Chapter 8

Convolutional Neural Networks

Until now, we simply flattened 2-D images into one-dimensional vectors, and fed the vectors to a neural network which is fully connected (see the definition of "fully connected" later). Thus, the order of pixels in the vector does not matter. In fact, nearby pixels are typically correlated to each other. Therefore, the fully connected neural networks do not leverage the nature of image structure, and thus are not efficient for learning image data.

This chapter will introduce convolutional neural networks (CNNs), a powerful family of neural networks that were originally designed for computer vision and now are extended to audio, text and time series analysis and deep reinforcement learning. 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 tasks. This chapter covers:

- Motivation of a convolution layer
- The operation of convolution
- Pool layer
- LeNet-5
- Backpropagation in CNNs
- Batch normalization for CNNs
- Construction and training of CNNs using PyTorch

8.1 Architecture of Convolutional Neural Networks

8.1.1 Motivation of convolutional neural networks

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 $3 \times 32 \times 32$ (3 color channels, 32 wide, 32 high), a single fully connected node in a first hidden layer would have $3 \times 32 \times 32 = 30723$*32*32 = 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 3x1000x1000. One node in the first hidden layer requires 3,000,000 weights (connections) (ignoring 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: depth (or *channel*), width, height. For instance, a color image consists of three channels: red, green, and blue. Each channel is a 2-D tensor specifying the values for the spatial pixels. A channel is also called a *feature map*.

Fig. 8.1 illustrates a convolution layer with four convolution filters (or kernels). Each filter is a set of weights, which is used to generate one feature map at the output. The nodes in a CNN layer (can be viewed as elements in the feature maps) will only be connected to a small region of the output from the previous layer, and all of the nodes in the same feature map share the same filter but with different inputs from the previous layer. For example, the input to the layer is an image, denoted as $a^{[0]}$, a 3-dimension array or tensor. The nodes in the hidden layer, denoted as $a^{[1]}$, are arranged as a 3-dimension array too, but with a different shape in general (in this case channel is 4). Each filter is responsible to generate one channel or one feature map. Thus, the hidden layer has 4 feature maps. Now let's focus on one particular feature map, say the one generated by W1. Any node (or element) in this feature map is connected to a subregion of $a^{[0]}$ with weights W1. 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.

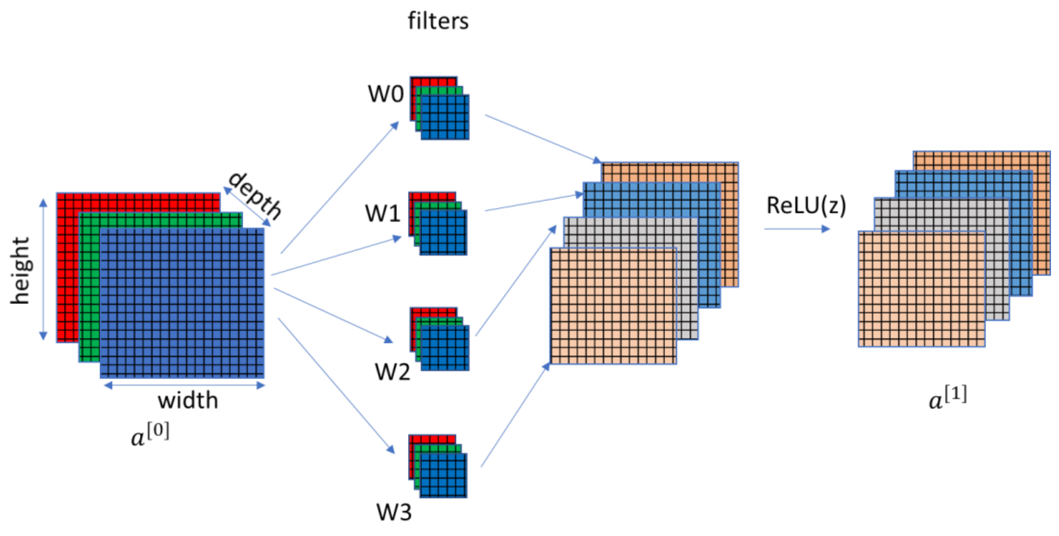


Fig.8.1 A convolution layer with 4 filters (channels)

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 (plus a bias) is computed as one data point on the feature map. All the feature maps are computed in the same way but with different filters. Like fully connected networks, an activation function (e.g. ReLU) is usually applied to the results of the convolution operations in elementwise.

8.1.2 Architecture of convolutional neural networks

A convolutional Neural Network consists of multiple layers such as the input layer, convolution layers, pooling layers, and fully connected layers, illustrated in Fig.8.2. The convolution layer applies filters to the input image to extract features, the pooling layer downsamples the extracted features to reduce computation for high-level features, and the fully connected layers make the final prediction. The network learns the optimal filters and the weights in the fully connected layers through backpropagation and gradient descent. A typical deep CNN usually includes many convolution layers and pooling layers.

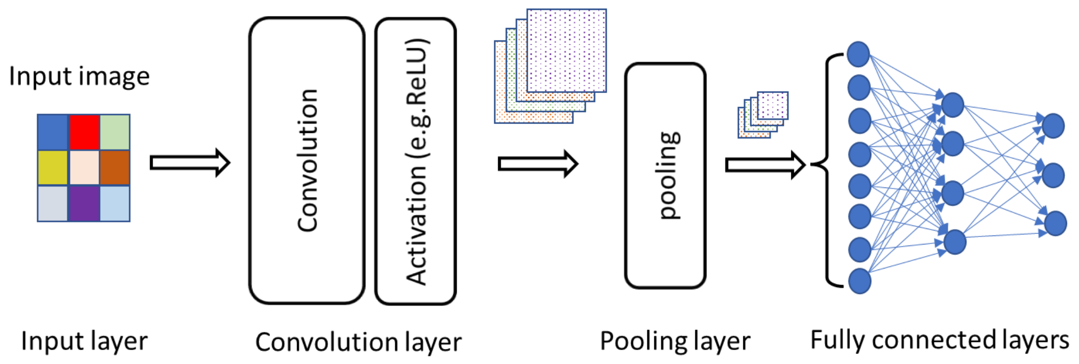


Fig.8.2 A simple convolutional neural network

8.2 Convolution Layer

8.2.1 Convolution operation

The convolution layer (CONV) uses filters that perform convolution operations as it is scanning the input with respect to its dimensions.

The filters have small widths and small heights, but the same depth as that of the input. For example, the possible filter size can be $n_c \times f \times f$, where f represents the width and the height of the filter, and n_c is the depth of the input (i.e. the number of channels of the input). The typical value of f is much smaller than the width or height of the input. In one conv layer, all the filters have the same shape, and the width is equal to the height.

During the forward propagation, we slide each filter across the whole input volume one step at a time, where each step is called a stride, and compute the dot product between the filter weights and the patch from input volume. As we slide the filters, we'll get a 2-D feature map for each filter. By stacking the resulting 2-D feature maps for all filters, we'll get a 3-D output volume with a depth equal to the number of filters.

To make it concrete, consider a numerical example of convolution, shown in Fig.8.3. For simplicity, we assume that the input X has one channel with a shape of [3,3], and the output has two channels (or feature maps). The bias is zero. The filters for two channels are W0 and W1 with a shape of [2,2]. The filters slide across the input by one pixel at a time. The output feature map in each channel has a shape of [2,2]. Fig.8.4 shows a convolution example for the input with a depth of 2 and the output with two feature maps.

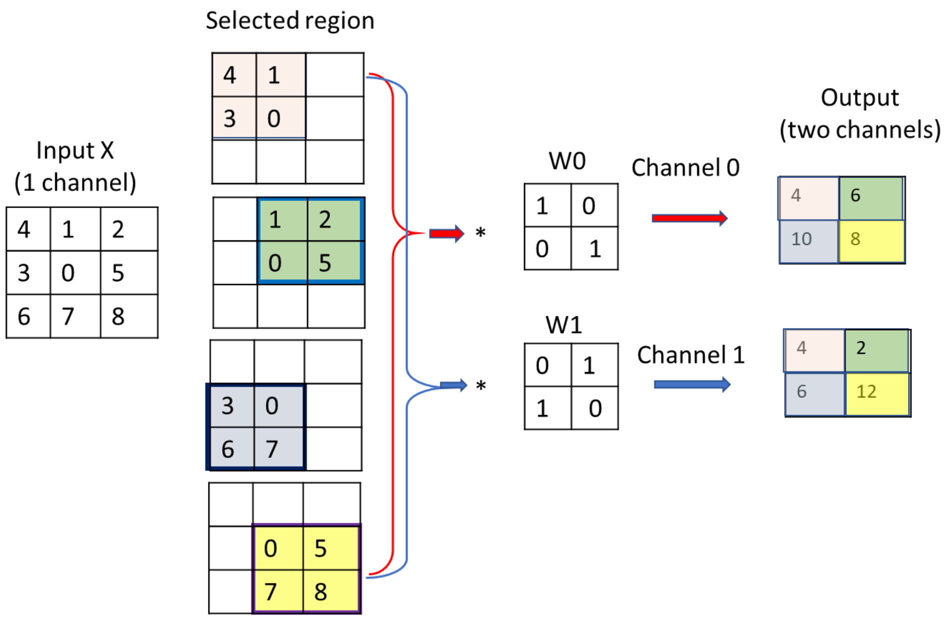


Fig.8.3 A numerical example of convolution computation: two channels, stride=1.

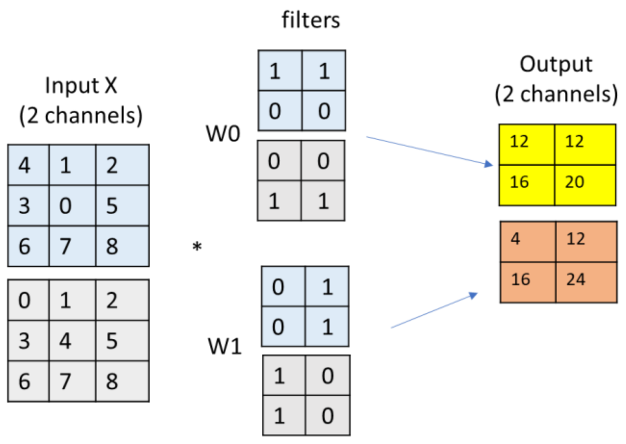


Fig.8.4 Convolution: input with 2 channels, output with 2 channels

8.2.2 Stride and zero-padding

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

1) **Filter size**, denoted by f, is the width and the height of the filter. Thus, the volume of the filter is represented by $\left(n_c^{[l-1]}, f, f\right)$, where $n_c^{[l-1]}$ is the depth of the filter matching the depth of the input data volume. For example, in Fig.8.4, $f=2$, $n_c^{[l-1]}=2$.
2) **Depth** (or channel), denoted by $n_c^{[l]}$, is the number of filters at layer l, which is also equal to the number of feature maps to be delivered. For example, in Fig.8.4, $n_c^{[l]}=2$.
3) **Stride,** denoted by s, is the step size for one slide. When computing the convolution, we start with the patch window at the upper-left corner of the input tensor, and then slide it over all locations both down and to the right. In Fig.8.3 and Fig.8.4, we slide the filters by one pixel at a time. However, sometimes, either for computational efficiency or because we wish to downsample for a smaller output data volume, we move the filter more than one pixel at a time, skipping the intermediate locations. We refer to the number of rows and columns traversed per slide as the stride. Fig.8.5 shows the case with stride of 2.

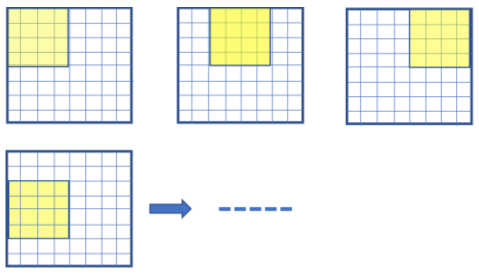


Fig.8.5 Illustration of stride=2

4) **Zero-padding**: denoted by p, specifies how many rows and columns of zeros are added around the input data volume. Sometimes it will be convenient to pad the input volume with zeros around the border. 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). Zero-padding p refers to adding p zero-rows on the top and p zero-rows on the bottom, p zero-columns on the left and p zero-columns on the right. Fig.8.6 shows zero-padding of 1. In Fig.8.3 and Fig.8.4, p=0 (no zero-padding).

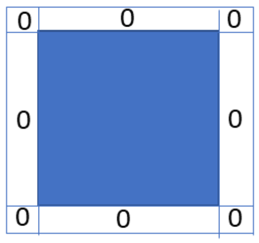


Fig.8.6 Zero-padding p=1

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

$$\left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times \left\lfloor \frac{n+2p-f}{s} + 1 \right\rfloor \times n_c^{[l]} \tag{8.1}$$

As an example, the output data in Fig.8.3 has a size $(2 \times 2 \times 2)$ with n=3, p=0, f=2, s=1, $n_c^{[l]}$=2.

We can describe a CNN by specifying the data volume and filter hyperparameters at each layer. As an example, Fig.8.7 shows a ConvNet consisting of a series of convolution layers. The ConvNet takes an image $(3 \times 39 \times 39)$ as input. The first hidden layer, defined by the filters specified in the box, delivers the output of size $(10 \times 37 \times 37)$. The two subsequent hidden layers are similarly specified with different hyperparameters. The output layer is a fully connected layer with softmax activation.

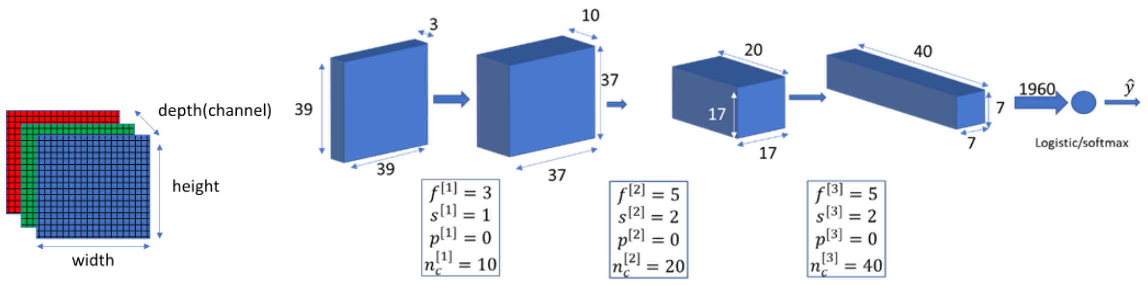


Fig.8.7 A ConvNet architecture

8.2.3 Convolution implementation: matrix multiplication

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.8.8 illustrates these connections, where each little square in the output feature maps 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 at 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. In CNNs, for any node (*i,j*) of layer *l*, its *receptive field* refers to all the nodes (from all the previous layers, shown by the yellow volume) that are involved in the calculation of this (*i,j*) node in layer *l* during the forward propagation. For example, the output nodes and their corresponding receptive fields are highlighted in the same color in Fig.8.3.

The output of the convolution can be represented by

$$z(k,i,j) = b(k) + \sum_{k'=0}^{n_c^{[l-1]}} \sum_{v=0}^{f-1} \sum_{u=0}^{f-1} x(k',i',j')W(k,k',u,v),$$

$$i' = i \times s + u,\ j' = j \times s + v \tag{8.2}$$

where

$x(k,i,j)$ is the output of the node at location *(i,j)* in feature map *k* in layer *l-1*.
$z(k,i,j)$ 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 $W(k,k',u,v)$ is the weight for the connection between the node $x(k',i\times s+u,j\times s+v)$ at layer *l-1* and the node $z(k\ i,j)$ at layer *l*.
$b(k)$ is the bias for feature map *k*.

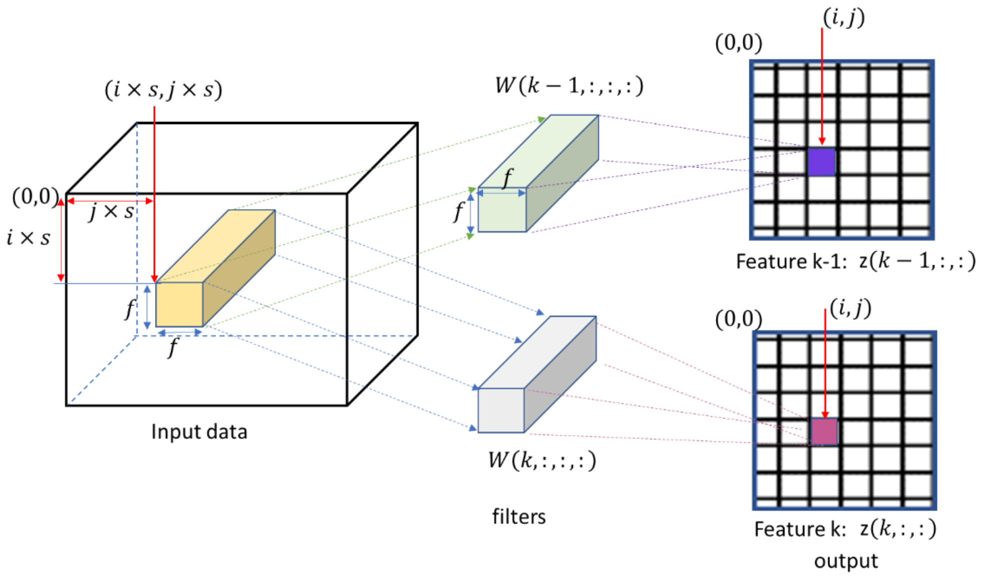


Fig. 8.8 The node connections between two layers in a convolution layer.

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 a matrix multiplication. This is visualized by an example in Fig.8.9.

1) The local regions in the input image are stretched out into **columns** in an operation commonly called *x2col*. For example, consider a filter shown in Fig.8.9. Since the input is [10x37x37] and it is to be convolved with 10x5x5 filters at stride 2, we would take one receptive field in the input and stretch it into a column vector of size 10*5*5 = 250. The convolution involves a total (37-5)/2+1 = 17 strides along both width and height, which leads to 17*17=289 receptive fields. As the result, the output matrix X_col has a size of [250 x 289], where each column corresponds to a receptive field. 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.8.9, since there are 20 filters of size [10x5x5], this would give a matrix W_row of size [20 x 250].
3) The result of a convolution is now equivalent to performing the matrix multiplication 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 [20x17x17].

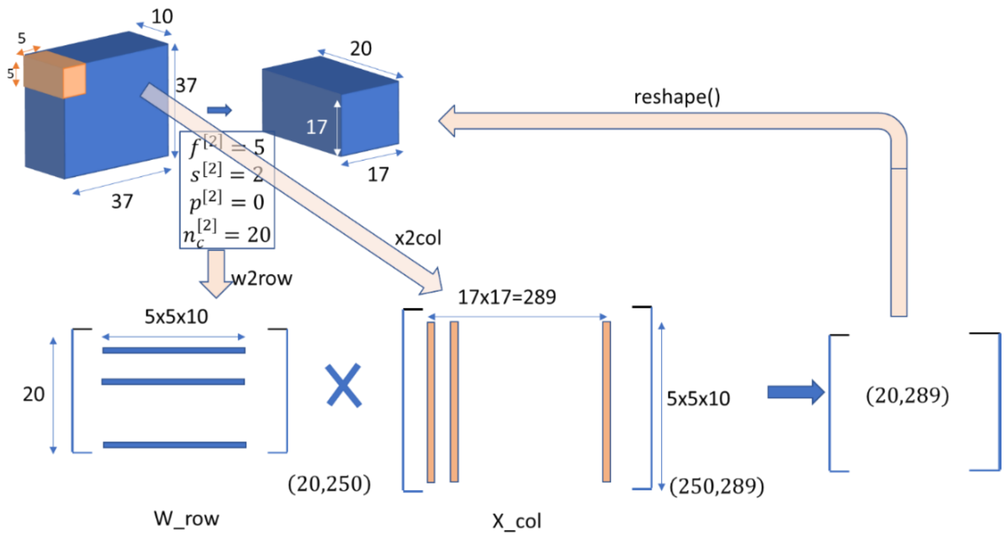


Fig.8.9 Implementation of convolution as matrix multiplication

8.3 Pooling Layer and Fully Connected Layer

8.3.1 Pooling layer (POOL)

It is common to insert a pooling layer between two Conv layers in a ConvNet. Its function is to progressively reduce the spatial size of the representation and extract the high-level features. The pooling layer operates independently on each feature map of the input and resizes it spatially, using either the MAX operation or average operation. One example 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). As the results of inserting pooling layers, the nodes in later layers are sensitive to a larger receptive field in the input images, and eventually the output of the entire neural network can learn a global representation of the input image, e.g. does it contain a dog? An example is shown in Fig.8.10. The depth dimension remains unchanged.

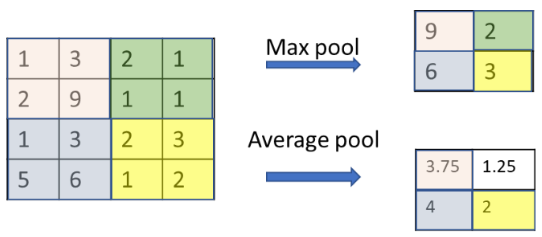


Fig.8.10 Pool layer with f=2, s=2.

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.

8.3.2 Fully connected layer (FC)

After the convolution and pooling layers, the resulting feature maps are flattened into a one-dimensional vector so that they can be passed into fully connected layers for categorization or regression. In a fully connected layer, all the nodes have full connections to all activations from the previous layer. FC layers are usually found at the end of CNN architectures and can be used to optimize objectives.

It is helpful to summarize the key points for CNN layers in Table 8.1.

	Conv layer	**Pooling layer**	**FC layer**
Description	Filter size: $n_c^{[l-1]} \times f \times f$ $n_c^{[l]}$ filters Stride: s Zero-padding: p	Max (or average): $f \times f$ Stride: $s = f$ Zero-padding: p	Input size: $n^{[l-1]}$ Output size: $n^{[l]}$
Input size	$n_c^{[l-1]} \times n \times n$	$n_c^{[l-1]} \times n \times n$	$n^{[l-1]}$
Output size	$n_c^{[l]} \times o \times o$	$n_c^{[l-1]} \times o \times o$	$n^{[l]}$
Number of parameters	$\left(n_c^{[l-1]} \times f \times f + 1\right) \times n_c^{[l]}$	0	$\left(n^{[l-1]} + 1\right) \times n^{[l]}$
Notes	$o = \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$ One bias per filter	$o = \left\lfloor \frac{n + 2p - f}{s} + 1 \right\rfloor$ Pooling in channel-wise	Input is flattened. One bias per node

8.3.3 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 1998. The LeNet-5 was originally proposed to recognize hand-written digits. The architecture is illustrated by Fig.8.11. In the original version of LeNet-5, the activation functions for conv1, conv2, FC3(fully connected), and FC4(fully connected) are sigmoid functions, and pool layers use averaging pool without extra activation function. Here, we use ReLU() to replace sigmoid activation and use max pooling for a variation of LeNet-5.

The input of LeNet-5 is a 3D tensor of image with a shape (3, 32, 32) (or (3, 28,28) with zero-padding p=2). The first convolutional layer has 6 output channels with filter size (5,5) for each channel and ReLU activation. The output (6, 28, 28) of ReLU activation is reduced to (6, 14, 14) by the subsequent MAX pooling layer. The second convolutional layer has 16 channels with filter

(5,5) and stride =1. After the ReLU activation, the data pass through a Max pooling layer with output (16, 5, 5). Then we flatten this output (16, 5, 5) to a vector of 400 elements as the input of the subsequent fully-connected layer with 120 nodes. Two more fully-connected layers are designed to deliver 84 and 10 outputs, respectively. The output of the last layer with softmax activation corresponds to the predicted probability vector of the 10 classes.

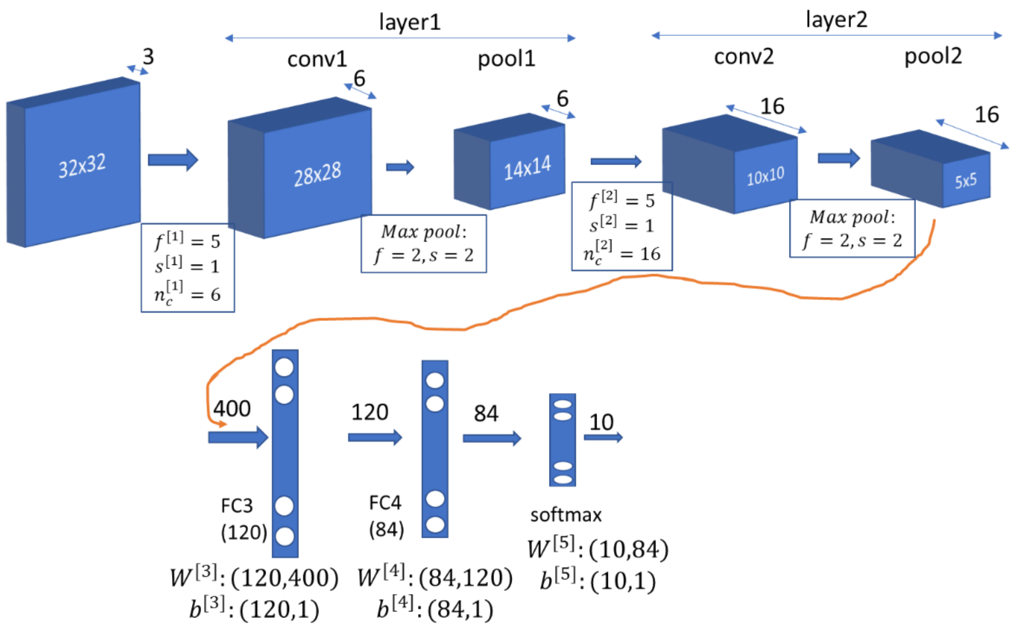


Fig. 8.11 Architecture of LeNet-5

8.4 Backpropagation in CNNs

We train CNNs in the same way as we do for regular fully connected neural networks. Specifically, we define a loss function, and then calculate the gradients of the loss with respect to (or. w.r.t.) the learnable parameters and update the parameters with the gradient descent algorithm. However, the backpropagation in CNNs requires the special consideration of: 1) the shared and local computation in the convolution layers; and 2) the pooling operation in the pooling layers. For the backpropagation, we have to compute the partial derivatives for conv layers and pooling layers.

8.4.1 Backpropagation in conv layers

The output of a convolution operation is usually represented as a 3-D tensor, denoted by $z(k, i, j)$, where k is the feature map (or channel) index, and (i, j) represents a datapoint location in the k feature map. Each datapoint in the output, $z(k, i, j)$, can be calculated by equation (8.2). During the backpropagation, we calculate the derivatives of the loss with respect to the parameters and the input of each layer.

Now let's focus on a particular conv layer. Suppose we are given the derivatives of the loss w.r.t. the output of the conv layer, denoted as $\frac{\partial L}{\partial z(k,i,j)}$, and we want to calculate the derivatives of the loss w.r.t. the parameters and the input of the layer, denoted as $\frac{\partial L}{\partial W(k,c,u,v)}$ and $\frac{\partial L}{\partial x(c,m,n)}$, where d refers to the channel dimension, and both (u, v) and (m, n) refer to the height-width dimension.

A) Partial derivatives w.r.t filters

Consider one of the filters, say the k-th filter $W(k, :, :, :)$, i.e. k is viewed as a constant, as shown in Fig.8.12. Since each data point $z(k, i, j)$ is the linear combination of its receptive field and the filter, each weight in the filter impacts the loss through all datapoints in the k-th feature map $z(k, :, :)$, By the chain rule, we have

$$\frac{\partial L}{\partial W(k,c,u,v)} = \sum_i \sum_j \frac{\partial L}{\partial z(k,i,j)} \cdot \frac{\partial z(k,i,j)}{\partial W(k,c,u,v)} \tag{8.3a}$$

$$\frac{\partial L}{\partial b(k)} = \sum_i \sum_j \frac{\partial L}{\partial z(k,i,j)} \cdot \frac{\partial z(k,i,j)}{\partial b(k)} \tag{8.3b}$$

According (8.2), we have

$$\frac{\partial z(k,i,j)}{\partial W(k,c,u,v)} = x(c, i \times s + u, j \times s + v) \tag{8.4a}$$

$$\frac{\partial z(k,i,j)}{\partial b(k)} = 1 \tag{8.4b}$$

where s is the stride. Thus, the derivatives of the loss w.r.t the parameters can be calculated by

$$\frac{\partial L}{\partial W(k,c,u,v)} = \sum_i \sum_j \frac{\partial L}{\partial z(k,i,j)} \cdot x(c, i \times s + u, j \times s + v) \tag{8.4a}$$

$$\frac{\partial L}{\partial b(k)} = \sum_i \sum_j \frac{\partial L}{\partial z(k,i,j)} \cdot \frac{\partial z(k,i,j)}{\partial b(k)} = \sum_i \sum_j \frac{\partial L}{\partial z(k,i,j)} \tag{8.5b}$$

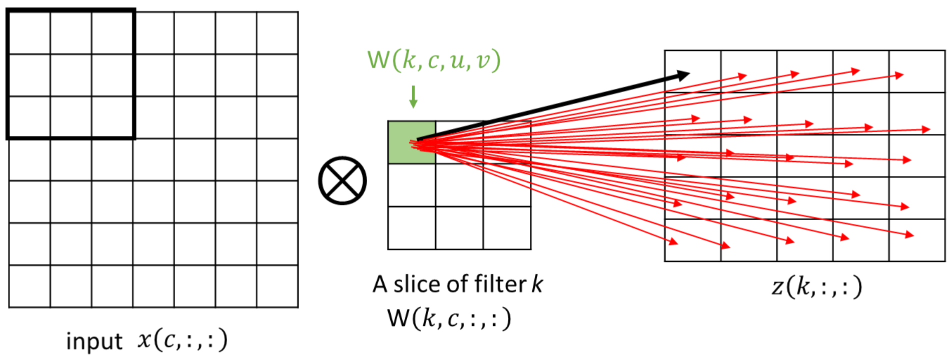


Fig.8.12 Each weight in filters effects all outputs of the convolution

The result in (8.4a) can be interpreted as the convolution between the input $x(:, :, :)$ and the filter $\frac{\partial L}{\partial z(:,:,:)}$. When the stride $s = 1$, (8.4a) can rewritten as

$$\frac{\partial L}{\partial W(k,c,u,v)} = \sum_i \sum_j x(c, i + u, j + v) \cdot \frac{\partial L}{\partial z(k,i,j)} \tag{8.6}$$

One can verify that (8.6) is the definition of the 2-D convolution between $x(c,:,:)$ and the filter $\frac{\partial L}{\partial z(k,:,:)}$, with a stride of 1. Thus, the partial derivatives of the loss w.r.t the weights can be calculated by the convolution illustrated in Fig.8.13.

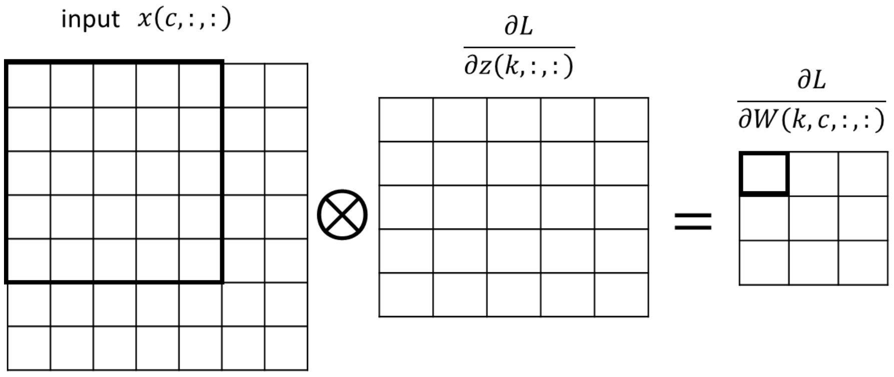

input $x(c,:,:)$
$\frac{\partial L}{\partial z(k,:,:)}$
$\frac{\partial L}{\partial W(k,c,:,:)}$

Fig.8.13 Computation of the partial derivatives of the loss w.r.t weights when stride s=1.

More generally, by examining (8.4a) carefully, we find out that the partial derivatives of the loss w.r.t weights can be calculated as the convolution (stride =1) between the input $x(c,:,:)$ and the dilated version of $\frac{\partial L}{\partial z(k,:,:)}$, illustrated in Fig.8.14. The dilated version of $\frac{\partial L}{\partial z(k,:,:)}$ is obtained by inserting $s-1$ rows (columns) of zeros between any two rows (columns) of the matrix $\frac{\partial L}{\partial z(k,:,:)}$. Thus, the derivatives of the loss w.r.t the parameters can be calculated by

$$\frac{\partial L}{\partial W(k,c,:,:)} = Convolution\left(x(c,:,:), Dilation\left[\frac{\partial L}{\partial z(k,:,:)}\right]\right) \tag{8.7}$$

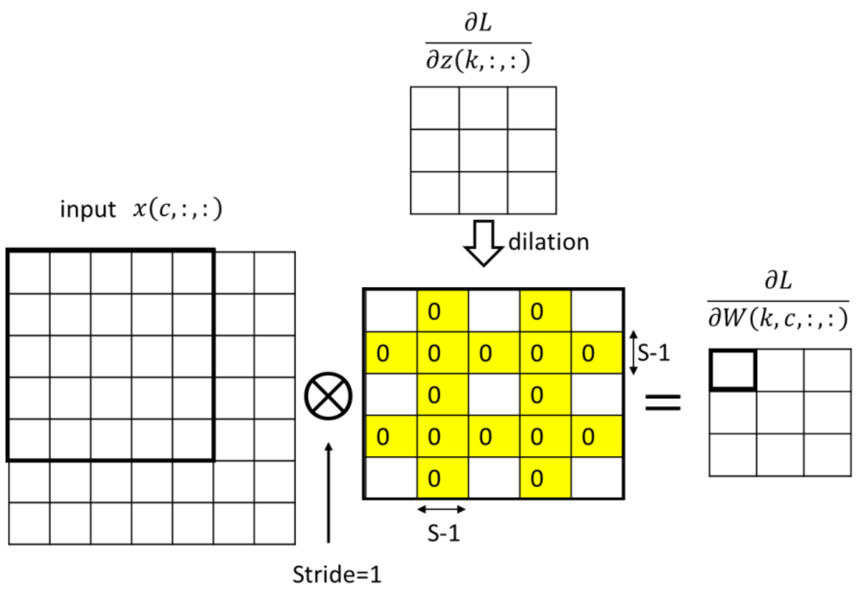

$\frac{\partial L}{\partial z(k,:,:)}$
dilation
input $x(c,:,:)$
0
S-1
S-1
Stride=1
$\frac{\partial L}{\partial W(k,c,:,:)}$

Fig.8.14 Computation of the partial derivatives of the loss w.r.t weights

B) Partial derivatives w.r.t inputs

In the backpropagation, we are given the derivatives of the loss w.r.t the output of the conv layer, denoted as $\frac{\partial L}{\partial z(k,i,j)}$, and we want to calculate the derivatives of the loss w.r.t the input of the layer, denoted as $\frac{\partial L}{\partial x(c,m,n)}$. According to the definition of convolution (8.2), one data point in the input affects multiple data points in all output feature maps $z(k,:,:)$ for all k. Fig.8.15 shows the case where the input size is $c \times 7 \times 7$, three filters with each filter size $c \times 3 \times 3$, stride s=1. Thus, the output feature size is $3 \times 5 \times 5$. As an example, one input data point highlighted by red affects the output region highlighted by the red frames.

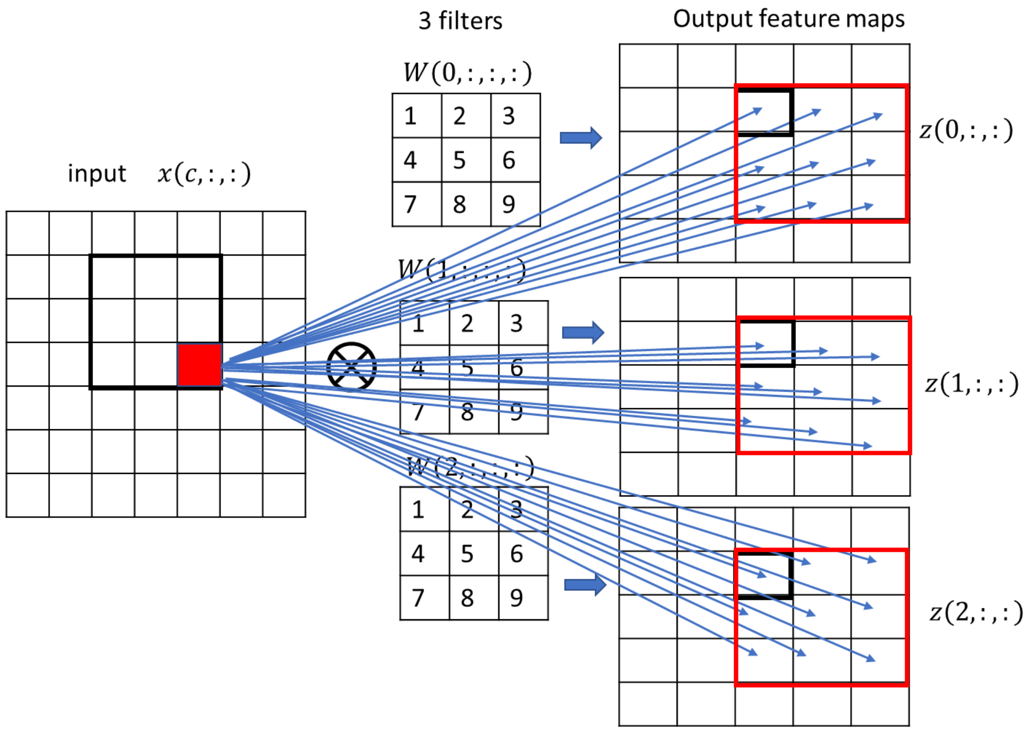


Fig.8.15 How one input data point affects multiple output data points (stride=1)

Applying the chain rule to (8.2), we have

$$\frac{\partial L}{\partial x(c,i\times s+u,j\times s+v)} = \sum_k \sum_i \sum_j \frac{\partial L}{\partial z(k,i,j)} \cdot \frac{\partial z(k,i,j)}{\partial x(c,i\times s+u,j\times s+v)} = \sum_k \sum_i \sum_j \frac{\partial L}{\partial z(k,i,j)} \cdot W(k,c,u,v) \quad (8.8)$$

For simplicity, we consider the stride s=1. Then, (8.8) can rewritten as

$$\frac{\partial L}{\partial x(c,i+u,j+v)} = \sum_k \sum_i \sum_j \frac{\partial L}{\partial z(k,i,j)} \cdot W(k,c,u,v) \quad (8.9)$$

Now let $m = i + u$, and $n = j + v$, we have

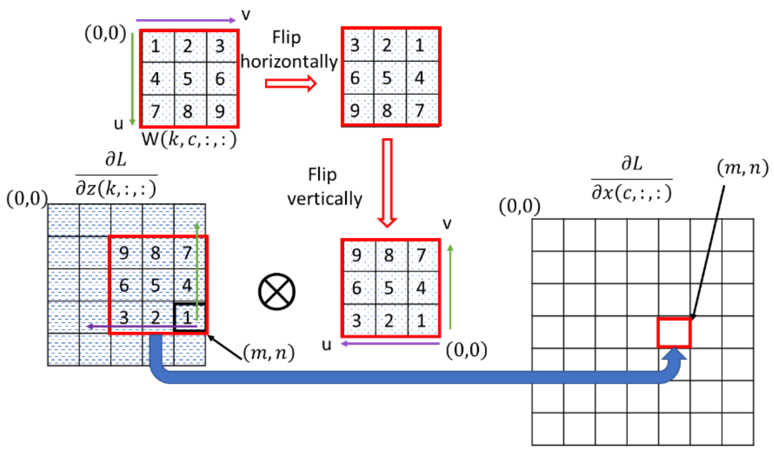
$$\frac{\partial L}{\partial x(c,m,n)} = \sum_k \sum_{u=0}^{f-1} \sum_{v=0}^{f-1} \frac{\partial L}{\partial z(k,m-u,n-v)} \cdot W(k,c,u,v) \quad (8.10)$$

Note that $\frac{\partial L}{\partial z(k,m-u,n-v)}$ will be treated as zero if $m - u$ or $n - v$ is negative or more than the maximal index of the output feature matrix. (8.10) shows that, to compute the derivative at

$x(c, m, n)$, we simultaneously scan the filter and the derivative map of z, and add their component-wise products, illustrated in Fig.8.16. Incrementing u and v scans the filter starting at (0,0), top to bottom and left to right, respectively. At the same time, it scans the derivative map of z starting at (m, n), bottom to top and right to left, respectively.

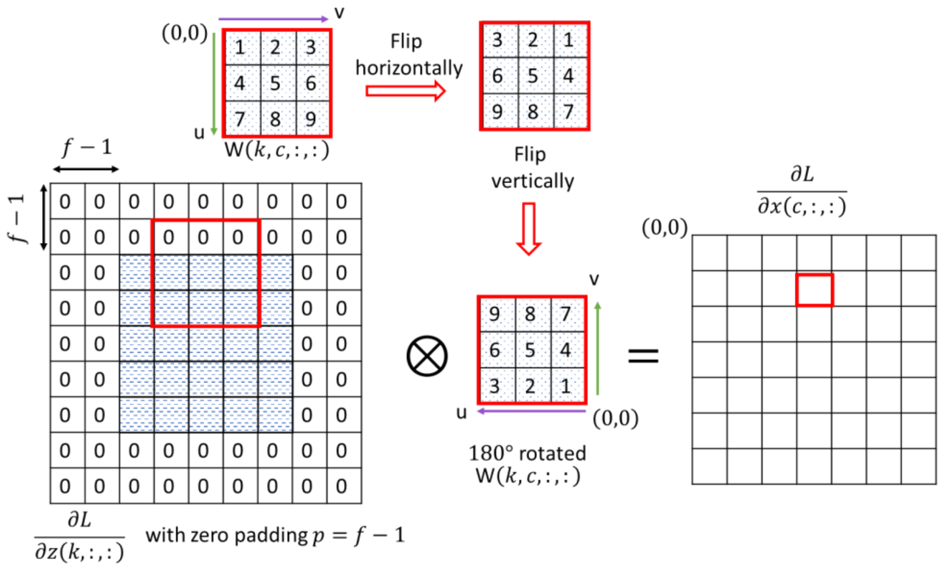
Thus, equivalently, the partial derivatives of the loss w.r.t the input at channel c, specified by (8.10), can be implemented as the ***full*** 3-D convolution between the derivatives of the loss w.r.t the output feature map $z(:,:,:)$ and the ***rotated*** filters $W(:, c, :, :)$ for the input channel c. This is illustrated in Fig.8.17. The full convolution requires $f - 1$ padding zeros around $\frac{\partial L}{\partial z(k,:,:)}$ for all k. We rotate the filters $W(k, c, :, :)$ for all k, by flipping it horizontally and then vertically. Note that the full convolution is performed in three dimensions, including k dimension. The derivatives of the loss w.r.t the input of the layer can be represented by

$$\frac{\partial L}{\partial x(c,:,:)} = full\ Convolution\left(\left[\frac{\partial L}{\partial z(:,:,:)}\right], 180°\ rotate\ [W(:, c, :, :)]\right) \tag{8.11}$$



(0,0)
v
u
W(k, c, :, :)
Flip horizontally
Flip vertically
$\frac{\partial L}{\partial z(k,:,:)}$
(m, n)
$\frac{\partial L}{\partial x(c,:,:)}$

Fig.8.16 Illustration of the computation of (8.9)



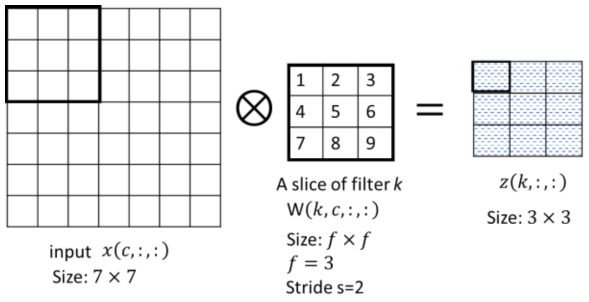
(0,0)
v
u
W(k, c, :, :)
Flip horizontally
Flip vertically
f − 1
$\frac{\partial L}{\partial z(k,:,:)}$
with zero padding $p = f - 1$
180° rotated W(k, c, :, :)
$\frac{\partial L}{\partial x(c,:,:)}$

Fig.8.17 The computation of the partial derivatives of the loss w.r.t the inputs (stride =1)

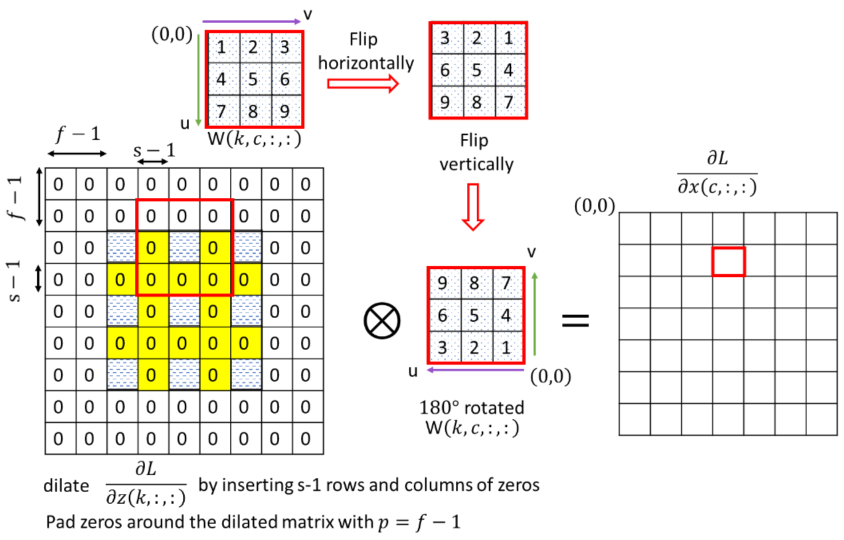
Now we consider a convolution layer with stride $s \geq 1$. To compute (8.8), we follow the same way illustrated in Fig.8.17, except that we need to dilate the output gradient matrices by inserting s-1 zeros between each row and column. Thus, the derivatives of the loss w.r.t the input of the layer can be calculated by

$$\frac{\partial L}{\partial x(c,:,:)} = full\ Convolution\left(dilation\left[\frac{\partial L}{\partial z(:,:,:)}\right], 180°\ rotate\ [W(:,c,:,:)]\right) \tag{8.12}$$

As an example, Fig.8.18 shows a case where the input feature map size is 7×7, filter size is 3×3 with stride $s = 2$. As the result, the output feature map size is 3×3, shown in Fig.8.18 (a). To calculate the backpropagation, i.e., $\frac{\partial L}{\partial x(c,:,:)}$, as shown in Fig.8.18(b), we dilate the output gradients $\frac{\partial L}{\partial z(k,:,:)}$ by inserting s-1 zeros between rows and columns (highlighted as yellow), and then pad $f - 1$ zero rows and columns around the dilated output gradients for a full convolution. Finally, the dilated and zero-padded output gradients will be convoluted with the 180°rotated the filters $W(k, c, :, :)$. The results for all k will be summed up.



(a) Forward pass



(0,0)
v
u
W(k, c, :, :)
Flip horizontally
Flip vertically
180° rotated W(k, c, :, :)
f − 1
s − 1
∂L/∂x(c, :, :)
dilate ∂L/∂z(k, :, :) by inserting s-1 rows and columns of zeros
Pad zeros around the dilated matrix with p = f − 1

(b) Backpropagation

Fig.8.18 The partial derivatives of the loss w.r.t the input for a convolution layer with stride s.

8.4.2 Backpropagation in pooling layers

Since the pooling layers do not have any weights, we only need to calculate the input gradients, i.e., the partial derivatives of the loss w.r.t the input, given the output gradients.

A) Max pooling layers

The forward pass of a max pooling layer, with filter size $f \times f$ and stride s, can be described as

$$z(c,i,j) = \max_{\substack{i\times s \le m \le i\times s+f-1 \\ j\times s \le n \le j\times s+f-1}} \{x(c,m,n)\} \tag{8.13}$$

Note that s is usually equal to f. Equation (8.13) can be represented equivalently by

$$z(c,i,j) = x\big(c, p(i,j)\big) \tag{8.14a}$$

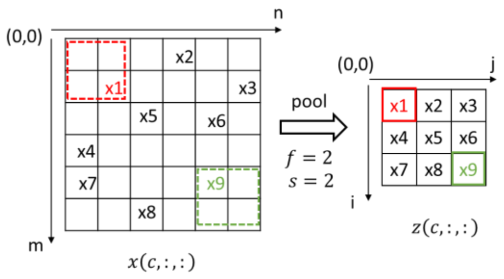
where $p(i,j)$ is the position of the maximal value in $x(c,:,:)$ which leads to $z(c,i,j)$

$$p(i,j) = \operatorname*{argmax}_{\substack{i\times s \le m \le i\times s+f-1 \\ j\times s \le n \le j\times s+f-1}} \{x(c,m,n)\} \tag{8.14b}$$

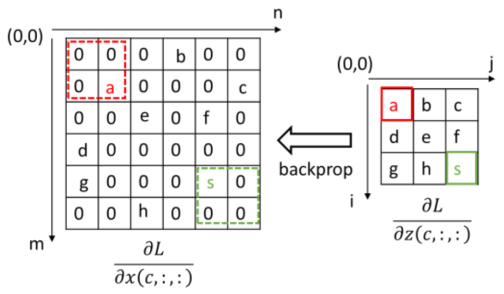
Since only the maximal value in each sliding filter position is passed to the output, the gradients at the z map will be propagated to those max-value positions. Thus, we have the backpropagation for a max pooling layer

$$\frac{\partial L}{\partial x(c,m,n)} = \begin{cases} \frac{\partial L}{\partial z(c,i,j)} & if\ (m,n) = p(i,j) \\ 0 & otherwise \end{cases} \tag{8.15}$$

An example is illustrated in Fig.8.19.



(a) Forward pass. x1,…,x9 represent the maximal values in the pooling windows



(0,0)
n
m
$\frac{\partial L}{\partial x(c,:,:)}$
backprop
(0,0)
j
i
$\frac{\partial L}{\partial z(c,:,:)}$

0	0	0	b	0	0
0	a	0	0	0	c
0	0	e	0	f	0
d	0	0	0	0	0
g	0	0	0	s	0
0	0	h	0	0	0

a	b	c
d	e	f
g	h	s

(b) Backpropagation of a max pooling layer

Fig.8.19 The forward pass and the backpropagation for a max pooling layer with $f = 2, s = 2$.

B) Average pooling layers

An average pooling layer, with the filter size $f \times f$ and stride s, can be described by

$$z(c,i,j) = \frac{1}{f\times f}\sum_{m=i\times s}^{i\times s+f-1}\sum_{n=j\times s}^{j\times s+f-1} x(c,m,n) \tag{8.16}$$

By the chain rule, the derivative of the loss w.r.t the input can be calculated by

$$\frac{\partial L}{\partial x(c,m,n)} = \frac{\partial L}{\partial z(c,i,j)}\frac{\partial z(c,i,j)}{\partial x(c,m,n)} = \frac{\partial L}{\partial z(c,i,j)}\cdot\frac{1}{f\times f}$$

$$\text{where } i\times s \le m \le i\times s+f-1 \text{ and } j\times s \le n \le j\times s+f-1 \tag{8.17}$$

Equation (8.17) shows that the derivate at one point in the output feature map is evenly divided and distributed to the corresponding region in the input feature map.

8.5 Batch Normalization for CNNs

In Section 6.6.2, we introduced the concept of batch normalization, with an emphasis on fully connected layers. In the batch normalization, we normalize each feature signal differently, which requires two learnable parameters (β,γ) per feature signal. However, since all elements in the same feature map share weights connected to the previous layer in CNNs, the batch normalization in CNNs will be slightly different from that in fully connected layers.

Now we treat the batch normalization for convolutional layers. Fig.8.20 shows the batch process with a batch normalization in a convolutional layer. A batch is represented as a 4-dimensional tensor. The batch before normalization is denoted by $X[B,C,H,W]$, and the normalized batch is denoted by $Y[B,C,H,W]$, where B,C,H, and W indicate batch, channel, height, and width dimensions, respectively.

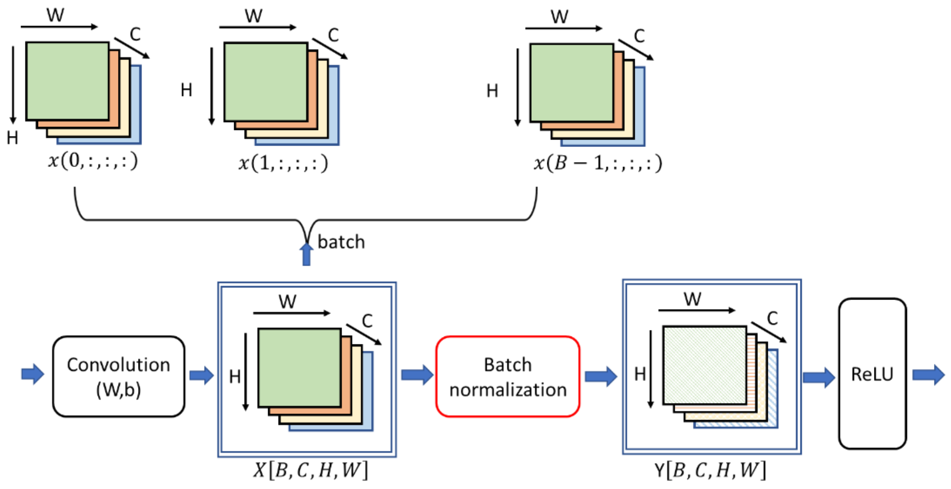


Fig.8.20 Batch process with batch normalization in a convolutional layer

In a convolutional layer, different elements of the same feature map, at different locations, denoted as $x(:,c,:,:)$, are normalized in the same way (using the same mean, the same variance, and the same learnable parameters β,γ), that is, one normalization per channel, instead of per node for fully connected layers. The batch normalization for one channel (or feature map), say channel c, can be described by the following equations.

$$\text{Mini-batch mean: } \mu_c = \frac{1}{B \cdot H \cdot W} \sum_{b=0}^{B-1} \sum_{h=0}^{H-1} \sum_{w=0}^{W-1} x(b,c,h,w) \qquad (8.18a)$$

$$\text{Mini-batch variance:} {\sigma_c}^2 = \frac{1}{B \cdot H \cdot W} \sum_{b=0}^{B-1} \sum_{h=0}^{H-1} \sum_{w=0}^{W-1} (x(b,c,h,w) - \mu_c)^2 \qquad (8.18b)$$

$$\text{Normalize: } \hat{x}(b,c,h,w) = \frac{x(b,c,h,w) - \mu_c}{\sqrt{{\sigma_c}^2 + \epsilon}}$$

$$\text{where } \ b \in [0, B-1], h \in [0, H-1], w \in [0, W-1] \qquad (8.18c)$$

$$\text{Scale and shift: } y(b,c,h,w) = \gamma_c \cdot \hat{x}(b,c,h,w) + \beta_c$$

$$\text{where } \ b \in [0, B-1], h \in [0, H-1], w \in [0, W-1] \qquad (8.18d)$$

(8.18a) and (8.18b) calculate the mean and the variance for all elements in channel c across the batch. (8.18c) normalizes the batch data for channel c. Note that ϵ is a small positive number to avoid dividing by zero. Two learnable parameters γ_c and β_c in (8.18d) are used to scale and shift the batch data, and thus to improve the representation ability of the network. $y(b,c,h,w)$ in (8.18d) is the result of batch normalization at the location (h,w) in the channel c for the image b. Fig.8.21 illustrates a batch normalization for a batch with four channels.

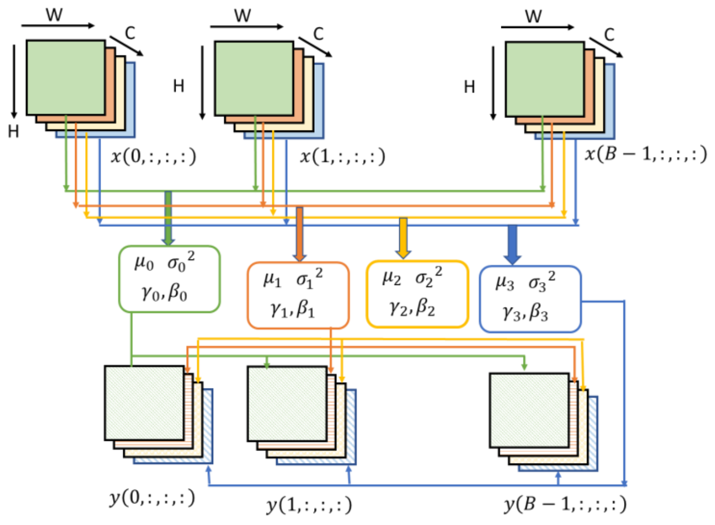


Fig.8.21 Batch normalization: normalize all elements in one channel by the same parameters.

8.6 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 effort is required to deal with convolution operations and pooling operations at matrix level. PyTorch provides all we need to build and train CNNs in a very efficient way.

In this section, we will give a step-by-step tutorial on how to develop and train a convolution neural network. Compared to LeNet-5, our CNN has smaller convolution filters and a smaller number of channels, and simpler fully-connected layers for 2-class classification, as shown in Fig.8.22.

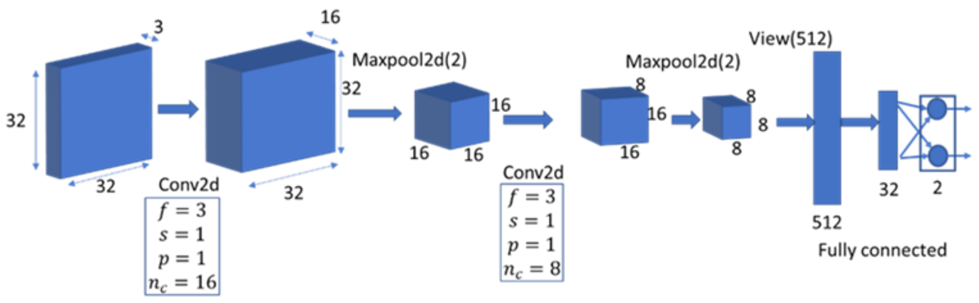


Fig.8.22 A simple CNN

8.6.1 Dataset

We generate a 2-class dataset (including train and test) from CIFAR10 dataset. 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).

The two selected classes are: airplane and bird, which are labeled as “0” and “1”, respectively in our dataset. First, we import the required packages, and then generate training set cifar2 and test set cifar2_test from CIFAR10.

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

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, transform=tra
nsform)
cifar10_test = datasets.CIFAR10(data_path, train=False, download=False,transfor
m=transform )

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)
#cifar2
```

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

8.6.2 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 important modules for CNNs: *nn.Conv2d*() and *nn.MaxPool2d*() from *torch.nn* package, and functions from *torch.nn.functional* package. We use the class *torch.nn.Module* to construct our own CNNs by instantiating modules (e.g. *nn.Conv2d*()) and/or calling functions (e.g. *nn.functional.max_pool2d*()).

Conv2d()

The torch.nn package 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 channels, 16 output channels, filter size 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 [batch_size, C, H,W] will be determined by a few relevant hyperparameters. 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

Consider a CNN shown in Fig.8.22 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. The CNN can be defined as the following class ***Net***().

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()

        #define layers with learnable parameters
        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)          # function
        out = F.max_pool2d(out,2)      # function
        out = self.conv2(out)
        out = torch.tanh(out)          # function
        out = F.max_pool2d(out,2)      # function
        out = out.view(-1, 8 * 8 * 8) # function
        out = self.fc1(out)
        out = torch.tanh(out)          # function
        out = self.fc2(out)
        return out
```

We define our CNN by a class, named *Net*, which is a subclass of *nn.Module*. The class *Net*, consists of two functions: __*init*__(self) and *forward*(self, x). The first one, __*init*__(self) is the constructor of the class, where we typically define some submodules, e.g. *nn.Conv2d*(), *nn.Linear*(), and assign these submodules to *self* for use in the *forward* function.

The *forward* (self, x) function is the place where we define our neural network forward computation, taking x as the input and returning output. Typically, our computation will use submodules defined in __*init*__(self) and functions defined in the package *torch.nn.functional*.

torch.nn.functional

In the function *forward* (self, x), some computations are implemented by modules from the package *torch.nn*, such as *conv1*, *conv2*, *fc1* and *fc2* while others are implemented by torch functions such as *torch.tanh* () or functions from the package *torch.nn.functional*, such as *F.max_pool2d* (), though the corresponding modules for tanh() activation and max pool operation are available as modules from the package *torch.nn*: *nn.Tanh*() and *nn.MaxPool2d*(2), respectively. The reason is explained below.

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, which could be carried by the submodules with no learning parameters (e.g. nn.Tanh(), nn.MaxPool2d(2)), are favorably implemented in *functional* format.

Indeed, *torch.nn.functional* provides many functions that work like the modules we find in nn. But a function takes both the data input and learning parameters as arguments when the function is called. 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 of input arguments."

Therefore, in the function *forward* (self, x), the computations without learning parameters (e.g. activation, max or average pooling) are usually implemented by functions while the computations involving learning parameters (e.g. convolution, linear combination) are implemented by modules.

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

```
model = Net()
numel_list = [p.numel() for p in model.parameters()]
sum(numel_list), numel_list
       (18090, [432, 16, 1152, 8, 16384, 32, 64, 2])
print(img.unsqueeze(0).shape)
model(img.unsqueeze(0))
       torch.Size([1, 3, 32, 32])
       tensor([[ 0.0510, -0.0475]], grad_fn=<AddmmBackward>)
```

8.6.3 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:
            outputs = model(imgs)    # imgs:[batch_size,C,H,W]accepted by conv2d
            loss = loss_fn(outputs, labels)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            loss_train += loss.item()
        if epoch == 1 or epoch % 10 == 0:
            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,
)
```

```
2023-09-05 15:44:20.501572 Epoch 1, Training loss 0.643967665684451
2023-09-05 15:46:01.669627 Epoch 10, Training loss 0.37506938341316903
2023-09-05 15:47:47.958945 Epoch 20, Training loss 0.32019150741161057
2023-09-05 15:49:29.642442 Epoch 30, Training loss 0.2998704523037953
2023-09-05 15:52:55.265520 Epoch 40, Training loss 0.2842075728876576
2023-09-05 15:54:45.228890 Epoch 50, Training loss 0.2652401124026365
2023-09-05 15:56:25.186468 Epoch 60, Training loss 0.24640889541738353
2023-09-05 15:58:05.560932 Epoch 70, Training loss 0.23124178608132015
2023-09-05 15:59:46.228611 Epoch 80, Training loss 0.21609638830658737
2023-09-05 16:01:26.292905 Epoch 90, Training loss 0.20230945754962362
2023-09-05 16:03:07.255794 Epoch 100, Training loss 0.18751443300847034
```

Based on the training loss displayed above, we may increase the number of training epochs to further reduce the training loss for a better classification performance if no overfitting.

8.6.4 Testing the trained model

Now we validate the model by calculating the classification accuracies on the training set and the test set.

```
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():
            for imgs, labels in loader:
                outputs = model(imgs)
                _, predicted = torch.max(outputs, dim=1)
                total += labels.shape[0]
                correct += int((predicted == labels).sum())
        print("Accuracy {}: {:.2f}".format(name , correct / total))


validate(model, train_loader, test_loader)
```

```
Accuracy train: 0.92
Accuracy test: 0.89
```

8.6.5 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 along with the trained parameters without re-training.

Save

```
data_path = '../torch_tutorial/data'
torch.save(model.state_dict(), data_path + 'parameters.pt')
```

Load

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

```
  <All keys matched successfully>
```

```
print(img.unsqueeze(0).shape)
loaded_model(img.unsqueeze(0))
```

```
torch.Size([1, 3, 32, 32])
tensor([[-0.8502,  1.5916]], grad_fn=<AddmmBackward>)
```

8.6.6 CIFAR-10 Image Classifier

In this section, we will implement LeNet-5 to classify 10 classes in dataset CIFAR-10.

1) Import basic packages.

```
import torch
import torchvision
import torchvision.transforms as transforms
import matplotlib.pyplot as plt
import numpy as np
import datetime
```

2) Prepare trainloader and testloader

We specify a transform for tensor and normalization, and then download the datasets from CIFAR10, and finally create trainloader and testloader with a specified batch size (e.g. 4).

```
batch_size=32
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=True, transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size,
                    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=batch_size,
                       shuffle=False, num_workers=2)
```

3) Visualize image examples

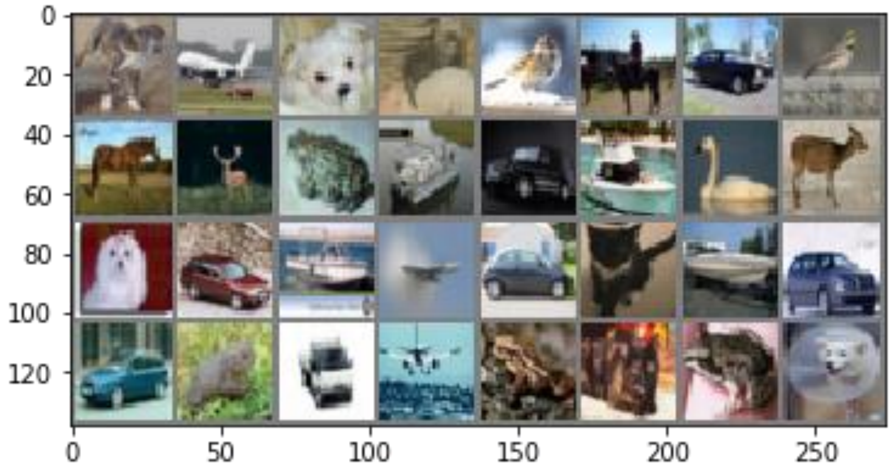
To further understand the datasets and data loaders, we get an item (or batch) from trainloader and display all images and labels in the batch.

```
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
def imshow(img):
    img = img / 2 +0.5   # unnormalize
    plt.imshow(img.permute(1,2,0))
    plt.show()

# get some random training images
dataiter= iter(trainloader) # return an iterator assigned to dataiter
images, labels = dataiter.next() # get the current iteration on dataiter: [batc
h_size, C, H, W]

# show images in one batch
imshow(torchvision.utils.make_grid(images))

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



```
 dog   plane dog   cat   bird  horse  car  bird
 horse deer  frog  ship  car   ship  bird deer
 dog   car   ship  plane car   cat   ship  car
 car   frog  truck plane frog   cat  frog  dog
```

4) Define LeNet-5 architecture

The architecture of LeNet-5 is illustrated in Fig.8.11. Here, we add two batch normalization layers to the architecture (see the details in the following code).

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

class Net(nn.Module):
    def __init__(self):
        # define self functions for all layers with learning parameters
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3,6,5)
        # define function sefl.conv1 for Conv layer1

        self.bn1 = nn.BatchNorm2d(6)
        self.conv2 = nn.Conv2d(6,16,5)
        self.bn1 = nn.BatchNorm2d(6) # batch normalization
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120,84)
        self.bn2 = nn.BatchNorm1d(84) # batch normalization
        self.fc3 = nn.Linear(84,10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = F.max_pool2d(x,2,2)
        x = self.bn1(x) # batch normalization
        x = self.conv2(x)
        x = F.relu(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.bn2(x) # batch normalization
        x = self.fc3(x)
        return x
net = Net()
```

5) Train LeNet-5

```
import torch.optim as optim
criterion = nn.CrossEntropyLoss()
#optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
optimizer = optim.Adam(net.parameters(), lr=0.001)
for epoch in range(5):  # 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 % 100 == 99:    # print every 100 mini-batches
            print('[%d, %5d] loss: %.3f' %
                  (epoch + 1, i + 1, running_loss / 100))
            running_loss = 0.0
```

```
print('Finished Training')
```

```
[1,   100] loss: 1.898
[1,   200] loss: 1.657
[1,   300] loss: 1.584
[1,   400] loss: 1.502
[1,   500] loss: 1.470
[1,   600] loss: 1.459
[1,   700] loss: 1.427
[1,   800] loss: 1.375
[1,   900] loss: 1.391
[1,  1000] loss: 1.331
[1,  1100] loss: 1.346
[1,  1200] loss: 1.336
[1,  1300] loss: 1.356
[1,  1400] loss: 1.321
[1,  1500] loss: 1.319
[2,   100] loss: 1.223
[2,   200] loss: 1.204
…
[5,  1400] loss: 0.991
[5,  1500] loss: 0.988
Finished Training
```

6) Test the trained LeNet-5

First, we calculate the overall accuracy of 10 classes on the test set.

```
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: 63 %
```

Then, we calculate the classification accuracy on each class.

```
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(len(labels)):
            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 : 68 %
Accuracy of   car : 72 %
Accuracy of  bird : 47 %
Accuracy of   cat : 34 %
Accuracy of  deer : 63 %
Accuracy of   dog : 52 %
Accuracy of  frog : 79 %
Accuracy of horse : 68 %
Accuracy of  ship : 71 %
Accuracy of truck : 76 %
```

The results show that the performance is not good. A reader is encouraged to improve it by trying different hyperparameters: the number of epochs, batch size, learning rate, optimizer and so on.

Summary

This chapter presents the basic concepts of convolutional neural networks. A convolutional neural network typically consists of convolutional layers with non-linear activation, pooling layers and fully connected layers. A convolution layer is specified by the number of input channels, the number of output channels, filter size, stride, and zero-padding. The convolution operation for one layer can be implemented by a matrix multiplication. The pooling layers down sample the feature maps. In the end of the neural networks, the fully connected layers perform the predictions either for classification or regression. Batch normalization can be applied to convolutional layers.

We also describe the gradient backpropagation in convolutional layers and pooling layers. In convolutional layers, the gradient backpropagation can be implemented by a certain type of convolution. For the pooling layers, the gradients at the output are passed directly to the corresponding locations at the input.

In PyTorch, CNN net is implemented as a class nn.Module, instead of nn.Sequentail(). At the end of the chapter, we present the detailed implementations of CNN examples in PyTorch.

Files: C:/Users/weido/ch11/ch11_1.ipynb (sections: 8.6.1-8.6.5)

C:/Users/weido/ch11/LeNet_5.ipynb (section 8.6.6)

Further reading

[1] The pioneer work for convolutional neural networks:

Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, "Gradient-Based Learning Applied to Document Recognition", Proc of the IEEE November 1998, Page(s): 2278 – 2324.

[2] A comprehensive review on deep learning, including CNNs:

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, "Deep learning", Nature volume 521, pages: 436–444 (2015).

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

Exercises

1. Consider a convolution operation with one feature map at input and three feature maps at output. Manually do the following computations.

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

   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. Write a Python program to compute the convolution in Exercise 1 by two different methods:

   1) Using nested loops defined equation (8.2).

   2) Using a matrix multiplication, illustrated in Fig.8.9.

3. Find the shape of the output and the number of parameters for each of the following layers. Note that 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.

4. Suppose that the input of a pooling layer is x with a shape of [2,6,6], and the pooling filter size is 2×2 with a stride of 2. What is the shape of the output? Calculate the output for the input given below.

$x[0,:,:]$

1	3	2	1	0	9
2	9	1	1	1	2
1	3	2	3	5	5
5	6	1	2	2	7
9	7	4	1	8	8
1	3	7	2	1	7

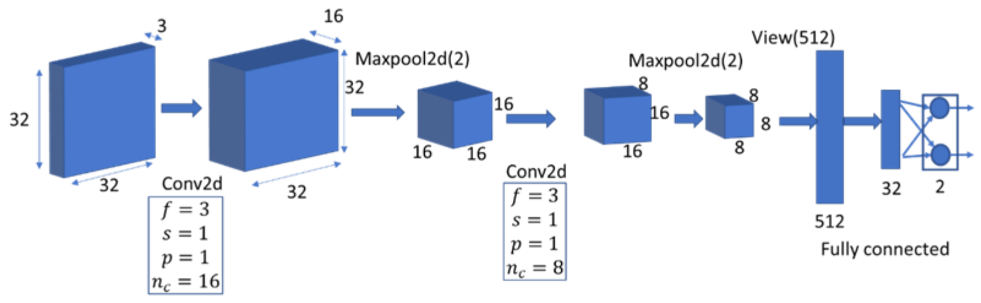
$x[1,:,:]$

4	7	2	8	3	0
2	0	1	4	1	0
8	3	8	3	7	1
4	0	1	5	2	3
8	6	4	1	5	7
1	4	0	0	0	9

   a) Max pooling

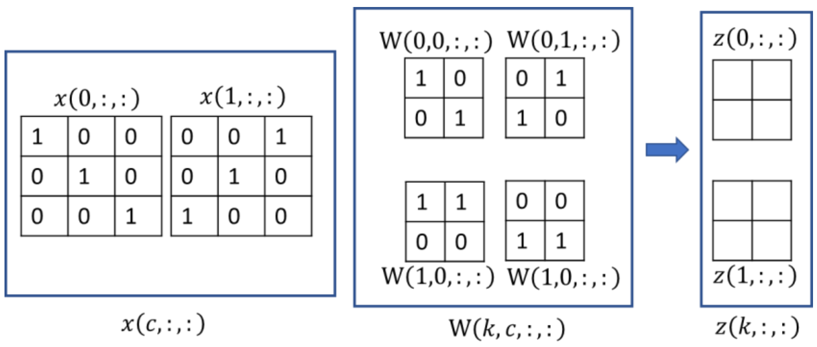
   b) Average pooling

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

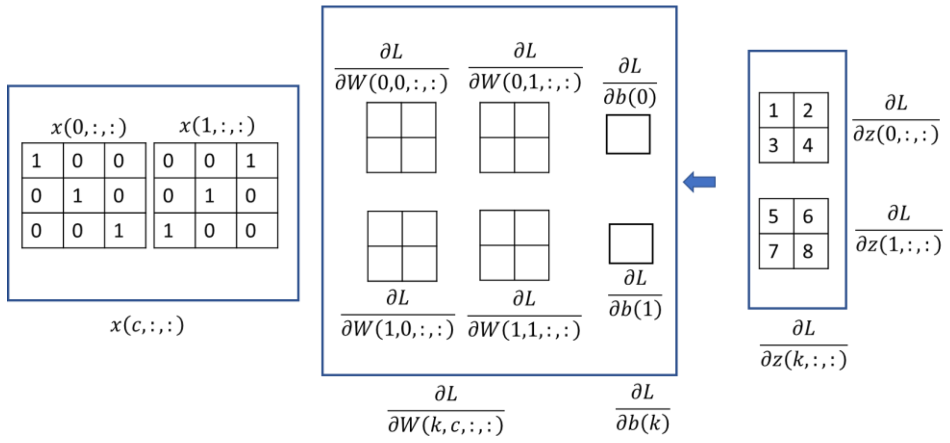


6. Consider a convolutional layer below. The shapes of input, filter and output are $x[2,3,3], W[2,2,2,2], z[2,2,2]$, respectively.

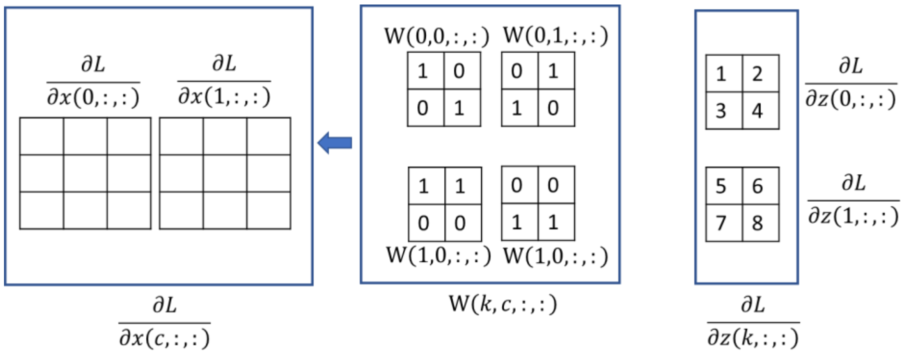
1) Calculate the output for the given input values and weights.



2) Calculate the gradient of the loss w.r.t the weights and bias, given the input values and the gradient of the loss at the output.

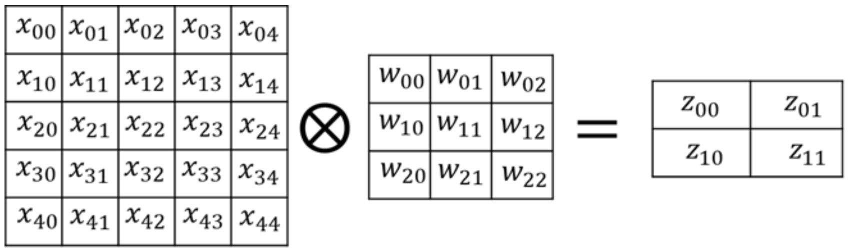

$x(0,:,:)$
1 0 0
0 1 0
0 0 1
$x(1,:,:)$
0 0 1
0 1 0
1 0 0
$x(c,:,:)$
$\frac{\partial L}{\partial W(0,0,:,:)}$
$\frac{\partial L}{\partial W(0,1,:,:)}$
$\frac{\partial L}{\partial b(0)}$
$\frac{\partial L}{\partial W(1,0,:,:)}$
$\frac{\partial L}{\partial W(1,1,:,:)}$
$\frac{\partial L}{\partial b(1)}$
$\frac{\partial L}{\partial W(k,c,:,:)}$
$\frac{\partial L}{\partial b(k)}$
1 2
3 4
$\frac{\partial L}{\partial z(0,:,:)}$
5 6
7 8
$\frac{\partial L}{\partial z(1,:,:)}$
$\frac{\partial L}{\partial z(k,:,:)}$

3) Calculate the gradient of the loss w.r.t the input, given the filter weights and the gradient at the output.



$\frac{\partial L}{\partial x(0,:,:)}$
$\frac{\partial L}{\partial x(1,:,:)}$
$\frac{\partial L}{\partial x(c,:,:)}$
W(0,0,:,:) W(0,1,:,:)
1 0 / 0 1
0 1 / 1 0
1 1 / 0 0
0 0 / 1 1
W(1,0,:,:) W(1,0,:,:)
W(k,c,:,:)
1 2 / 3 4
5 6 / 7 8
$\frac{\partial L}{\partial z(0,:,:)}$
$\frac{\partial L}{\partial z(1,:,:)}$
$\frac{\partial L}{\partial z(k,:,:)}$

7. Consider a convolutional layer with a stride s=2, shown below.



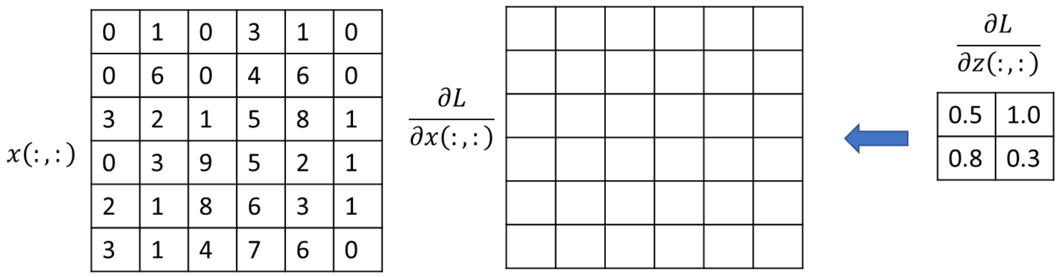
x_{00} x_{01} x_{02} x_{03} x_{04}
x_{10} x_{11} x_{12} x_{13} x_{14}
x_{20} x_{21} x_{22} x_{23} x_{24}
x_{30} x_{31} x_{32} x_{33} x_{34}
x_{40} x_{41} x_{42} x_{43} x_{44}
$\otimes$
w_{00} w_{01} w_{02}
w_{10} w_{11} w_{12}
w_{20} w_{21} w_{22}
$=$
z_{00} z_{01}
z_{10} z_{11}

Stride s=2

1) Justify equation (8.7) by deriving the gradient $\frac{\partial L}{\partial W}$, given the input X and $\frac{\partial L}{\partial Z}$, using the chain rule, based on this convolutional layer.

2) Justify equation (8.12) by deriving the gradient $\frac{\partial L}{\partial X}$, given the weights W and $\frac{\partial L}{\partial Z}$, using the chain rule, based on this convolutional layer.

8. For a pooling layer with filter 3×3 and stride s=3, the input and the output gradient are given below. Calculate the backpropagation, i.e., $\frac{\partial L}{\partial x(:,:)}$, for max pooling and average pooling, respectively.



$x(:,:)$
$\frac{\partial L}{\partial x(:,:)}$
$\frac{\partial L}{\partial z(:,:)}$

0	1	0	3	1	0
0	6	0	4	6	0
3	2	1	5	8	1
0	3	9	5	2	1
2	1	8	6	3	1
3	1	4	7	6	0

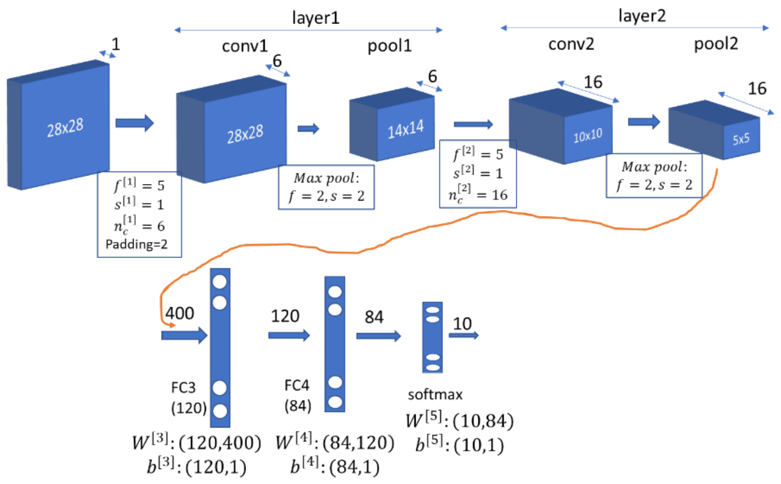
0.5	1.0
0.8	0.3

9. In section **8.6.6 CIFAR-10 Image Classifier**, LeNet-5 with two batch normalization layers is defined as Net().

   1) Calculate how many learnable parameters in the neural network.
   2) Write a few Python statements to automatically generate the number of parameters for each layer and the total number of parameters.
   3) Compare your calculation result with the result generated by the Python code.

```
numel_list = [p.numel() for p in net.parameters()]
sum(numel_list), numel_list
 (62186, [450, 6, 6, 6, 2400, 16, 48000, 120, 10080, 84, 84, 84, 840, 10])
```

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


layer1
conv1
pool1
layer2
conv2
pool2
1
28x28
6
28x28
6
14x14
16
10x10
16
5x5
$f^{[1]} = 5$
$s^{[1]} = 1$
$n_c^{[1]} = 6$
Padding=2
Max pool: $f = 2, s = 2$
$f^{[2]} = 5$
$s^{[2]} = 1$
$n_c^{[2]} = 16$
Max pool: $f = 2, s = 2$
400
120
84
10
FC3 (120)
FC4 (84)
softmax
$W^{[3]}$: (120,400)
$b^{[3]}$: (120,1)
$W^{[4]}$: (84,120)
$b^{[4]}$: (84,1)
$W^{[5]}$: (10,84)
$b^{[5]}$: (10,1)