m-1)(s-1)*. Then the transposed convolution is equivalent to the direct convolution on the stretched input with f-1-p zero padding. Thus, the output size of the transposed convolution is

$$m + (m-1)(s-1) + f - 1 - 2p \tag{11}$$

Fig.11(b) shows an example with m=2, k=3, p=1, s=2. The output size is 3.

```
input = torch.randn(1, 1, 2, 2)   # batches =1, channels=1, input size 3x3
model=nn.ConvTranspose2d(1, 1, 3, 2, 1, bias=False)  #input channels=1, output channel=1, kernel=3x3, stride=2, padding=1.
output = model(input)
output.size()
        torch.Size([1, 1, 3, 3])
```

11.3.5 Example of transposed convolution

In a generative adversarial network (GAN), we design a subnet work, called generator, to generate a fake image. One of the implementations of the generator is to transform a random vector to an image tensor through a few transposed convolution layers, as shown in Fig. 12, proposed in [].

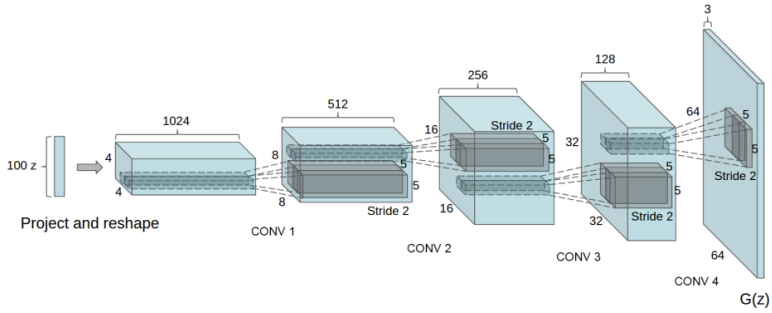
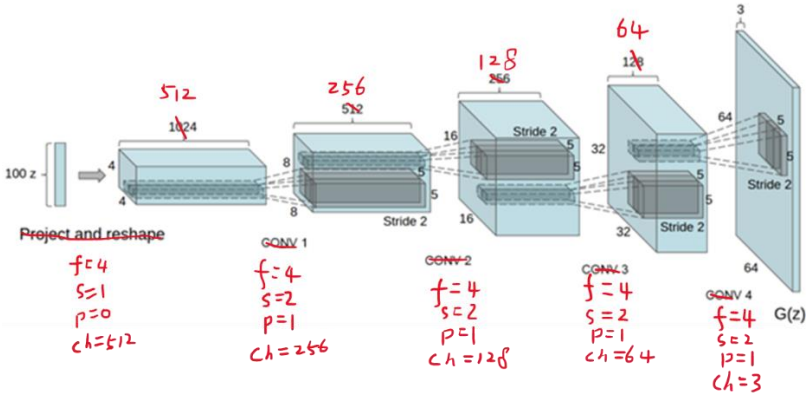


Fig. 2 transposed convolutions for generator in GAN [*]

[*] Radford, A., Metz, L., and Chintala, S. (2015). Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434 .

A variation of the generator is implemented as below, where nz=100, ngf=64, nc=3. The generator receives a batch of random noise vectors in a 4D tensor [batch_size, nz, 1, 1] and delivers a batch of fake images in a 4D tensor [batch_size, nc, 64, 64].




```
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d( nz, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
```

```
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d( ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d( ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d( ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        return self.main(input)
```

11.4 Implement CNNs in PyTorch

A convolution neural network may include convolutional layers, pooling layers, and fully connected layers. If we use NumPy data structure to implement a CNN, a large amount of efforts is required to deal with convolution operations and pooling operations at matrix level. Furthermore, it is almost impossible to derive an analytical gradient expression of the loss function with respect to all parameters, which is needed in the training process. PyTorch provides all we need to build and train CNNs in a very efficient way.

We will explain the key PyTorch points based on the dataset generated as follows.

```
import torch
import torchvision.transforms as transforms
from torchvision import datasets
import torch.nn.functional as F
import torch.optim as optim

%matplotlib inline
from matplotlib import pyplot as plt

data_path = '../torch_tutorial/data'
transform=transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5),(0.5,0.5,0.5))
    ])
cifar10 = datasets.CIFAR10(data_path, train=True, download=False, transfo
rm=transform)
cifar10_test = datasets.CIFAR10(data_path, train=False, download=False,tr
ansform=transform )
```

create sub datasets for only two classes: airplane or bird. label: 0 for airplaine, 1 for bird

```
label_map = {0: 0, 2: 1}
class_names = ['airplane', 'bird']
cifar2 = [(img, label_map[label]) for img, label in cifar10 if label in [
0, 2]]
```

```
cifar2_test = [(img, label_map[label]) for img, label in cifar10_test if
label in [0, 2]]

img, label=cifar2[1]
print(img.shape, label)

[output] torch.Size([3, 32, 32]) 1
```

11.4.1 Modules in torch.nn and functions in torch.nn.functional

In the previous chapter, we stacked modules nn.linear, nn.tanh, nn.LogSoftmax into a higher level module nn.Sequential() to construct a fully connected neural network. In this section, we will introduce two modules (i.e. Conv2d and MaxPool2d) dedicated for CNNs, and a alternative approach to nn.Sequential() for model construction: nn.Module to handle special issues in CNNs.

Conv2d()

In fact, The torch.nn module provides convolutions for 1, 2, and 3 dimensions: nn.Conv1d for time series, nn.Conv2d for images, and nn.Conv3d for volumes or videos. We focus on nn.Conv2d here. For example,

```
nn.Conv2d(3, 16, kernel_size=3, padding=1)
```

defines a convolution module with 3 input features, 16 output features, filter size is 3×3, zero-padding p=1, stride s is 1 by default.

```
conv = nn.Conv2d(3, 16, kernel_size=3)
conv

[output] Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1))

conv.weight.shape, conv.bias.shape

[output] (torch.Size([16, 3, 3, 3]), torch.Size([16]))
```

nn.Conv2d() requires that the input data tensor has a shape of [batch_size, C, H,W], and the output shape will be determined by a few relevant hyperparameters in a format [batch_size, C, H,W]. For example,

```
img, _ = cifar2[0]
output = conv(img.unsqueeze(0))
img.unsqueeze(0).shape, output.shape

[output](torch.Size([1, 3, 32, 32]), torch.Size([1, 16, 30, 30]))
```

nn.MaxPool2d()

Similarly, nn.MaxPool2d() can be used to implement max pooling layer. It accepts the input data with a tensor shape [batch_size, C,H,W] and return the output tensor with a reduced size.

```
pool = nn.MaxPool2d(2)
output = pool(img.unsqueeze(0))
img.unsqueeze(0).shape, output.shape
[output](torch.Size([1, 3, 32, 32]), torch.Size([1, 3, 16, 16]))
```

Subclassing nn.Module

When we want to build models that do more complex things (e.g. reshape a tensor by *view*) than just applying one layer after another, we need to subclass nn.Module for a more flexibility, instead of simply stacking layers into nn.Sequential(). In order to subclass nn.Module, at a minimum we need to define a *forward* function that takes the inputs to the module and returns the output. This is where we define our module's computation. Typically, our computation will use other modules—predefined like Linear, Conv2d, or customized. To include these submodules, we typically define them in the constructor __init__ and assign them to self for use in the forward function. Consider a CNN shown in Fig. 8 to be implemented for an image classification task. The image [3,32,32] propagates through the following layers: conv2d, maxpool2d, conv2d, maxpool2d, flatten to 512, fully connected (FC) layer (from 512 to 32) with Tanh() activation, and FC (from 32 to 2) with linear only.

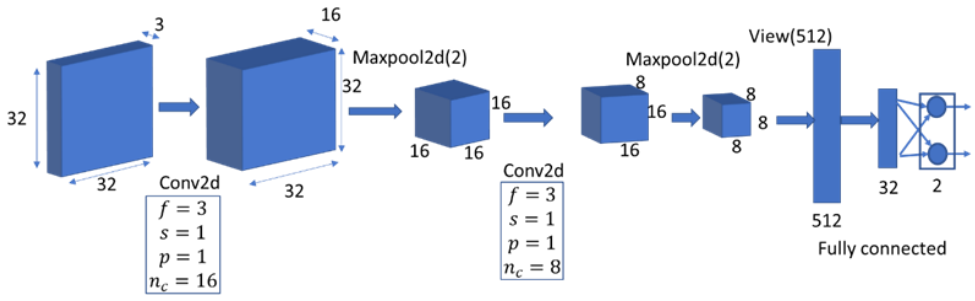


Fig.8 A CNN architecture

To implement this CNN, we create the *Net* class as follows.

```
class Net(nn.Module):
    def __init__(self):
            super().__init__()
            self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
            self.act1 = nn.Tanh()
            self.pool1 = nn.MaxPool2d(2)
            self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
            self.act2 = nn.Tanh()
            self.pool2 = nn.MaxPool2d(2)
            self.fc1 = nn.Linear(8 * 8 * 8, 32)
            self.act3 = nn.Tanh()
            self.fc2 = nn.Linear(32, 2)
    def forward(self, x):
            out = self.pool1(self.act1(self.conv1(x)))
            out = self.pool2(self.act2(self.conv2(out)))
            out = out.view(-1, 8 * 8 * 8)
            out = self.act3(self.fc1(out))
            out = self.fc2(out)
            return out
```

In the forward (self, x), we can manipulate the output of *self.pool2* directly and call view on it to turn it into a B × N vector. Note that we leave the batch dimension as –1 in the call to view, since in principle we don't know how many samples will be in the batch. Then we can make a instance of Net(), model, and access its parameter information.

```
model = Net()
numel_list = [p.numel() for p in model.parameters()]
       sum(numel_list), numel_list
[output](18090, [432, 16, 1152, 8, 16384, 32, 64, 2])
```

torch.nn.functional

The parameters in any submodule defined in __init__(self) are automatically registered in the top-level module Net() for autgrad in training process. Thus, for a purpose of coding efficiency, the operations carried by the submodules with no learning parameters (e.g. nn.Tanh(), nn.MaxPool2d(2)) can be moved to forward (self,x), but implemented in *functional* format.

Indeed, *torch.nn.functional* provides many functions that work like the modules we find in nn. But instead of working on the input arguments and stored parameters like the module counterparts, they take inputs and parameters as arguments to the function call. For instance, the functional counterpart of nn.Linear is nn.functional.linear, which is a function that has signature linear(input, weight, bias=None). The weight and bias parameters are arguments to the function. Thus, By "functional" here we mean "having no internal state"—in other words, "whose output value is solely and fully determined by the value input arguments." A more efficient version of Net() class would be:

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(16, 8, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(8 * 8 * 8, 32)
        self.fc2 = nn.Linear(32, 2)
    def forward(self, x):
        out = self.conv1(x)
        out = torch.tanh(out)
        out = F.max_pool2d(out,2)
        out = self.conv2(out)
        out = torch.tanh(out)
        out = F.max_pool2d(out,2)
        out = out.view(-1, 8 * 8 * 8)
        out = self.fc1(out)
        out = torch.tanh(out)
        out = self.fc2(out)
        return out
```

Before we train the CNN, we can simply check the computation flow of Net() on a single image as follows.

```
model = Net()
model(img.unsqueeze(0))
tensor([[-0.0185, -0.2155]], grad_fn=<AddmmBackward>)
```

11.4.2 Training CNNs

Construct training loop

The way of training a neural network using PyTorch, developed in the previous chapter, can be exactly applied to CNNs. Recall that the training loop consists of two nested loops: an outer one over the epochs and an inner one over the DataLoader that produces batches from the Dataset.

We specify the following steps in each loop:

1) Feed the inputs through the model (the forward pass). For Conv2d, the shape should be [batch, C, H, W]

2) Compute the loss (also part of the forward pass) using the outputs and labels as arguments.
3) Zero any old gradients.
4) Call loss.backward() to compute the gradients of the loss with respect to all parameters (the backward pass).
5) Have the optimizer update all the parameters in toward lower loss.

```
import datetime
def training_loop(n_epochs, optimizer, model, loss_fn, train_loader):
    for epoch in range(1, n_epochs + 1):
        loss_train = 0.0
        for imgs, labels in train_loader: #loop over dataset in batches
            outputs = model(imgs)         #feed batch to forward path
            loss = loss_fn(outputs, labels) #compute loss to be minimized
            optimizer.zero_grad()           #zero gradients for new iteration
            loss.backward()        #compute grad
            optimizer.step()       #update parameters
            loss_train += loss.item()  #accumulate loss over an epoch
        if epoch == 1 or epoch % 10 == 0:  #print the loss every 10 epochs
            print('{} Epoch {}, Training loss {}'.format(
            datetime.datetime.now(), epoch,
            loss_train / len(train_loader)))
```

Prepare for training

To run the training loop, we need to define train loader, model, optimizer, and loss function.

```
train_loader = torch.utils.data.DataLoader(cifar2, batch_size=64,shuffle=True)
model = Net() #
optimizer = optim.SGD(model.parameters(), lr=1e-2) #
loss_fn = nn.CrossEntropyLoss()
```

Run training loop

```
training_loop(
n_epochs = 100,
optimizer = optimizer,
model = model,
loss_fn = loss_fn,
train_loader = train_loader,
)
```

```
2020-08-03 09:38:46.480035 Epoch 1, Training loss 0.46304524058748964
2020-08-03 09:40:29.285060 Epoch 10, Training loss 0.34347422468434474
2020-08-03 09:42:22.660210 Epoch 20, Training loss 0.30677239587352534
2020-08-03 09:44:15.896499 Epoch 30, Training loss 0.285058619490095
2020-08-03 09:46:08.943939 Epoch 40, Training loss 0.26342424038489154
2020-08-03 09:48:01.860155 Epoch 50, Training loss 0.2442988303436595
2020-08-03 09:49:54.658291 Epoch 60, Training loss 0.22970903650590568
2020-08-03 09:51:49.370439 Epoch 70, Training loss 0.21459012900947766
2020-08-03 09:53:44.102215 Epoch 80, Training loss 0.20181653743526737
2020-08-03 09:55:37.420555 Epoch 90, Training loss 0.18856283588109501
2020-08-03 09:57:30.989276 Epoch 100, Training loss 0.1755762194657022
```

11.4.3 Testing the trained model

```
test_loader = torch.utils.data.DataLoader(cifar2_test, batch_size=64,
shuffle=False)

def validate(model, train_loader, test_loader):
    for name, loader in [("train", train_loader), ("test", test_loader)]:
        correct = 0
        total = 0
        with torch.no_grad():    # do not computer gradients
            for imgs, labels in loader:
                outputs = model(imgs)
                _, predicted = torch.max(outputs, dim=1)
                #get the highest index for max value

                total += labels.shape[0]  # total number of examples in dataset
                correct += int((predicted == labels).sum())
        print("Accuracy {}: {:.2f}".format(name , correct / total))

validate(model, train_loader, test_loader)
```

```
Accuracy train: 0.93
Accuracy test: 0.88
```

11.4.4 Save and load the trained model

In many applications, it takes long time to train a model. If the performance of trained model meets the design specifications, it is desirable to save the parameters for later use. When we deploy a trained model to a product, we just need to load the model and the saved parameters without re-training.

Save

```
torch.save(model.state_dict(), data_path + 'parameters.pt')
```

Load

```
loaded_model = Net()
loaded_model.load_state_dict(torch.load(data_path + 'parameters.pt'))
```

11.4.5 CIFAR-10 Image Classifier

In the previous chapter (section 10.6), we designed a 3-layer fully connected neural network to classify a subset of CIFAR-10 with two classes. In the previous section, a CNN (in Fig.8) was implemented using PyTorch. In this section, we will implement LeNet-5 to classify 10 classes in dataset CIFAR-10. CIFAR-10 consists of 60,000 (32 × 32) color (RGB) images (50000 in train set and 10000 in test set), labeled with an integer corresponding to 1 of 10 classes: airplane (0), automobile (1), bird (2), cat (3), deer (4), dog (5), frog (6), horse (7), ship (8), and truck (9).

```
#----prepare data loader

import torch
import torchvision
import torchvision.transforms as transforms

transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize((0.5,0.5,0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=False, transform=transform)
```

```
#-----download=False: already downloaded; download=True: download now

trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=False, transform=transform)

testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)



#------ visualize some examples

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')

import matplotlib.pyplot as plt
import numpy as np

def imshow(img):
    img = img / 2 +0.5  # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg,(1,2,0)))
    plt.show()

# get some random training images
dataiter= iter(trainloader)
images, labels = dataiter.next()

# show images
imshow(torchvision.utils.make_grid(images))

# print labels
print(' '.join ('%5s' % classes[labels[j]] for j in range(4)))



#----define LeNet-5

import torch.nn as nn
import torch.nn.functional as F

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3,6,5)
        self.pool = nn.MaxPool2d(2,2)
        self.conv2 = nn.Conv2d(6,16,5)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120,84)
        self.fc3 = nn.Linear(84,10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        # x = self.pool(x)
        x = F.max_pool2d(x,2,2)
        x = self.conv2(x)
        x = F.relu(x)
        # x = self.pool(x)
        x = F.max_pool2d(x,2,2)
        x = x.view(-1, 16*5*5)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
```

```
        return x
net = Net()


#---- train LeNet-5

import torch.optim as optim
criterion = nn.CrossEntropyLoss()
#optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.5)
optimizer = optim.Adam(net.parameters(), lr=0.001)

for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 1000 == 999:    # print the batch loss every 1000 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 1000))
            running_loss = 0.0
print('Finished Training')



#--- testing

correct = 0
total = 0
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print('Accuracy of the network on the 10000 test images: %d %%' % (
    100 * correct / total))

Accuracy of the network on the 10000 test images: 59 %

class_correct = list(0. for i in range(10))
class_total = list(0. for i in range(10))
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).squeeze()
        for i in range(4):
            label = labels[i]
            class_correct[label] += c[i].item()
            class_total[label] += 1


for i in range(10):
```

```
    print('Accuracy of %5s : %2d %%' % (
        classes[i], 100 * class_correct[i] / class_total[i]))
```

```
Accuracy of plane : 64 %
Accuracy of   car : 74 %
Accuracy of  bird : 30 %
Accuracy of   cat : 36 %
Accuracy of  deer : 60 %
Accuracy of   dog : 36 %
Accuracy of  frog : 81 %
Accuracy of horse : 70 %
Accuracy of  ship : 74 %
Accuracy of truck : 67 %
```

Summary

This chapter presents the basic concepts of convolution neural networks. A convolution layer is specified by the number of input channels, the number of output channels, filter size, stride, and zero-padding. In addition to mathematical equation, the operation of convolution is usually visualized by illustrations such as Fig.2, Fig.4 and Fig.5. In PyTorch, CNN net is implemented as a class nn.Module, instead of nn.Sequentail().

References

[1] Chapter 8, "Deep Learning with PyTorch", Eli Stevens, Luca Antiga, Thomas Viehmann, Foreword by Soumith Chintala. 2020 by Manning Publications Co.

[2] https://pytorch.org/

Exercises

1. Consider a convolution operation with one feature map at input and three feature maps at output.

$$\text{Input } X = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

   a) filter matrix $W0 = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 0 \end{bmatrix}$, stride s=1, compute the output feature map Y0.

   b) Filter matrix $W1 = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \end{bmatrix}$, stride s=1, compute the output feature map Y1.

   c) Filter matrix $W2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, stride s=1, compute the output feature map Y2.

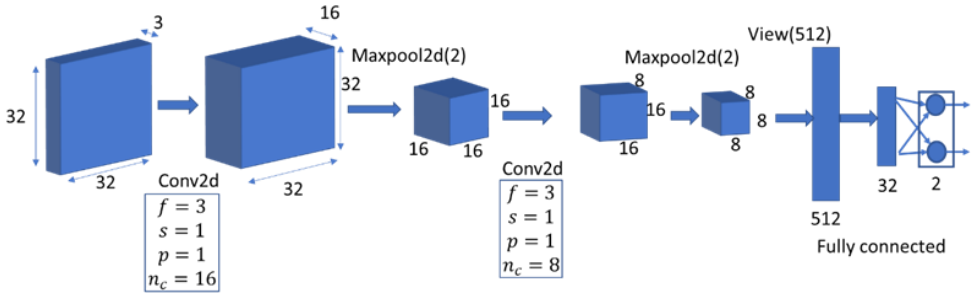
2. Find the shape of the outputs for the following convolutions. Note shape is denoted by [C, H, W] which matches PyTorch format, C is the number of channels (features), H is height, W is width.

a) Input shape [3, 64, 64], filter: f=6, s=2, p=1, n_c=10

b) Input shape [8, 17, 17], filter: f=5, s=1, p=0, n_c=10.

c) Input shape [3, 24, 24], Max pool: f=2, s=2.

3. How many parameters are there in the following neural network?



4. In this project, you will develop the LeNet-5 to recognize handwritten digits. The architecture is show below. The input image is black/white with size of 28x28. The datasets can be downloaded using torchvision.datasets.MNIST.