Chapter 10 Convolutional Neural Networks

> At present, artificial intelligence has not reached the level of 5 years old human, but the progress in perception is rapid. In the field of machine speech and visual recognition, there is no suspense to surpass humans in five to ten years. –Xiangyang Shen

We have introduced the basic theory of neural networks, the use of TensorFlow and the basic fully connected network model, and have a more comprehensive and in-depth understanding of neural networks. But for deep learning, we still have a little doubt. The depth of deep learning refers to the deeper layers of the network, generally more than 5 layers, and most of the neural network layers introduced so far are implemented within 5 layers. So what is the difference and connection between deep learning and neural networks?

Essentially, deep learning and neural networks refer to the same type of algorithm. In the 1980s, the network model based on the Multi-Layer Perceptron (MLP) mathematical model of biological neurons was called a neural network. Due to factors such as limited computing power and small data size at the time, neural networks were generally only able to train to a small number of layers. We call this type of neural network a shallow neural network (Shallow Neural Network). It is not easy for shallow neural networks to extract high-level features from data, and the general expression ability is not good. Although it has achieved good results in simple tasks such as digital picture recognition, it is quickly surpassed by the new support vector machine proposed in the 1990s.

Geoffrey Hinton, a professor at the University of Toronto in Canada, has long insisted on the research of neural networks. However, due to the popularity of support vector machines at that time, research related to neural networks encountered many obstacles. In 2006, Geoffrey Hinton proposed a layer-by-layer pre-training algorithm in [1], which can effectively initialize the Deep Belief Networks (DBN) network, thereby making it possible to train large-scale, deep layers (millions of parameters) of networks. In the paper, Geoffrey Hinton called the neural network Deep Neural Network, and the related research is also called Deep Learning (Deep Learning). From this point of view, deep learning and neural networks are essentially consistent in their designation, and deep learning focuses more on deep neural networks. The “depth” of deep learning will be most vividly reflected in the relevant network models in this chapter.

Before learning a deeper network model, let us first consider such a question: The theoretical research of neural networks was basically in place in the 1980s, but why did it fail to fully exploit the great potential of deep networks? Through the discussion of this question, we lead to the core content of this chapter: Convolutional Neural Networks. This is also a type of neural network that can easily reach hundreds of layers.

10.1 Problems with fully connected networks

First, let's analyze the problems of the fully connected network. Consider a simple 4-layer fully connected layer network. The input is a handwritten digital picture vector of 784 nodes after leveling. The number of nodes in the middle three hidden layers is 256, and the number of nodes in the output layer is 10, as shown in Figure 10.1.

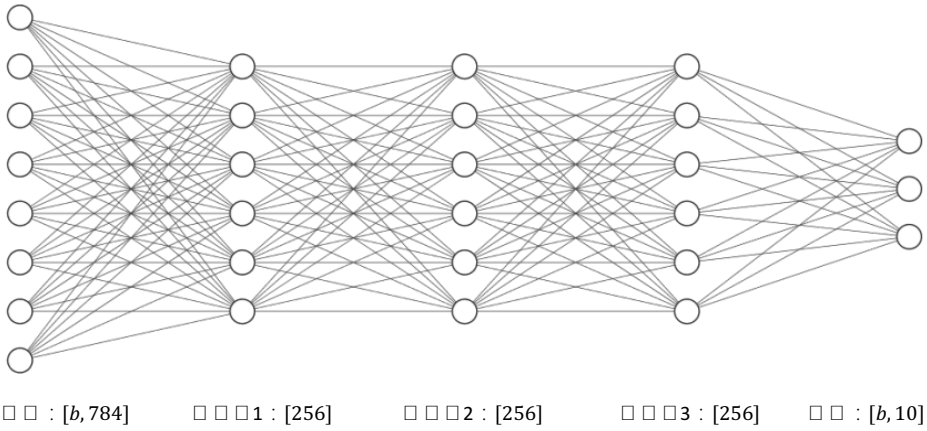


Figure 10.1 Diagram of 4-layer fully connected network structure

We can quickly build this network model through TensorFlow: add 4 Dense layers, and use the Sequential container to encapsulate it as a network object:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers,Sequential,losses,optimizers,datasets
# Create 4-layer fully connected network
model = keras.Sequential([
    layers.Dense(256, activation='relu'),
    layers.Dense(256, activation='relu'),
    layers.Dense(256, activation='relu'),
    layers.Dense(10),
])
# build model and print the model info
model.build(input_shape=(4, 784))
model.summary()
```

Use the summary() function to print out the statistical results of the parameters of each layer from the model, as shown in Table 10.1. How are the parameters of the network calculated? The weight scalar of each connecting line is considered as a parameter, so for a fully connected layer with n input nodes and m output nodes, there are a total of $n \cdot m$ parameters contained in the tensor $\boldsymbol{W}$, and m parameters are contained in the vector $\boldsymbol{b}$. Therefore, the total number of parameters of the fully connected layer is $n \cdot m + m$. Taking the first layer as an example, the input feature length is 784, the output feature length is 256, and the parameter amount of the current layer is $784 \cdot 256 + 256 = 200960$. The same method can be used to calculate the parameter amounts of the second, third, and fourth layers, which are $65792, 65792, and\ 2570$ respectively. The total parameter amount is about 340,000. In a computer, if you save a single

weight as a float type variable, you need to occupy at least 4 bytes of memory (float takes more memory in Python), then 340,000 parameters require at least about 1.34MB of memory. In other words, storing the network parameters alone requires 1.34MB of memory. In fact, the network training process also needs to cache the computation graph, gradient information, input and intermediate calculation results, etc., where gradient-related operations take up a lot of resources.

Table 10.1 Network parameter statistics

Layer	Hidden Layer 1	Hidden Layer 2	Hidden Layer 3	Output Layer
Number of parameters	200960	65792	65792	2570

So how much memory does it take to train such a network? We can simply simulate resource consumption on modern GPU devices. In TensorFlow, if you do not set the GPU memory occupation method, all GPU memory will be occupied by default. Here, the TensorFlow memory usage is set to be allocated on demand, and the GPU memory resources occupied by it are observed as follows:

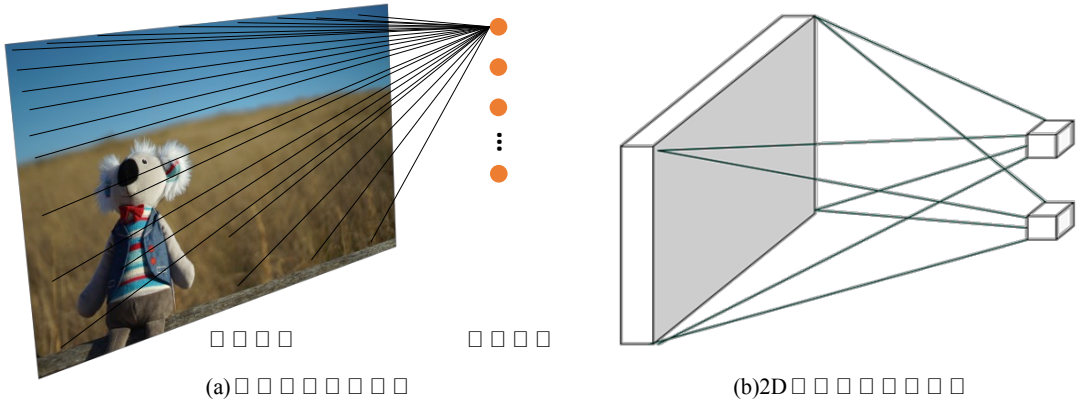
```
# List all GPU devices
gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
  try:
    # Set GPU occupation as on demand
    for gpu in gpus:
      tf.config.experimental.set_memory_growth(gpu, True)
  except RuntimeError as e:
    # excepting handling
    print(e)
```

The above code is inserted after the TensorFlow library imported and before the model created. TensorFlow is configured to apply for GPU memory resources as needed through tf.config.experimental.set_memory_growth(gpu, True). In this way, the amount of GPU memory occupied by TensorFlow is the amount required for the operation. When the Batch Size is set to 32, we observed that GPU memory occupied about 708MB and CPU memory occupied about 870MB during training. Because the deep learning frameworks have different design considerations, this number is for reference only. Even so, we can feel that the computational cost of the 4-layer fully connected layer is not small.

Back to the 1980s, what is the concept of 1.3MB network parameters? In 1989, Yann LeCun used a 256KB memory computer to implement his algorithm in the paper on handwritten zip code recognition [2]. This computer was also equipped with an AT&T DSP-32C DSP computing card (floating point computing capability is about 25 MFLOPS). For the 1.3MB network parameters, the computer with 256KB memory cannot even load the network parameters, let alone network training. It can be seen that the higher memory usage of the fully connected layer severely limits the development of the neural network towards a larger scale and deeper layers.

10.1.1 Local correlation

Next, we explore how to avoid the defect of excessively large parameters of the fully connected network. For the convenience of discussion, we take the scene of picture type data as an example. For 2D image data, before entering the fully connected layer, the matrix data needs to be flattened into a 1D vector, and then each pixel is connected to each output node in pairs as shown in Figure 10.2.



(a)□□□□□□□□□ (b)2D□□□□□□□□□

Figure 10.2 2D feature fully-connected diagram

It can be seen that each output node of the network layer is connected to all input nodes for extracting the feature information of all input nodes. This dense connection method is the root cause of the large number of parameters and the high computational cost of the fully connected layer. The fully connected layer is also called dense connection layer (Dense Layer), and the relationship between output and input is:

$$o_j = \sigma\left(\sum_{i \in \text{nodes}(I)} w_{ij} x_i + b_j\right)$$

where $\text{nodes}(I)$ represents the set of nodes in layer I.

So, is it necessary to connect the output node with all the input nodes? Is there an approximate simplified model? We can analyze the importance distribution of input nodes to output nodes, only consider a more important part of the input node, and discard the less important part of the node, so that the output node only needs to be connected to some input nodes, expressed as:

$$o_j = \sigma\left(\sum_{i \in \text{top}(I,j,k)} w_{ij} x_i + b_j\right)$$

Where $\text{top}(I, j, k)$ represents the top k node set in layer I that has the highest importance for the number node in layer J. In this way, the weighted connections of the fully connected layer can be reduced from $\|I\| \cdot \|J\|$ to $k \cdot \|J\|$, where $\|I\| \; and \; \|J\|$ represent the number of nodes in the I and J layers respectively.

Then the problem changes to exploring the importance distribution of the input node of layer I to the number output node j. However, it is very difficult to find out the importance distribution of each intermediate node. We can use prior knowledge to further simplify this problem.

In real life, there are a lot of data that use location or distance as a measure of importance distribution. For example, people who live closer to themselves are more likely to have greater influence on themselves (location correlation), and stock trend predictions should pay more attention to the recent trend (time correlation), each pixel of the picture is more related to the surrounding pixels (location correlation). Taking 2D image data as an example, if we simply think that the pixels with Euclidean Distance from the current pixel is less than or equal to $\frac{k}{\sqrt{2}}$ are more important, and those with the Euclidean distance is greater than $\frac{k}{\sqrt{2}}$ are less important. Then we can easily simplify the problem of finding the importance distribution of each pixel. As shown in Figure 10.3, the pixels where the solid grid is located are used as reference points, and the pixels whose Euclidean distance is less than or equal to $\frac{k}{\sqrt{2}}$ are represented by a rectangular grid. The pixels in the grid are more important, and the pixels outside the grid are less important. This window is called the Receptive Field, which characterizes the importance distribution of each pixel to the central pixel. The pixels within the grid will be considered, and the pixels outside the grid will be ignored for the central pixel.

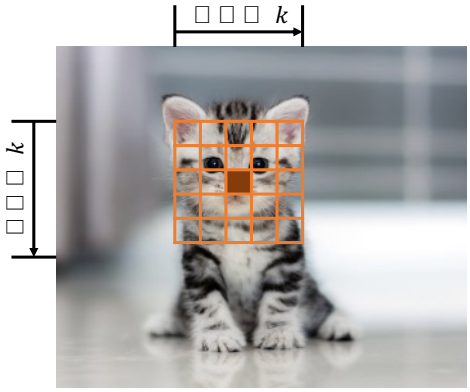


Figure 10.3 Importance distribution of pixels

This hypothetical characteristic of distance-based importance distribution is called local correlation. It only focuses on some nodes that are close to itself and ignores nodes that are far away. Under this assumption of importance distribution, the connection mode of the fully connected layer becomes as shown in Figure 10.4. The output node j is only connected to the local area (receptive field) centered by j, and has no connection to other pixels.

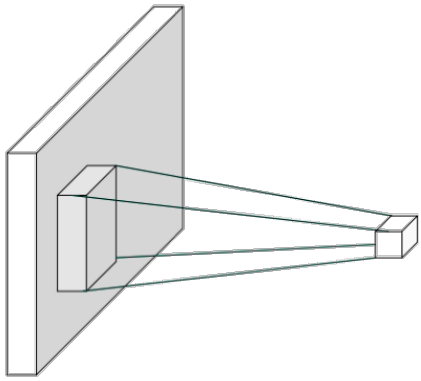


Figure 10.4 Locally connected network diagram

Using the idea of local correlation, we record the height and width of the receptive field window as k (the height and width of the receptive field may not be equal, for convenience, we only consider the case where the height and width are equal). The current node is connected with all pixels in the receptive field, regardless of other pixels outside. The input and output relationship of the network layer is expressed as follows:

$$o_j = \sigma\left(\sum_{\text{dist}(i,j)\leq\frac{k}{\sqrt{2}}} w_{ij}x_i + b_j\right)$$

Where $\text{dist}(i,j)$ represents the Euclidean distance between i and j nodes.

10.1.2 Weight sharing

Each output node is only connected to $k\times k$ input nodes in the receptive field, and the number of output layer nodes is $\|J\|$. So the number of the parameters of the current layer is $k\times k\times\|J\|$. Comparing to the fully connected layer, because k is usually small, such as 1, 3, 5, etc., so $k\times k \ll \|I\|$, which means it successfully reduced the amount of parameters.

Can the amount of parameters be further reduced, for example, can we only need $k\times k$ parameters to complete the calculation of the current layer? The answer is yes. Through the idea of weight sharing, for each output node o_j, the same weight matrix $\boldsymbol{W}$ is used, then no matter how many output nodes $\|J\|$ will be, the number of network layer parameters is always $k\times k$. As shown in Figure 10.5, when calculating the output pixel at the upper left corner, the weight matrix is used:

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

Multiply and accumulate with the pixels inside the corresponding receptive field as the output value of the upper left pixel. When calculating the lower right receptive field, share the weight parameters $\boldsymbol{W}$, that is, use the same weight parameters $\boldsymbol{W}$ to multiply and accumulate to get the output of the lower right pixel value. There are only $3\times 3 = 9$ parameters in the network layer at this time, and it has nothing to do with the number of input and output nodes.

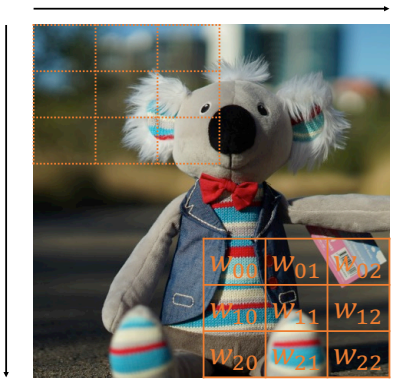


Figure 10.5 Weight sharing matrix diagram

By applying the idea of local correlation and weight sharing, we have successfully reduced the number of network parameters from $\|I\|\times\|J\|$ to $k\times k$ (to be precise, under the conditions of a single input channel and a single convolution kernel). This kind of weighted "local connection layer" network is actually a convolutional neural network. Next, we will introduce convolution operations from a mathematical perspective, and then formally learn the principles and implementation of convolutional neural networks.

10.1.3 Convolution operation

Under the a priori of local correlation, we propose a simplified "local connection layer". For all pixels in the window $k\times k$, feature information is extracted by multiplying and accumulating weights, and each output node extracts features corresponding to the receptive field area. information. This operation is actually a standard operation in the field of signal processing: discrete convolution operation. Discrete convolution operation has a wide range of applications in computer vision. Here is a mathematical explanation of the convolutional neural network layer.

In the field of signal processing, the convolution operation of 1D continuous signals is defined as the integration of two functions: function $f(\tau)$, function $g(\tau)$, where 中$g(\tau)$ becomes $g(n-\tau)$ after flipping and translation. The 1D continuous convolution is defined as:

$$(f\otimes g)(n)=\int_{-\infty}^{\infty} f(\tau)g(n-\tau)\mathrm{d}\tau$$

Discrete convolution replaces the integral operation with the accumulation operation:

$$(f\otimes g)(n)=\sum_{\tau=-\infty}^{\infty} f(\tau)g(n-\tau)$$

As for why convolution is defined in this way, I will not elaborate on it due to space limitations. We focus on 2D discrete convolution operations. In computer vision, the convolution operation is based on 2D picture function $f(m,n)$ and 2D convolution kernel $g(m,n)$, where $f(i,j)$ and $g(i,j)$ only exists in the effective area of the respective window, and the other areas are regarded as 0, as shown in Figure 10.6. The 2D discrete convolution is defined as:

$$[f \otimes g](m,n) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i,j)g(m-i,n-j)$$

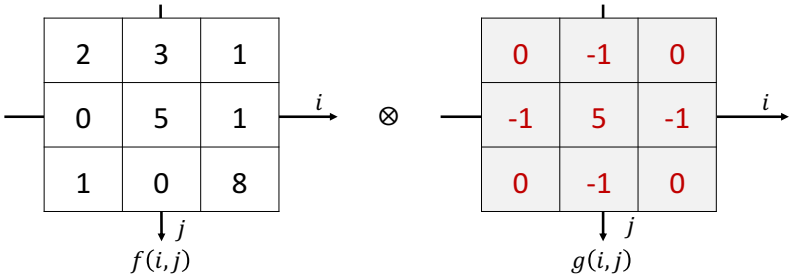


Figure 10.6 2D image function $f(i,j)$ and convolution kernel function $g(i,j)$

Let's introduce the 2D discrete convolution operation in detail. First, invert the convolution kernel function $g(i,j)$ (invert each time along the x and y directions) to become $g(-i,-j)$. When $(m,n) = (-1,-1)$, it means that the convolution kernel function $g(-1-i,-1-j)$ is flipped and then shifted one unit to the left and the upward. At this time:

$$[f \otimes g](-1,-1) = \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} f(i,j)g(-1-i,-1-j)$$
$$= \sum_{i\in[-1,1]} \sum_{j\in[-1,1]} f(i,j)g(-1-i,-1-j)$$

The 2D function only has valid values when $i \in [-1,1], j \in [-1,1]$. In other positions, it is 0. According to the calculation formula, we can get $[f \otimes g](0,-1) = 7$, as shown in Figure 10.7.

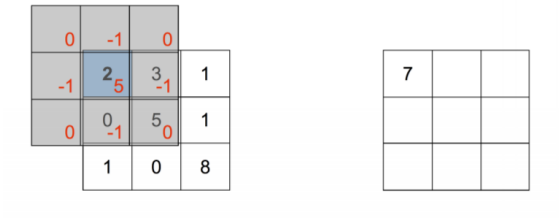


Figure 10.7 Discrete convolution operation-1

Similarly, when $(m,n) = (0,-1)$:

$$[f \otimes g](0,-1) = \sum_{i\in[-1,1]} \sum_{j\in[-1,1]} f(i,j)g(0-i,-1-j)$$

That is, after the convolution kernel is flipped, the unit is shifted upwards and the corresponding position is multiplied and accumulated, $[f \otimes g](0,-1) = 7$, as shown in Figure 10.8.

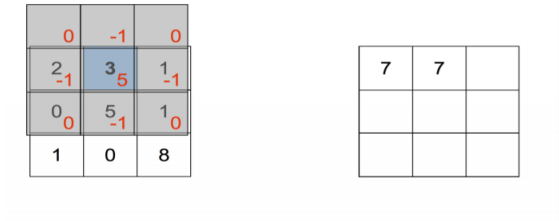


Figure 10.8 Discrete convolution operation -2

When $(m,n)=(1,-1)$:

$$[f\otimes g](1,-1)=\sum_{i\in[-1,1]}\sum_{j\in[-1,1]}f(i,j)g(1-i,-1-j)$$

That is, after the convolution kernel is flipped, it is translated to the right and upward by one unit, and the corresponding position is multiplied and accumulated, $[f\otimes g](1,-1)=1$, as shown in FigureFigure 10.9.

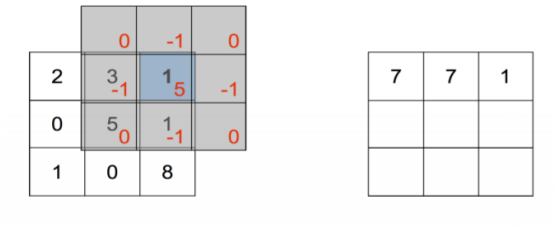


Figure 10.9 Discrete convolution operation -3

When $(m,n)=(-1,0)$:

$$[f\otimes g](-1,0)=\sum_{i\in[-1,1]}\sum_{j\in[-1,1]}f(i,j)g(-1-i,-j)$$

That is, after the convolution kernel is flipped, it is translated to the left by one unit, and the corresponding position is multiplied and accumulated, $[f\otimes g](-1,0)=1$, as shown in Figure 10.10.

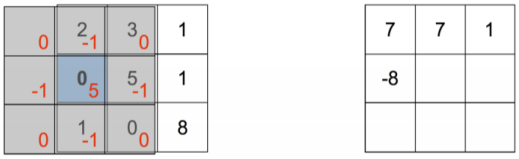


Figure 10.10 Discrete convolution operation -4

Cyclic calculation in this way, we can get all the values of the function $[f\otimes g](m,m), m\in[-1,1], n\in[-1,1]$, as shown in Figure 10.11 below.

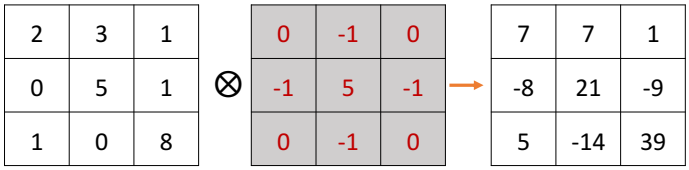


Figure 10.11 2D discrete convolution operation

So far, we have successfully completed the convolution operation of the picture function and the convolution kernel function to obtain a new feature map.

Recalling the operation of "weight multiplying and accumulating", we record it as $[f\cdot g](m,n)$:

$$[f\cdot g](m,n)=\sum_{i\in[-w/2,w/2]}\sum_{j\in[-h/2,h/2]}f(i,j)g(i-m,j-m)$$

Comparing it carefully with the standard 2D convolution operation, it is not difficult to find that the convolution kernel function $g(m,n)$ in "weight multiply-accumulate" has not been

flipped. For neural networks, the goal is to learn a function $g(m,n)$ to make $\mathcal{L}$ as small as possible. As for whether it is exactly the "convolution kernel" function defined in the convolution operation, it is not very important, because we will not directly use it. In deep learning, the function $g(m,n)$ is collectively called a convolution kernel (Kernel), sometimes called Filter, Weight, etc. Since the function $g(m,n)$ is always used to complete the convolution operation, the convolution operation has actually realized the idea of weight sharing.

Let's summarize the 2D discrete convolution operation process: each time by moving the convolution kernel and multiplying and accumulating with the receptive field pixels at the corresponding position of the picture, the output value at this position is obtained. The convolution kernel is a weight matrix $\boldsymbol{W}$ with rows and columns as size of k. The window corresponding to the size k on the feature map is the receptive field. The receptive field and the weight matrix are multiplied and accumulated to obtain the output value at this position. Through weight sharing, we gradually move the convolution kernel from the upper left to the right and downward to extract the pixel features at each position until the bottom right, completing the convolution operation. It can be seen that the two ways of understanding are the same. From a mathematical point of view, the convolutional neural network is to complete the discrete convolution operation of the 2D function; from the perspective of local correlation and weight sharing, the same effect can be obtained. Through these two perspectives, we can not only intuitively understand the calculation process of the convolutional neural network, but also rigorously derive from the mathematical point of view. It is based on convolution operations that convolutional neural networks can be so named.

In the field of computer vision, 2D convolution operations can extract useful features of data, and perform convolution operations on input images with specific convolution kernels to obtain output images with different characteristics. As shown in Table 10.2 below, some common convolution kernels and corresponding effects are listed.

Table 10.2 Common convolution kernels and their effect

			
$\begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}$	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	$\begin{bmatrix} 0.0625 & 0.125 & 0.0625 \\ 0.125 & 0.25 & 0.125 \\ 0.0625 & 0.125 & 0.0625 \end{bmatrix}$	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$
Original	Sharpen	Blur	Edge-sharpen

10.2 Convolutional Neural Network

The convolutional neural network makes full use of the idea of local correlation and weight

sharing, which greatly reduces the amount of network parameters, thereby improving training efficiency and making it easier to realize ultra-large-scale deep networks. In 2012, Alex Krizhevsky of the University of Toronto in Canada applied the deep convolutional neural network to the large-scale image recognition challenge ILSVRC-2012, and achieved a Top-5 error rate of 15.3% on the ImageNet dataset, ranking first. Comparing to the 2nd place, Alex reduced the Top-5 error rate by 10.9% [3]. This huge breakthrough has attracted strong industry attention. Convolutional neural networks quickly became the new favorite in the field of computer vision. Subsequently, in a series of tasks, convolution-based neural network models have been proposed one after another, and have achieved tremendous improvements in the original performance.

Now let's introduce the specific calculation process of the convolutional neural network layer. Taking 2D image data as an example, the convolutional layer accepts input feature maps $\boldsymbol{X}$ with height h and width w, and the number of channels c_{in}. Under the action of c_{out} convolution kernels with height h and width w and the number of channels c_{in}, feature maps with the height h' and width w' and c_{out} channels are generated. It should be noted that the height and width of the convolution kernel can be unequal. In order to simplify the discussion, we only consider the equal height and width cases, and then it can be easily extended to the case of unequal height and width.

We start with the discussion of the single-channel input and single-convolution kernel, and then generalize to the multi-channel input and single-convolution kernel, and finally discuss the most commonly used and most complex convolutional layer implementation of multi-channel input and multiple convolution kernels.

10.2.1 Single channel input and single convolution kernel

First, we discuss single-channel input $c_{in} = 1$, such as a gray-scale image with only one channel of gray value and a single convolution kernel $c_{out} = 1$. Take the input matrix $\boldsymbol{X}$ with size 5×5 and the convolution kernel matrix with size 3×3 as examples, as shown in Figure 10.12. The receptive field of the same size as the convolution kernel (the green box above the input $\boldsymbol{X}$) is first moved to the top left of the input $\boldsymbol{X}$. Select the receptive field element on the input and multiply it by the corresponding element of the convolution kernel (the middle box in the picture):

$$\begin{bmatrix} 1 & -1 & 0 \\ -1 & -2 & 2 \\ 1 & 2 & -2 \end{bmatrix} \odot \begin{bmatrix} -1 & 1 & 2 \\ 1 & -1 & 3 \\ 0 & -1 & -2 \end{bmatrix} = \begin{bmatrix} -1 & -1 & 0 \\ -1 & 2 & 6 \\ 0 & -2 & 4 \end{bmatrix}$$

The $\odot$ symbol indicates the Hadamard Product, that is, the corresponding element of the matrix is multiplied. The symbol @ (matrix multiplication) is another common forms of matrix operations. After the operation of the matrix, all 9 values are added:

$$-1 - 1 + 0 - 1 + 2 + 6 + 0 - 2 + 4 = 7$$

We get the scalar 7 and write to the position of the first row and first column of the output matrix, as shown in Figure 10.12.

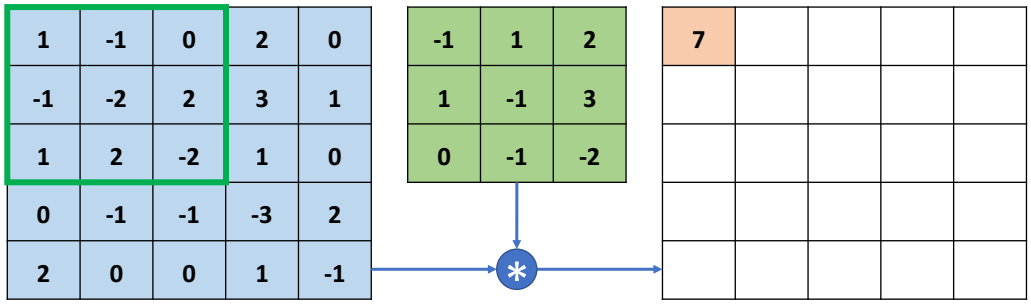


Figure 10.12 3×3 convolution operation−1

After the feature extraction of the first receptive field area is completed, the receptive field window moves one step unit (Strides, denoted as s, default is 1) to the right and select the 9 receptive field elements in the green box in Figure 10.13. Similarly, multiplying and accumulating the corresponding elements of the convolution kernel, we can get the output 10, which is written to the first row and second column position.

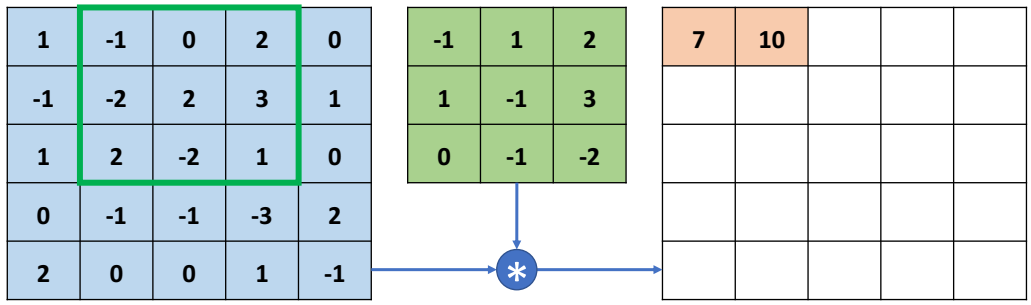


Figure 10.13 3×3 convolution operation -2

Move the receptive field window to the right by one step unit again, select the element in the green box in Figure 10.14, multiply and accumulate with the convolution kernel, get the output 3, and write to the first row and third column of the output, as shown in Figure 10.14.

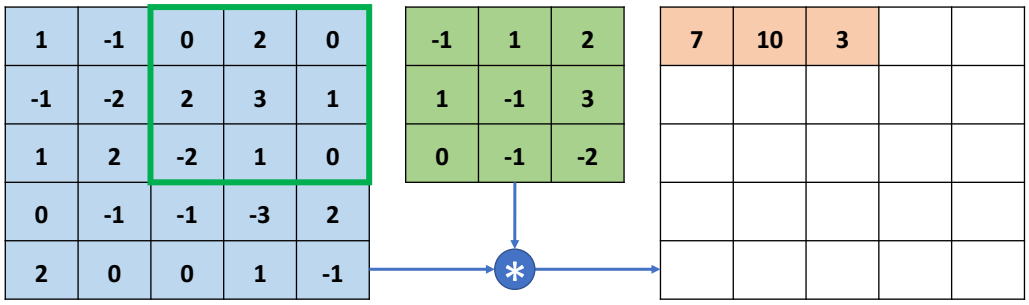


Figure 10.14 3×3 convolution operation -3

At this point, the receptive field has moved to the far right of the effective pixel input, and it cannot continue to move to the right (without filling the invalid element), so the receptive field window moves down by one step unit ($s = 1$) and returns to the beginning of the current line, continue to select the new receptive field element area, as shown in Figure 10.15, and the convolution kernel operation results in output -1. Because the receptive field moves down by one step, so the output value -1 is written in the second row and the first column position.

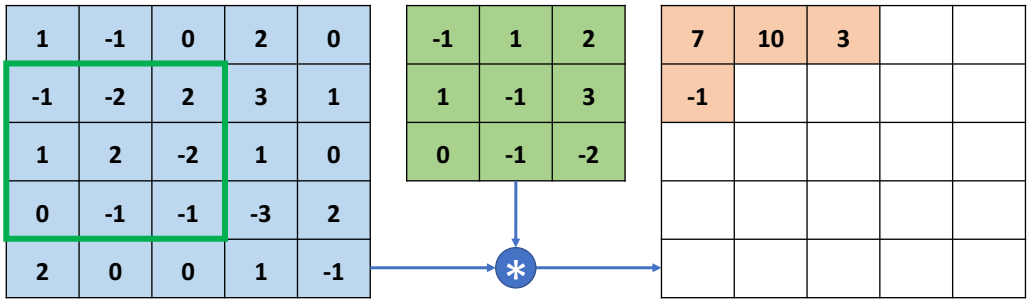


Figure 10.15 3×3 convolution operation-4

According to the above method, each time the receptive field moves right by one step ($s = 1$), if it exceeds the input boundary, it moves down by one step ($s = 1$) and returns to the beginning of the line until the receptive field moves to the rightmost and bottommost position, as shown below in Figure 10.16. Each selected receptive field element is multiplied by the corresponding element of the convolution kernel and written to the corresponding position of the output. In the end, we get a 3×3 matrix, which is slightly smaller than the input 5×5, this is because the receptive field cannot exceed the element boundary. It can be observed that the size of the output matrix of the convolution operation is determined by the size k of the convolution kernel, the height h and width w of the input $\boldsymbol{X}$, the moving step s, and whether boundaries are filled.

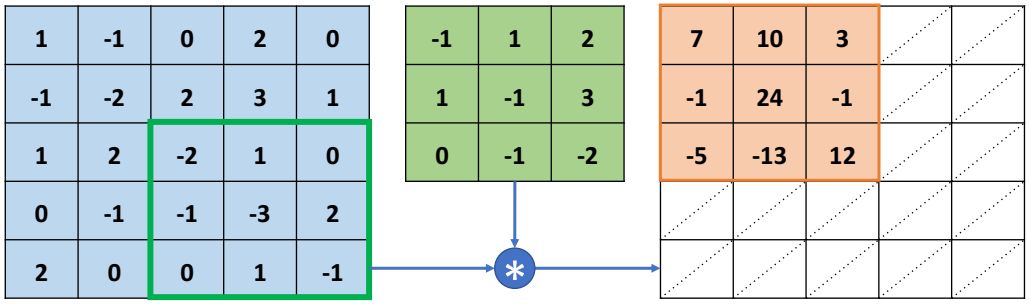


Figure 10.16 3×3 convolution operation -5

Now we have introduced the calculation process of single-channel input and single convolution kernel. The actual number of input channels of the neural network is often large. Next, we will learn the convolution operation method of multi-channel input and a single convolution kernel.

10.2.2 Multi-channel input and single convolution kernel

Multi-channel input convolutional layers are more common. For example, a color image contains three channels (R/G/B). The pixel value on each channel indicates the intensity of the R/G/B color. In the following, we take 3-channel input and a single convolution kernel as an example to extend the convolution operation of single-channel input to multi-channel. As shown in Figure 10.17, the leftmost 5×5 matrix of each row represents the input channels 1~3, the 3×3 matrix in the second column represents the channels 1~3 of the convolution kernel, and the matrix in the third column represents the middle matrix of the calculation on the current channel, the rightmost matrix represents the final output of the convolutional layer operation.

In the case of multi-channel input, the number of channels of the convolution kernel needs to match the number of input channels. The ith channel of the convolution kernel and the ith channel of the input $\boldsymbol{X}$ are calculated to obtain the first intermediate matrix, which can be then regarded as the case of single input and single convolution kernel. The corresponding elements of the intermediate matrix of all channels are added again as the final output.

The specific calculation process is as follows: In the initial state, as shown in Figure 10.17, the receptive field window on each channel synchronously falls on the leftmost and topmost positions on the corresponding channel. The receptive field area elements and the convolution kernel on each channel multiply and accumulate the matrix above the corresponding channel to obtain the intermediate variables of the output 7, -11, and -1 on the three channels, and then we can add these intermediate variables to get the output -5, and write it to the corresponding position.

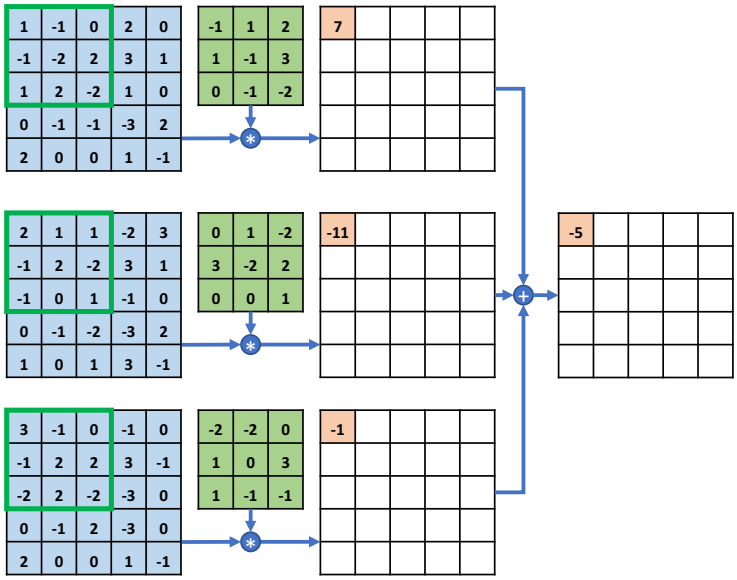


Figure 10.17 Mulit-channel input and single convolution kernel -1

Then, the receptive field window moves synchronously to the right by one step ($s = 1$) on each channel. At this time, the receptive field area elements are shown in Figure 10.18. The receptive field on each channel is multiplied by the matrix on the corresponding channel of the convolution kernel and is then accumulated to get the intermediate variables 10, 20, 20. We then add them up to get the output 50, and write the element position of the first row and second column.

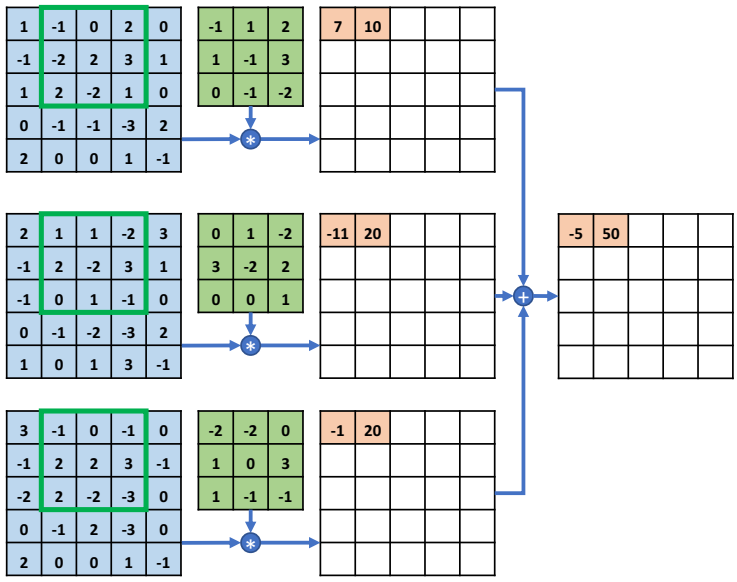


Figure 10.18 Mulit-channel input and single convolution kernel -2

In this way, the receptive field window is moved synchronously to the rightmost and bottommost positions. All the convolution operations of the input and the convolution kernel are completed, and the resulting 3×3 output matrix is shown in Figure 10.19.

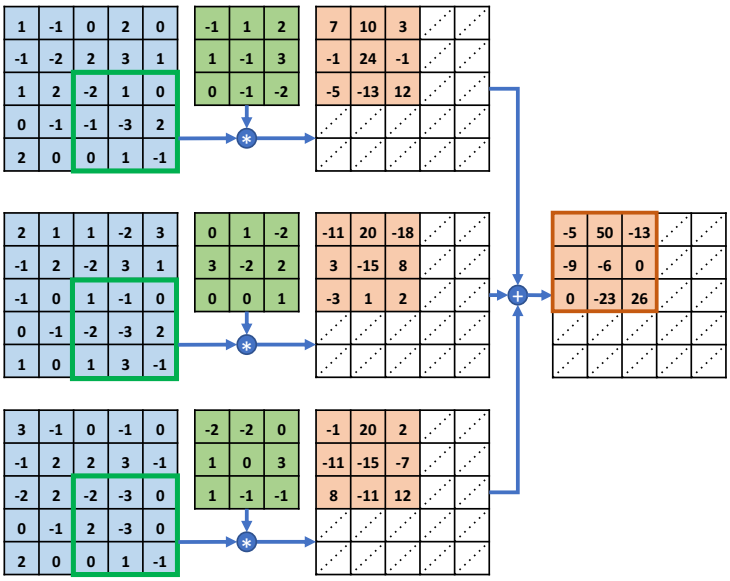


Figure 10.19 Mulit-channel input and single convolution kernel -3

The entire calculation diagram is shown in Figure 10.20. The receptive field at each input channel is multiplied by the corresponding channel of the convolution kernel to obtain intermediate variables equal to the number of channels. All of these intermediate variables are added to obtain the output value in current position. The number of input channels determines the number of convolution kernel channels. A convolution kernel can only get one output matrix, regardless of the number of input channels.

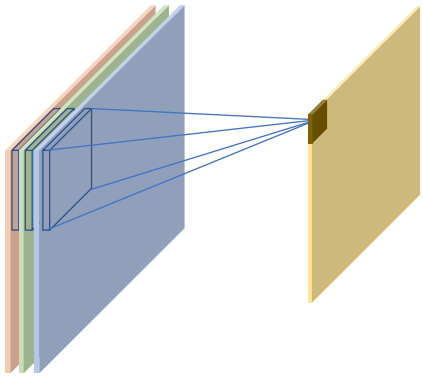


Figure 10.20 Mulit-channel input and single convolution kernel diagram

Generally speaking, a convolution kernel can only complete the extraction of a certain logical feature. When multiple logical features need to be extracted at the same time, it can be achieved by adding multiple convolution kernels to improve the expression ability of the neural network. This is the case of multi-channel input and multi-convolution kernels.

10.2.3 Multi-channel input and multi-convolution kernel

Multi-channel input and multi-convolution kernels are the most common forms of convolutional neural networks. We have already introduced the operation process of single convolution kernels. Each convolution kernel and input are convolved to obtain an output matrix.

When there are multiple convolution kernels, the ith ($i \in [1, n]$, n is the number of convolution kernels) convolution kernel and input $\boldsymbol{X}$ get the ith output matrix (also called the channel i of output tensor $\boldsymbol{O}$), and finally all the output matrix in the channel dimension stitch together (Stack operation to create a new dimension - the number of output channels) to generate an output tensor $\boldsymbol{O}$ that contains n channels.

Take a convolutional layer with 3 channels of input and 2 convolution kernels as an example. The first convolution kernel and input $\boldsymbol{X}$ get the first output channel, and the second convolution kernel and input $\boldsymbol{X}$ get the second output channel, as shown in Figure 10.21. The two output channels are stitched together to form the final output $\boldsymbol{O}$. The size k, stride size s, and padding settings of each convolution kernel are uniformly set, so as to ensure that each output channel has the same size to meet the conditions of stitching.

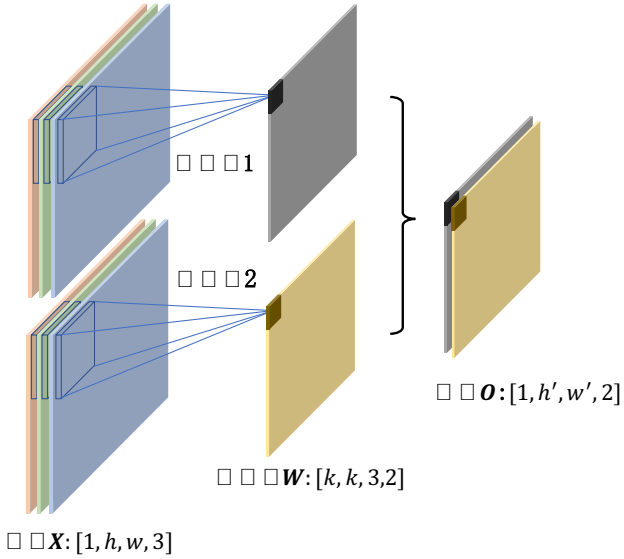


Figure 10.21 Diagram of multi convolution kernels

10.2.4 Stride size

In convolution operation, how to control the density of receptive field layout? For inputs with high information density, such as pictures with a large number of objects, in order to maximize the useful information, it is desirable to arrange the receptive field windows more densely during network design. For inputs with lower information density, such as a picture of the ocean, we can reduce the number of receptive fields in an appropriate amount. The control method of receptive field density is generally realized by moving strides.

The stride size refers to the unit of length for each movement of the receptive field window. For 2D input, it is divided into movement lengths in the x (right) direction and y (downward) direction. In order to simplify the discussion, we only consider the case of same stride size for both directions, which is also the most common setting in neural networks. As shown in Figure 10.22, the position of the receptive field window represented by the solid green line is the current position, and the dashed green line represents the position of the last receptive field. The movement length from the last position to the current position is the definition of the stride size. In

Figure 10.22, the stride length of the receptive field in the x direction is 2, which is expressed as $s = 2$.

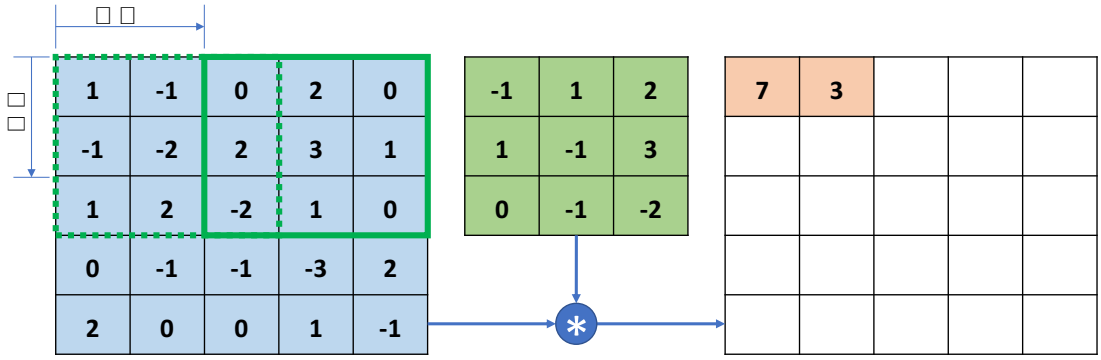


Figure 10.22 Diagram of step size

When the receptive field reaches to the right boundary of the input $\boldsymbol{X}$, it moves down one stride ($s = 2$) and returns to the beginning of the line as shown in Figure 10.23.

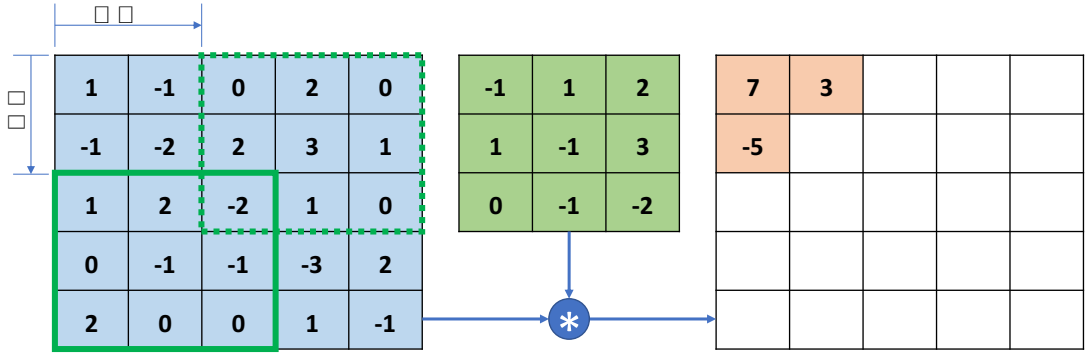


Figure 10.23 Convolution operation stride size demnostration-1

Circulate back and forth until the bottom and right edges are reached as shown in Figure 10.24. The final output height and width of the convolutional layer are only 2×2. Compared with the previous situation ($s = 1$), the output height and width are reduced from 3×3 to 2×2 and the number of receptive fields is reduced to only 4.

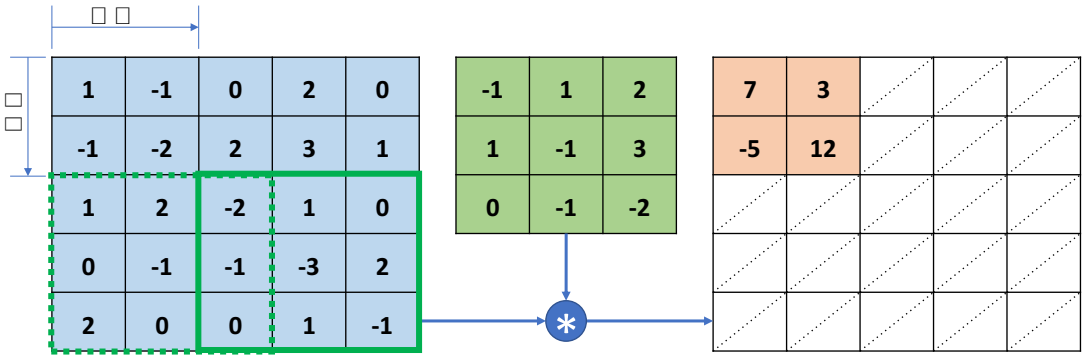


Figure 10.24 Convolution operation stride size demnostration-2

It can be seen that by setting the stride size, the extraction of information density can be effectively controlled. When the stride size is small, the receptive field moving window is small, which is helpful to extract more feature information and the size of the output tensor is larger; when the stride size is larger, the receptive field moving window is larger which is helpful to reduce the calculation cost and filter redundant information, and of course, the size of the output tensor is also smaller.

10.2.5 Padding

After the convolution operation, the height and width of the output will generally be smaller than the height and width of the input. Even when the stride size is 1, the height and width of the output will be slightly smaller than the input height and width. When designing a network model, it is sometimes desired that the height and width of the output can be the same as the height and width of the input, thereby facilitating the design of network parameters and residual connection. In order to make the height and width of the output equal to that of the input, it is common to increase the input by padding several invalid elements on the height and width of the original input. By carefully designing the number of filling units, the height and width of the output after the convolution operation can be equal to the original input, or even larger.

As shown in Figure 10.25, we can fill an indefinite number at the top, bottom, left or right boundaries. The default filled number is 0, and it can also be filled with customized data. In Figure 10.25, one row is filled in the upper and lower directions, and two columns are filled in the left and right directions.

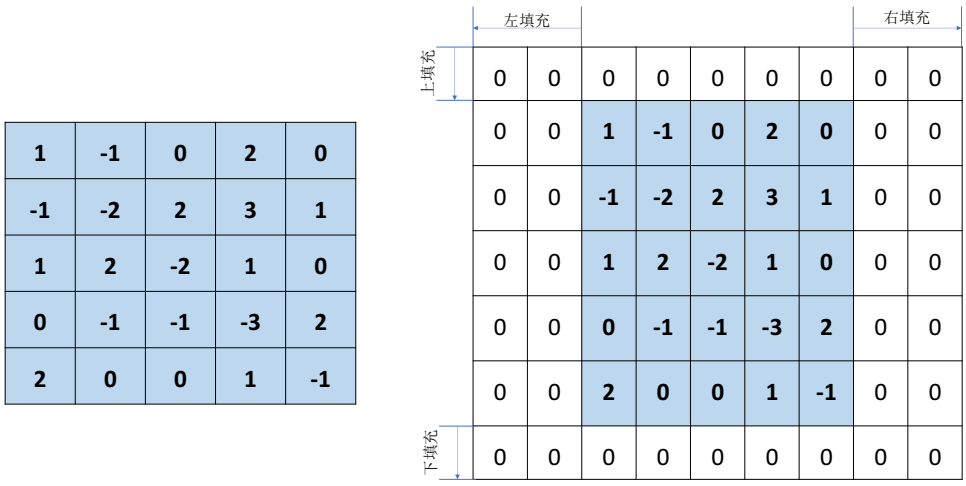


Figure 10.25 Matrix padding diagram

So how to calculate the convolutional layer after filling? We can simply replace the input $\boldsymbol{X}$ with the new tensor $\boldsymbol{X}'$ obtained after filling. As shown in Figure 10.26, the initial position of the receptive field is at the upper left of $\boldsymbol{X}'$. Similar as before, the output 1 is obtained and written to the corresponding position of the output tensor.

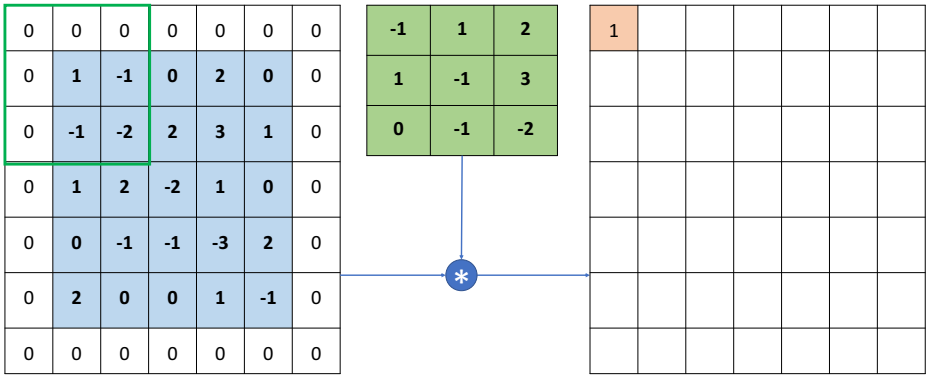


Figure 10.26 Convolution operation after padding-1

Move the stride by one unit and repeat the operation to get the output 0, as shown in Figure 10.27.

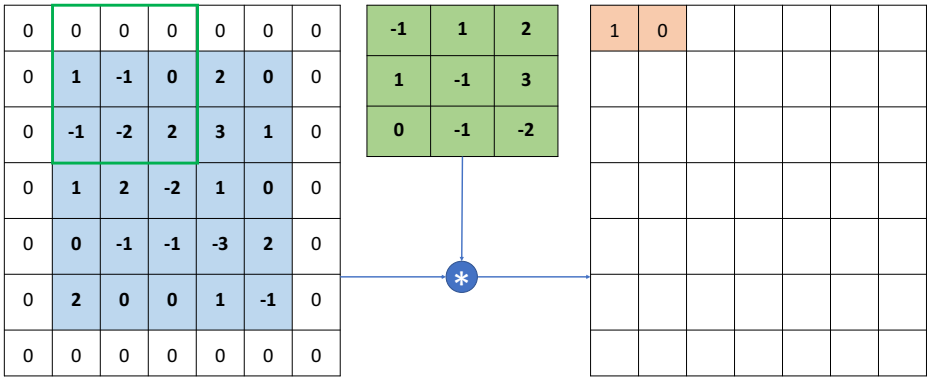


Figure 10.27 Convolution operation after padding -2

Looping back and forth, the resulting output tensor is shown in Figure 10.28.

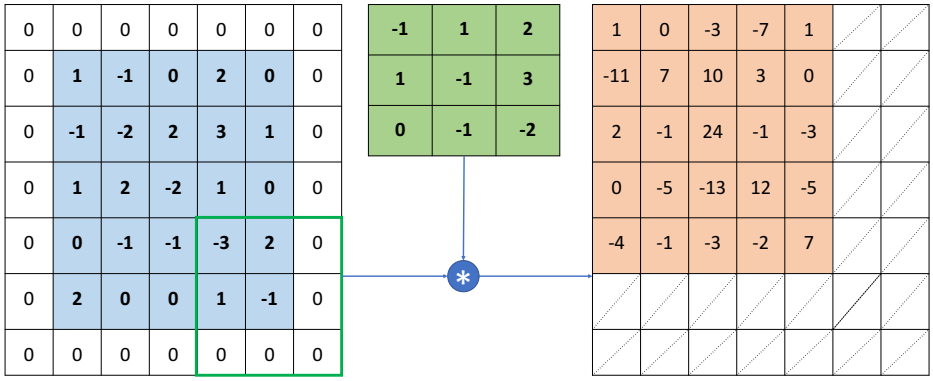


Figure 10.28 Convolution operation after padding -3

Through the carefully designed padding scheme, that is, filling one unit ($p = 1$) up, down, left, and right, we can get the result $\boldsymbol{O}$ that has the same height and width of the input. Without padding, as shown in Figure 10.29, we can only get the output slightly smaller than the input.

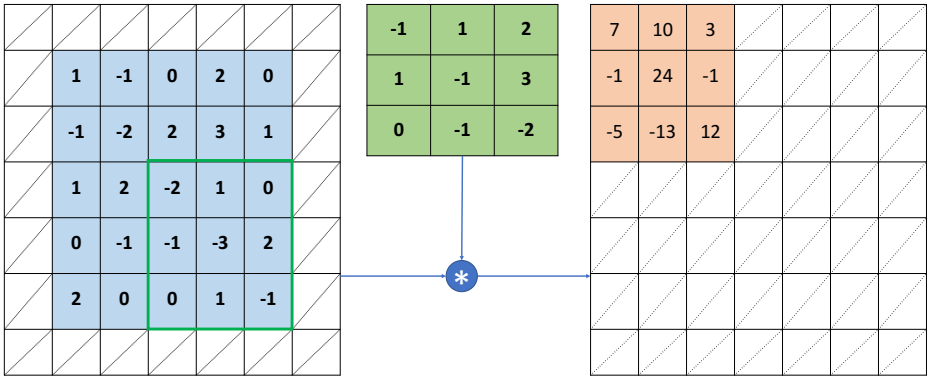


Figure 10.29 Convolution output without padding

The output size $[b, h', w', c_{out}]$ of the convolutional neural layer is determined by the number of convolution kernels c_{out}, the size of the convolution kernel k, the stride size s, the

number of padding p (only considering the same number of top and bottom paddings p_h, and the same number of left and right paddings p_w) and the height h and width w of the input $\boldsymbol{X}$. The mathematical relationship between can be expressed as:

$$h' = \left\lfloor \frac{h + 2 \cdot p_h - k}{s} \right\rfloor + 1$$

$$w' = \left\lfloor \frac{w + 2 \cdot p_w - k}{s} \right\rfloor + 1$$

Where p_h and p_w indicate the padding quantity in the height and width directions respectively, and $\lfloor \cdot \rfloor$ indicates rounding down. Taking the above example as an example, $h = w = 5$，$k = 3$，$p_h = p_w = 1$，$s = 1$, the output are:

$$h' = \left\lfloor \frac{5 + 2 * 1 - 3}{1} \right\rfloor + 1 = \lfloor 4 \rfloor + 1 = 5$$

$$w' = \left\lfloor \frac{5 + 2 * 1 - 3}{1} \right\rfloor + 1 = \lfloor 4 \rfloor + 1 = 5$$

In TensorFlow, when 在$s = 1$, if you want the output $\boldsymbol{O}$ and input $\boldsymbol{X}$ to be equal in height and width, you only need to simply set the parameter padding="SAME" to make TensorFlow automatically calculate the number of padding, which is very convenient.

10.3 Convolutional layer implementation

In TensorFlow, you can either build a neural network through a low-level implementation of custom weights, or you can directly call a high-level API of convolutional layers to quickly build a complex network. We mainly take 2D convolution as an example to introduce how to implement a convolutional neural network layer.

10.3.1 Custom weights

In TensorFlow, the 2D convolution operation can be easily realized through the tf.nn.conv2d function. tf.nn.conv2d performs a convolution operation based on input $\boldsymbol{X}$:$[b, h, w, c_{in}]$ and convolution kernel $\boldsymbol{W}$:$[k, k, c_{in}, c_{out}]$ to get the output $\boldsymbol{O}$:$[b, h', w', c_{out}]$, where c_{in} represents the number of input channels, c_{out} indicates the number of convolution kernels which is also the number of output channels.

```
In [1]:
x = tf.random.normal([2,5,5,3]) # input with 3 channels with height and
width 5
# Create w using [k,k,cin,cout] format, 4 3x3 kernels
w = tf.random.normal([3,3,3,4])
# Stride is 1, padding is 0,
out = tf.nn.conv2d(x,w,strides=1,padding=[[0,0],[0,0],[0,0],[0,0]])
Out[1]: #  shape of output tensor
TensorShape([2, 3, 3, 4])
```

The format of the padding parameter is:

```
padding=[[0,0],[top,bottom],[left,right],[0,0]]
```

For example, if one unit is filled up in all directions (top, bottom, left and right), the padding parameter is as the following:

```
In [2]:
x = tf.random.normal([2,5,5,3]) # input with 3 channels with height and
width 5
# Create w using [k,k,cin,cout] format, 4 3x3 kernels
w = tf.random.normal([3,3,3,4])
# Stride is 1, padding is 0,
out = tf.nn.conv2d(x,w,strides=1,padding=[[0,0],[1,1],[1,1],[0,0]])
Out[2]: # shape of output tensor
TensorShape([2, 5, 5, 4])
```

In particular, by setting the parameters padding='SAME' and strides=1, we can get the same size for the input and output of the convolutional layer, wherein the specific number of padding is automatically calculated by TensorFlow. E.g:

```
In [3]:
x = tf.random.normal([2,5,5,3]) # input
w = tf.random.normal([3,3,3,4]) # 4 3x3 kernels
# Stride is 1,padding is "SAME"
# padding="SAME" gives use same size only when stride=1
out = tf.nn.conv2d(x,w,strides=1,padding='SAME')
Out[3]: TensorShape([2, 5, 5, 4])
```

When $s > 1$, setting padding='SAME' would cause the output height and width to decrease $\frac{1}{s}$ of original size. E.g:

```
In [4]:
x = tf.random.normal([2,5,5,3])
w = tf.random.normal([3,3,3,4])
out = tf.nn.conv2d(x,w,strides=3,padding='SAME')
Out [4]:TensorShape([2, 2, 2, 4])
```

The convolutional neural network layer is the same as the fully connected layer, and the network can be set with a bias vector. The tf.nn.conv2d function does not implement the calculation of the bias vector. We can add the bias manually. E.g:

```
# Create bias tensor
b = tf.zeros([4])
# Add bias to convolution output. It'll broadcast to size of [b,h',w',cout]
out = out + b
```

10.3.2 Convolutional layer classes

Through the convolution layer classes layers.Conv2D, you can directly define the convolution kernel $\boldsymbol{W}$ and bias tensor $\boldsymbol{b}$, and directly call the class instance to complete the forward calculation of the convolution layer. In TensorFlow, the naming of APIs has certain rules. Objects with uppercase letters generally represent classes, and all lowercases generally represent functions, such as layers.Conv2D represents convolutional layer classes, and nn.conv2d represents convolution functions. Using the class method will automatically create the required weight tensor and bias vector. The user does not need to memorize the definition format of the convolution kernel tensor, so it is easier and more convenient to use, but we also lose some flexibility. The function interface needs to define weights and bias by itself, which is more flexible.

When creating a new convolutional layer class, you only need to specify the number of convolution kernel parameters filters, the size of the convolution kernel kernel_size, the stride, padding, etc. A convolutional layer with 4 3×3 convolution kernels is created as follows (the step stride is 1, and the padding scheme is'SAME'):

```
layer = layers.Conv2D(4,kernel_size=3,strides=1,padding='SAME')
```

If the height and width of the convolution kernel are not equal, and the stride along different directions is not equal neither, it is necessary to design the kernel_size parameter in the tuple format (k_h, k_w) and the strides parameter (s_h, s_w). Create 4 3×4 convolution kernels as follows (s_h= 2 in the vertical direction, and s_w = 1 in the horizontal direction):

```
layer = layers.Conv2D(4,kernel_size=(3,4),strides=(2,1),padding='SAME')
```

After the creation is complete, the forward calculation can be completed by calling the instance (__call__ method), for example:

```
In [5]:
layer = layers.Conv2D(4,kernel_size=3,strides=1,padding='SAME')
out = layer(x) # forward calculation
out.shape # shape of output
Out[5]:TensorShape([2, 5, 5, 4])
```

In class Conv2D, the convolution kernel tensor $\boldsymbol{W}$ and bias $\boldsymbol{b}$ are saved, and the list of $\boldsymbol{W}$ and $\boldsymbol{b}$ can be returned directly through the class member trainable_variables. E.g:

```
In [6]:
# Return all trainable variables
layer.trainable_variables
Out[6]:
[<tf.Variable 'conv2d/kernel:0' shape=(3, 3, 3, 4) dtype=float32, numpy=
 array([[[[ 0.13485974, -0.22861657,  0.01000655,  0.11988598],
         [ 0.12811887,  0.20501086, -0.29820845, -0.19579397],
         [ 0.00858489, -0.24469738, -0.08591779, -0.27885547]], …
 <tf.Variable 'conv2d/bias:0' shape=(4,) dtype=float32, numpy=array([0., 0.,
0., 0.], dtype=float32)>]
```

This layer.trainable_variables class member is very useful in obtaining the variables to be optimized in the network layer. You can also directly call class instance layer.kernel, layer.bias to access $\boldsymbol{W}$ and $\boldsymbol{b}$.

10.4 Hands-on LeNet-5

In the 1990s, Yann LeCun et al. proposed a neural network for recognition of handwritten digits and machine-printed character pictures, which was named LeNet-5 [4]. The proposal of LeNet-5 enabled the convolutional neural network to be successfully commercialized at that time, and was widely used in tasks such as postcode and check number recognition. The following figure 10.30 is the network structure diagram of LeNet-5. It accepts digital and character pictures of size 32×32 as input and then passes through the first convolution layer to obtain the tensor with shape $[b,28,28,6]$. After a downsampling layer, the tensor size is reduced to $[b,14,14,6]$. After the second convolutional layer, the tensor shape becomes $[b,10,10,16]$. After similar downsampling layer, the tensor size is reduced to $[b,5,5,16]$. Before entering the fully connected layer, the tensor is converted to shape $[b,400]$ and feed into two fully connected layers with the number of input nodes 120 and 84 respectively. A tensor with shape $[b,84]$ is obtained, and finally goes through the Gaussian connections layer.

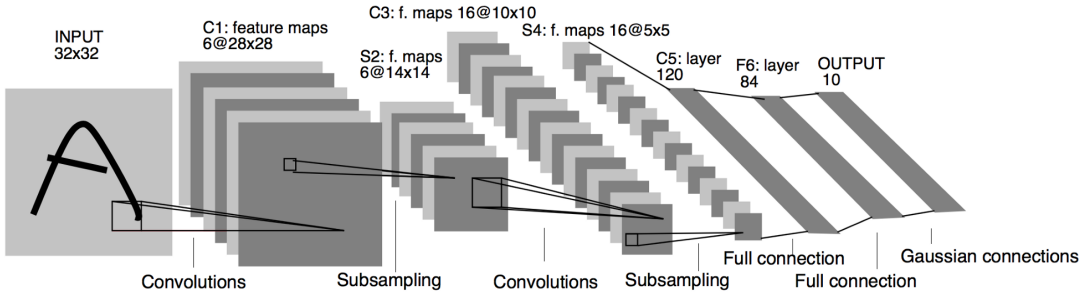


Figure 10.30 LeNet-5 structure [4]

It now appears that the LeNet-5 network has fewer layers (2 convolutional layers and 2 fully connected layers), fewer parameters, and lower computational cost, especially with the support of modern GPUs, which can be trained in minutes.

We have made a few adjustments based on LeNet-5 to make it easier to implement using modern deep learning frameworks. First, we adjust the input shape from 32×32 to 28×28, and then implement the two downsampling layers as the maximum pooling layer (reducing the height and width of the feature map, which will be introduced later), and finally replacing the Gaussian connections layer with a fully connected layer. The modified network is also referred to as the LeNet-5 network hereinafter. The network structure diagram is shown in Figure 10.31.

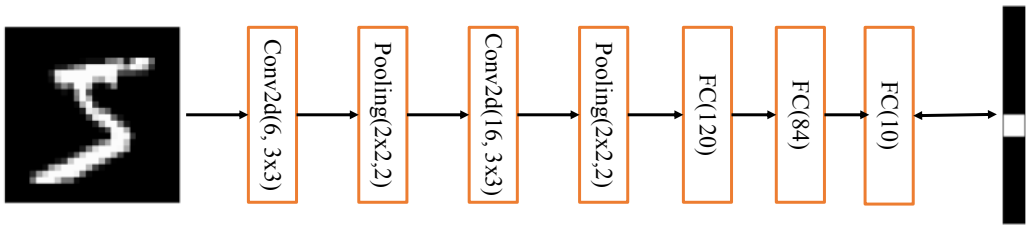


Figure 10.31 Modified LeNet-5 structure

We train the LeNet-5 network based on the MNIST handwritten digital picture data set and test its final accuracy. We have already introduced how to load the MNIST dataset in TensorFlow, so I won't go into details here.

First create LeNet-5 through the Sequential container as follows:

```
from tensorflow.keras import Sequential

network = Sequential([
    layers.Conv2D(6,kernel_size=3,strides=1), # Convolutional layer with 6
3x3 kernels
    layers.MaxPooling2D(pool_size=2,strides=2), # Pooling layer with size 2
    layers.ReLU(), # Activation function
    layers.Conv2D(16,kernel_size=3,strides=1), # Convolutional layer with 16
3x3 kernels

    layers.MaxPooling2D(pool_size=2,strides=2), # Pooling layer with size 2
    layers.ReLU(), # Activation function
    layers.Flatten(), # Flatten layer

    layers.Dense(120, activation='relu'), # Fully-connected layer
    layers.Dense(84, activation='relu'), # Fully-connected layer
    layers.Dense(10) # Fully-connected layer
                    ])
# build the network
network.build(input_shape=(4, 28, 28, 1))
# network summary
network.summary()
```

The summary () function counts the parameters of each layer, print out the network structure information and details of the parameters of each layer, as shown in Table 10.3, we can compare with the parameter scale of the fully connected network 10.1.

Table 10.3 Network parameter statistics

Layer	Convolutional layer 1	Convolutional layer 2	Fully connected layer 1	Fully connected layer 2	Fully connected layer 3

Paramater amount	60	880	48120	10164	850

It can be seen that the parameter amount of the convolutional layer is very small, and the main parameter amount is concentrated in the fully connected layer. Because the convolutional layer reduces the input feature dimension a lot, the parameter amount of the fully connected layer is not too large. The parameter amount of the entire model is about 60K, and the number of fully connected network parameters in Table 10.1 reaches 340,000, so convolutional neural networks can significantly reduce the amount of network parameters while increasing the depth of the network.

In the training phase, first add a dimension ($[b, 28,28,1]$) to the original input of shape $[b, 28,28]$ in the data set, and send it to the model for forward calculation to obtain the output tensor with shape $[b, 10]$. We create a new cross-entropy loss function class for processing classification tasks. By setting the from_logits=True flag, the softmax activation function is implemented in the loss function, and there is no need to manually add the loss function, which improves numerical stability. Code is as below:

```
from tensorflow.keras import losses, optimizers
# Create loss function
criteon = losses.CategoricalCrossentropy(from_logits=True)
```

The training implementation is as follows：

```
    # Create Gradient tape environment
    with tf.GradientTape() as tape:
        # Expand input dimension =>[b,28,28,1]
        x = tf.expand_dims(x,axis=3)
        # Forward calculation, [b, 784] => [b, 10]
        out = network(x)
        # One-hot encoding, [b] => [b, 10]
        y_onehot = tf.one_hot(y, depth=10)
        # Calculate cross-entropy
        loss = criteon(y_onehot, out)
```

After obtaining the loss value, the gradient between the loss and the network parameter network.trainable_variables is calculated by TensorFlow's gradient recorder tf.GradientTape(), and the network weight parameter is automatically updated by the optimizer object as below:

```
    # Calcualte gradient
    grads = tape.gradient(loss, network.trainable_variables)
    # Update paramaters
    optimizer.apply_gradients(zip(grads, network.trainable_variables))
```

The training can be completed after repeating the above steps several times.

In the testing phase, since there is no need to record gradient information, the code generally does not need to be written in the environment “with tf.GradientTape() as tape”. After the output obtained by the forward calculation passes the softmax function, we get the probability P that the network predicts that the current picture $\boldsymbol{x}$ belongs to the category i ($i \in [0,9]$). Use the argmax

function to select the index of the element with the highest probability as the current prediction category, compare it with the real label, and calculate the number of True samples in the comparison result. The number of samples with correct predictions divided by the total sample number gives us the test accuracy of the network.

```
        # Use correct to record the number of correct predictions
        # Use total to record the total number
        correct, total = 0,0
        for x,y in db_test: # Loop through all samples
            # Expand dimension =>[b,28,28,1]
            x = tf.expand_dims(x,axis=3)
            # Forward calculation to get probability, [b, 784] => [b, 10]
            out = network(x)
            # Techinically, we should pass out to softmax() function firs.
             # But because softmax() doesn't change the order the numbers, we
        omit the softmax() part.
            pred = tf.argmax(out, axis=-1)
            y = tf.cast(y, tf.int64)
            # Calculate the correct prediction number
            correct += float(tf.reduce_sum(tf.cast(tf.equal(pred, y),tf.floa
t32)))
            # Total sample number
            total += x.shape[0]
        # Calculate accuracy
        print('test acc:', correct/total)
```

After cyclically training 30 Epochs on the data set, the training accuracy of the network reached 98.1%, and the test accuracy also reached 97.7%. For the simple handwritten digital picture recognition tasks, the old LeNet-5 network can already achieve good results, but for slightly more complex tasks, such as color animal picture recognition, LeNet-5 performance will drop sharply.

10.5 Representation learning

We have introduced the working principle and implementation method of the convolutional neural network layer. The complex convolutional neural network model is also based on the stacking of convolutional layers. In the past, researchers have discovered that the deeper the network layer, the stronger the model's expressive ability, and the more likely it is to achieve better performance. So what are the characteristics of the stacked convolutional network, so that the deeper the layer, the stronger the network's expressive ability?

In 2014, Matthew D. Zeiler et al. [5] tried to use visual methods to understand exactly what convolutional neural networks learned. By mapping the feature map of each layer back to the input picture using the "Deconvolutional Network", we can view the learned feature distribution, as shown in Figure 10.32. It can be observed that the features of the second layer correspond to the extraction of the underlying images such as edges, corners, colors, etc.; the third layer starts to

capture the middle features of texture; the fourth and fifth layers present some features of the object, such as puppy faces, bird's feet and other high-level features. Through these visualizations, we can experience the feature learning process of the convolutional neural network to a certain extent.

Figure 10.32 Visualization of Convolutional Neural Network features [5]

The image recognition process is generally considered to be a representation learning process. Starting from the original pixel features received, it gradually extracts low-level features such as edges and corners, then mid-level features such as textures, and then high-level features such as object parts. The last network layer learns classification logic based on these learned abstract feature representations. The higher the layer and the more accurate the learned features, the more favorable the classification of the classifier is, thereby obtaining better performance. From the perspective of representation learning, convolutional neural networks extract features layer by layer, and the process of network training can be considered as a feature learning process. Based on the learned high-level abstract features, classification tasks can be conveniently performed.

Applying the idea of representation learning, a well-trained convolutional neural network can often learn better features. This feature extraction method is generally universal. For example, learning the representation of head, foot, body and other characteristics on cat and dog tasks can also be used to some extent on other animals. Based on this idea, the first few feature extraction layers of the deep neural network trained on task A can be migrated to task B, and only the classification logic of task B (represented as the last layer of the network) needs to be trained. This method is a type of transfer learning, also known as fine-tuning.

10.6 Gradient propagation

After completing the handwritten digital image recognition exercise, we have a preliminary understanding of the use of convolutional neural networks. Now let's solve a key problem. The convolutional layer implements discrete convolution operations by moving the receptive field. So how does its gradient propagation work?

Consider a simple case where the input is a 3×3 single-channel matrix, and a 2×2 convolution kernel is used to perform the convolution operation. We then calculate the error between the flattened output and the corresponding label, as shown in Figure 10.33. Let's discuss the gradient update method for this case.

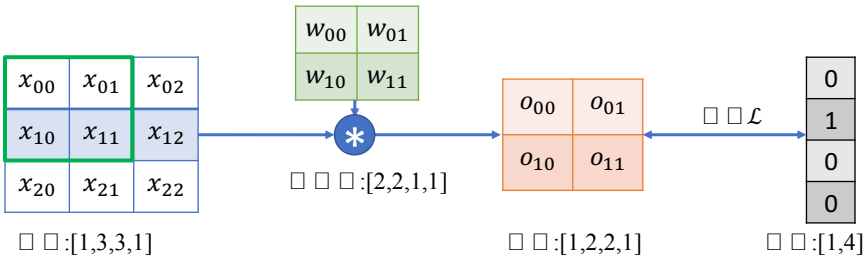


Figure 10.33 Gradient propagation example for the convolutional layer

First derive the expression of the output tensor $\boldsymbol{O}$:

$$o_{00} = x_{00}w_{00} + x_{01}w_{01} + x_{10}w_{10} + x_{11}w_{11} + b$$
$$o_{01} = x_{01}w_{00} + x_{02}w_{01} + x_{11}w_{10} + x_{12}w_{11} + b$$
$$o_{10} = x_{10}w_{00} + x_{11}w_{01} + x_{20}w_{10} + x_{21}w_{11} + b$$
$$o_{11} = x_{11}w_{00} + x_{12}w_{01} + x_{21}w_{10} + x_{22}w_{11} + b$$

Taking w_{00} gradient calculation as an example, decompose by chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_{00}} = \sum_{i\in\{00,\,01,\,10,\,11\}} \frac{\partial \mathcal{L}}{\partial o_i}\frac{\partial o_i}{\partial w_{00}}$$

where $\frac{\partial \mathcal{L}}{\partial o_i}$ can be directly derived from the error function. Let's consider $\frac{\partial o_i}{\partial w_i}$:

$$\frac{\partial o_{00}}{\partial w_{00}} = \frac{\partial(x_{00}w_{00} + x_{01}w_{01} + x_{10}w_{10} + x_{11}w_{11} + b)}{w_{00}} = x_{00}$$

Similarly, one can derive:

$$\frac{\partial o_{01}}{\partial w_{00}} = \frac{\partial(x_{01}w_{00} + x_{02}w_{01} + x_{11}w_{10} + x_{12}w_{11} + b)}{w_{00}} = x_{01}$$

$$\frac{\partial o_{10}}{\partial w_{00}} = \frac{\partial(x_{10}w_{00} + x_{11}w_{01} + x_{20}w_{10} + x_{21}w_{11} + b)}{w_{00}} = x_{10}$$

$$\frac{\partial o_{11}}{\partial w_{00}} = \frac{\partial(x_{11}w_{00} + x_{12}w_{01} + x_{21}w_{10} + x_{22}w_{11} + b)}{w_{00}} = x_{11}$$

It can be observed that the method of cyclically moving the receptive field does not change the derivatization of the network layer, and the derivation of the gradient is not complicated. But when the number of network layers increases, the artificial gradient derivation will become very cumbersome. But don't worry, the deep learning framework can help us automatically complete the gradient calculation and update of all parameters, we only need to design the network structure.

10.7 Pooling layer

In the convolutional layer, the height and width of the feature map can be reduced by adjusting the stride size parameter s, thereby reducing the amount of network parameters. In fact, in addition to setting the stride size, there is a special network layer that can reduce the parameter amount as well, which is known as the Pooling Layer.

The pooling layer is also based on the idea of local correlation. By sampling or aggregating information from a group of locally related elements, we can obtain new element values. In particular, the Max Pooling layer selects the largest element value from the local related element set, and the Average Pooling layer calculates the average value from the local related element set. Taking a 5×5 Max Pooling layer as an example, suppose the receptive field window size $k = 2$ and stride $s = 1$, as shown in Figure 10.34. The green dotted box represents the position of the first receptive field, and the set of receptive field elements is

$$\{1, -1, -1, -2\}$$

According Max Pooling, we have

$$x' = \max(\{1, -1, -1, -2\}) = 1$$

If the Average Pooling operation is used, the output value would be

$$x' = \text{avg}(\{1, -1, -1, -2\}) = -0.75$$

After calculating the receptive field of the current position, similar to the calculation step of the convolutional layer, the receptive field is moved to the right by several units according to the stride size. The output becomes

$$x' = \max(-1,0, -2,2) = 2$$

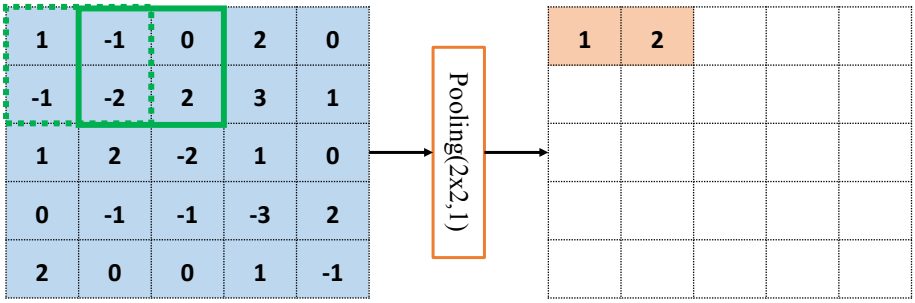


Figure 10.34 Max Pooling example-1

In the same way, gradually move the receptive field window to the far right and calculate the output $x' = \max(2,0,3,1) = 1$. At this time, the window has reached the input edge. The receptive field window moves down by one stride and returns to the beginning of the line, as shown in Figure 10.35.

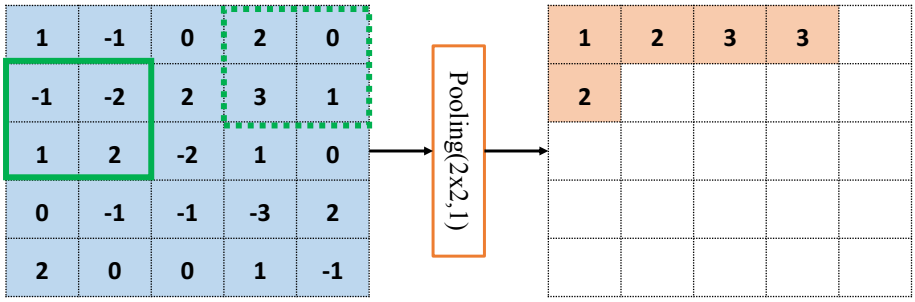


Figure 10.35 Max Pooling example -2

Loop back and forth until we reach the bottom and right, we get the output of the Max Pooling layer as shown in Figure 10.36. The length and width are slightly smaller than the input height and width.

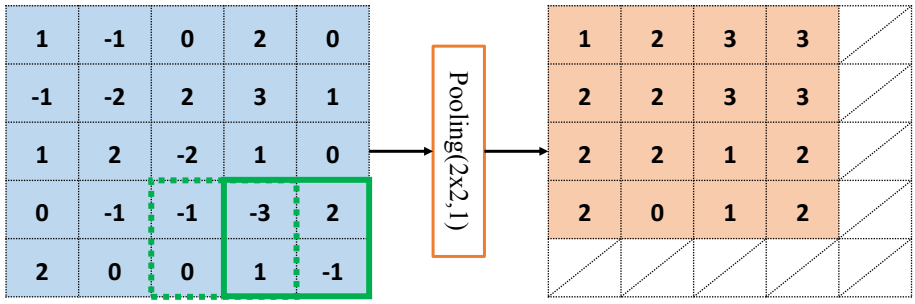


Figure 10.36 Max Pooling example -3

Because the pooling layer has no parameters to learn, the calculation is simple, and the size of the feature map can be effectively reduced, it is widely used in computer vision-related tasks.

By carefully designing the height, width k and stride parameter s of the receptive field of the pooling layer, various dimensionality reduction operations can be realized. For example, a common pooling layer setting is $k = 2,\ s = 2$, which can achieve the purpose of outputting only half of the input height and width. As shown in Figure 10.37 and Figure 10.38 below, the receptive field $k = 3$, stride size $s = 2$, input $\boldsymbol{X}$ has height and width 5×5, but the output only has height and width 2×2.

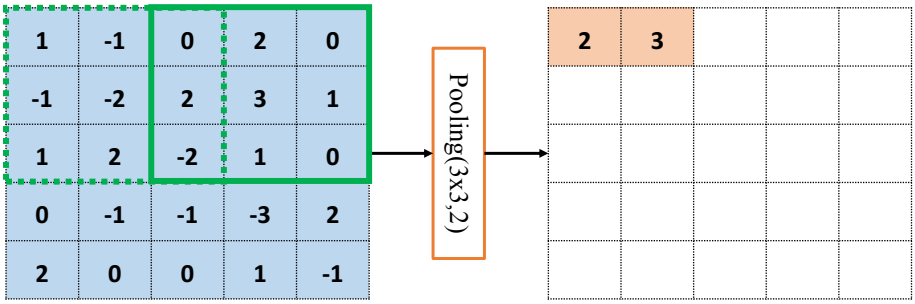


Figure 10.37 Pooling layer example (half size output)-1

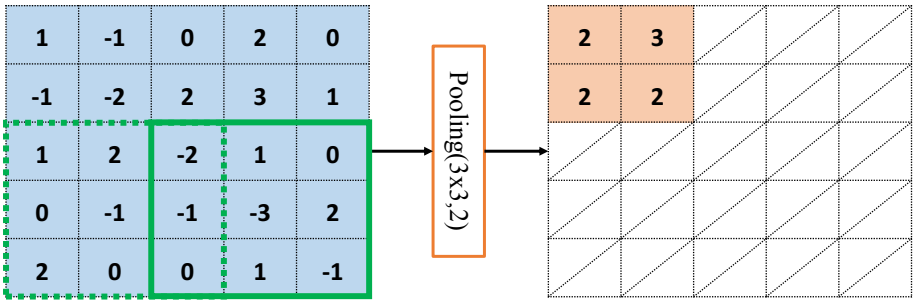


图 10.38 Pooling layer example (half size output)-2

10.8 BatchNorm layer

With the advent of convolutional neural networks, the amount of network parameters has been greatly reduced, making it possible for deep networks with dozens of layers. However, before the emergence of the residual network, the increasing number of neural network layers makes the training very unstable, and sometimes the network does not update or even does not converge for a long time. At the same time, the network is more sensitive to hyperparameters, and the slight change of hyperparameters will change training trajectory of the network completely.

In 2015, Google researcher Sergey Ioffe et al. proposed a method of parameter normalization and designed the Batch Normalization (abbreviated as BatchNorm, or BN) layer [6]. The proposal of the BN layer makes the setting of network hyperparameters more free, such as a larger learning rate, and more random network initialization. In the meantime, the network has a faster convergence speed and better performance. After the BN layer was proposed, it was widely used

in various deep network models. The convolutional layer, BN layer, ReLU layer, and pooling layer once became the standard unit blocks of network models. The stacking Conv-BN-ReLU-Pooling method often generates good model performance.

Why do we need to normalize the data in the network? It is difficult to explain this problem thoroughly from a theoretical level, even the explanation given by the author of the BN layer may not convince everyone. Rather than entangle the reasons, it is better to experience the benefits of data normalization through specific questions.

Consider the Sigmoid activation function and its gradient distribution. As shown in Figure 10.39 below, the derivative value of the Sigmoid function in the interval $x \in [-2,2]$ is distributed in the interval $[0.1, 0.25]$. When x > 2 or x < -2, the derivative of the Sigmoid function becomes very small, approaching 0, which is prone to gradient dispersion. In order to avoid the gradient dispersion phenomenon of the Sigmoid function due to too large or too small input, it is very important to normalize the function input to a small interval near 0. It can be seen from Figure 10.39 that after normalization, the value is mapped near 0, and the derivative value here is not too small, so that gradient dispersion is not easy to appear. This is an example of the benefit of normalization.

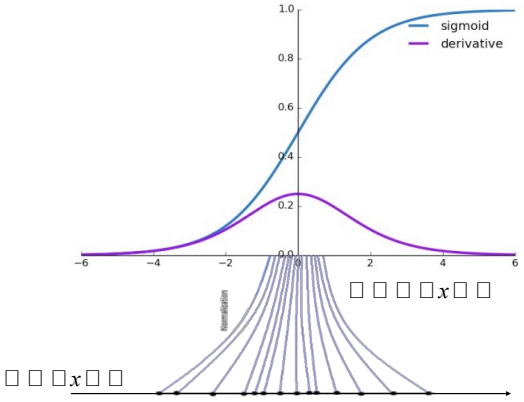


Figure 10.39 Sigmoid function and its derivative

Let's look at another example. Consider a linear model with 2 input nodes, as shown in Figure 10.40(a):

$$\mathcal{L} = a = x_1 w_1 + x_2 w_2 + b$$

Discuss the optimization problems under the following two input distributions:

- $x_1 \in [1,10]$，$x_2 \in [1,10]$
- $x_1 \in [1,10]$，$x_2 \in [100,1000]$

Because the model is relatively simple, two types of contour maps of the loss function can be drawn. Figure 10.40(b) is a schematic diagram of an optimized trajectory when $x_1 \in [1,10]$ and $x_2 \in [100,1000]$, and Figure 10.40(c) is a schematic diagram of an optimized trajectory when $x_1 \in [1,10]$ and $x_2 \in [1,10]$. The center of the ring in the figure is the global extreme point.

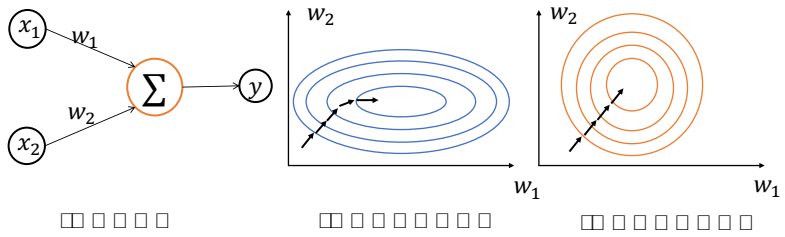


Figure 10.40 An example of data normalization

Consider

$$\frac{\partial \mathcal{L}}{\partial w_1} = x_1$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = x_2$$

When the input distributions are similar, and the partial derivative values are the same, the optimized trajectory of the function is shown in Figure 10.40(c); when the input distributions differ greatly, for example $x_1 \ll x_2$,

$$\frac{\partial \mathcal{L}}{\partial w_1} \ll \frac{\partial \mathcal{L}}{\partial w_2}$$

The equipotential line of the loss function is steeper on the axis, and a possible optimization trajectory is shown in Figure 10.40(b). Comparing the two optimized trajectories, it can be observed that when the distributions of x_1 and x_2 are similar, the convergence in Figure 10.40(c) is faster and the optimized trajectory is more ideal.

Through the above two examples, we can empirically conclude: when the network layer input distribution is similar, and the distribution is in a small range (such as near 0), it favors the function optimization more. So how to ensure that the input distribution is similar? Data normalization can achieve this purpose, and data can be mapped to:

$$\hat{x} = \frac{x - \mu_r}{\sqrt{{\sigma_r}^2 + \epsilon}}$$

Where μ_r is the mean and ${\sigma_r}^2$ the variance of all data, ϵ is a small number, such as $1\mathrm{e}-8$.

In the batch-based training phase, how to obtain all the input statistics μ_r and ${\sigma_r}^2$ of each network layer? Consider the mean μ_B and variance ${\sigma_B}^2$ within the Batch:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$${\sigma_B}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

It can be regarded as approximate of μ_r and ${\sigma_r}^2$, where m is the number of Batch samples. Therefore, in the training phase, through normalization

$$\hat{x}_{\text{train}} = \frac{x_{\text{train}} - \mu_B}{\sqrt{{\sigma_B}^2 + \epsilon}}$$

and approximate the overall mean μ_r and variance ${\sigma_r}^2$ using each batch's mean μ_B and variance ${\sigma_B}^2$.

In the test phase, we can normalize the test data using

$$\hat{x}_{\text{test}} = \frac{x_{\text{test}} - \mu_r}{\sqrt{{\sigma_r}^2 + \epsilon}}$$

The above operation does not introduce additional variables to be optimized, and the mean and variance are obtained through existing data, and do not need to participate in gradient update. In fact, in order to improve the expressive ability of the BN layer, the author of the BN layer introduced the "scale and shift" technique to map and transform the variables again:

$$\tilde{x} = \hat{x} \cdot \gamma + \beta$$

where the parameter γ scales the normalize variable again, and the parameter β realizes the translation operation. The difference is that the parameters γ and β are automatically optimized by the backpropagation algorithm to achieve the purpose of scaling and panning data distribution "on demand" at the network layer.

Let's learn how to implement the BN layer in TensorFlow.

10.8.1 Forward propagation

We denote the input of the BN layer as x and the output as $\tilde{x}$. The forward propagation process is discussed in training phase and testing phase.

Training phase: first calculate the current batch's mean μ_B and variance ${\sigma_B}^2$, and then normalize the data according to

$$\tilde{x}_{\text{train}} = \frac{x_{\text{train}} - \mu_B}{\sqrt{{\sigma_B}^2 + \epsilon}} \cdot \gamma + \beta$$

We then use

$$\mu_r \leftarrow \text{momentum} \cdot \mu_r + (1 - \text{momentum}) \cdot \mu_B$$
$${\sigma_r}^2 \leftarrow \text{momentum} \cdot {\sigma_r}^2 + (1 - \text{momentum}) \cdot {\sigma_B}^2$$

to iteratively update the statistical values μ_r and ${\sigma_r}^2$ of the global training data, where momentum is a hyperparameter that needs to be set to balance the update amplitude: when $\text{momentum} = 0$, μ_r and ${\sigma_r}^2$ are directly set as μ_B and ${\sigma_B}^2$ of the latest batch; when $\text{momentum} = 1$, μ_r and ${\sigma_r}^2$ remain unchanged. In TensorFlow, momentum is set to 0.99 by default.

Test phase: the BN layer uses

$$\tilde{x}_{\text{test}} = \frac{x_{\text{test}} - \mu_r}{\sqrt{{\sigma_r}^2 + \epsilon}} * \gamma + \beta$$

to calculate $\tilde{x}_{test}$, where $\mu_r, {\sigma_r}^2, \gamma, \beta$ come from the statistics or optimization results of the training phase, and are used directly in the test phase, and these parameters are not updated.

10.8.2 Backward propagation

In the backward update phase, the back propagation algorithm solves the gradients $\frac{\partial \mathcal{L}}{\partial \gamma}$ and $\frac{\partial \mathcal{L}}{\partial \beta}$ of the loss function, and automatically optimizes the parameters γ and βaccording to the gradient update rule.

It should be noted that for 2D feature map input $\boldsymbol{X}$: $[b, h, w, c]$, the BN layer does not calculate μ_B and ${\sigma_B}^2$ of every point, instead, it calculates μ_B and ${\sigma_B}^2$ on each channel on the channel axis c, so μ_B and ${\sigma_B}^2$ are the mean and variance of all other dimensions on each channel . Taking the input of shape $[100,32,32,3]$ as an example, the mean value on the channel axis c is calculated as follows:

```
In [7]:
x=tf.random.normal([100,32,32,3])
# Combine other dimensions except the channel dimension
x=tf.reshape(x,[-1,3])
# Calculate mean
ub=tf.reduce_mean(x,axis=0)
ub
Out[7]:
<tf.Tensor: id=62, shape=(3,), dtype=float32, numpy=array([-0.00222636, -
0.00049868, -0.00180082], dtype=float32)>
```

The has c channels, so c averaged values are generated.

In addition to the method of statistical data on the axis c, we can also easily extend the method to other dimensions, as shown in Figure 10.41:

- Layer Norm：Calculate the mean and variance of all features of each sample
- Instance Norm：Calculate the mean and variance of features on each channel of each sample
- Group Norm：Divide c channel into several groups, and count the feature mean and variance in the channel group of each sample

The Normalization method mentioned above is proposed by several independent papers, and it has been verified that it is equivalent or superior to the BatchNorm algorithm in some applications. It can be seen that the research of deep learning algorithms is not difficult. As long as you think more and practice your engineering ability, everyone will have the opportunity to publish innovative results.

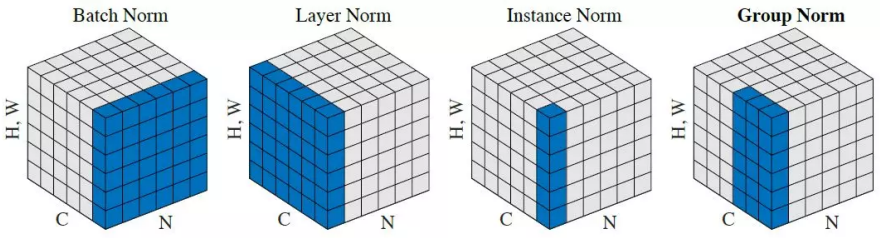


Figure 10.41 Different normalization illustration [7]

10.8.3 Implementation of BatchNormalization layer

In TensorFlow, the BN layer can be easily implemented through the layers.BatchNormalization() class:

```
# Create BN layer
layer=layers.BatchNormalization()
```

Different from the fully connected layer and the convolutional layer, the behavior of the BN layer in the training phase and the test phase is different. It is necessary to distinguish the training mode from the test mode by setting the training flag.

Take the network model of LeNet-5 as an example, add the BN layer after the convolutional layer, the code is as follows:

```
network = Sequential([
    layers.Conv2D(6,kernel_size=3,strides=1),
    # Insert BN layer
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=2,strides=2),
    layers.ReLU(),
    layers.Conv2D(16,kernel_size=3,strides=1),
    # Insert BN layer
    layers.BatchNormalization(),
    layers.MaxPooling2D(pool_size=2,strides=2),
    layers.ReLU(),
    layers.Flatten(),
    layers.Dense(120, activation='relu'),
    layers.Dense(84, activation='relu'),
    layers.Dense(10)
                    ])
```

In the training phase, you need to set the network parameter `training=True` to distinguish whether the BN layer is a training or testing model. The code is as follows:

```
    with tf.GradientTape() as tape:
        # Insert channel dimension
        x = tf.expand_dims(x,axis=3)
        # Forward calculation, [b, 784] => [b, 10]
        out = network(x, training=True)
```

In the testing phase, you need to set `training=False` to avoid wrong behavior in the BN layer. The code is as follows:

```
        for x,y in db_test:
            # Insert channel dimension
            x = tf.expand_dims(x,axis=3)
            # Forward calculation
```

```
out = network(x, training=False)
```

10.9 Classical Convolutional Network

Since the introduction of AlexNet [3] in 2012, a variety of deep convolutional neural network models have been proposed, among which the more representative ones are the VGG series [8], the GoogLeNet series [9], the ResNet series [10], and the DenseNet series [11]. The overall trend of their network layers is gradually increasing. Take the classification performance of the network model on the ImageNet dataset of the ILSVRC Challenge as an example. As shown in Figure 10.42, the network models before the emergence of AlexNet were all shallow neural networks, and the Top-5 error rate was above 25%. The AlexNet 8-layer deep neural network reduced the Top-5 error rate to 16.4%, and the performance was greatly improved. The subsequent VGG and GoogleNet models continued to reduce the error rate to 6.7%; the emergence of ResNet increased the number of network layers to 152 layers for the first time. The error rate is also reduced to 3.57%.

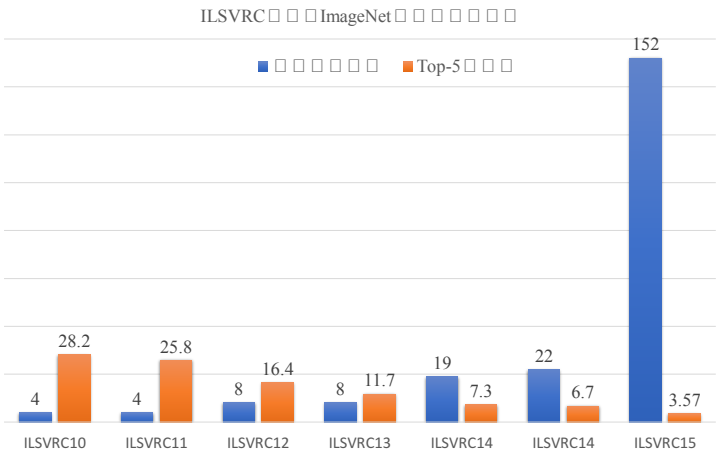


Figure 10.42 Model performance on classification tasks of ImageNet dataset

This section will focus on the characteristics of these network models.

10.9.1 AlexNet

In 2012, Alex Krizhevsky, the champion of the ImageNet data set classification task of the ILSVRC12 Challenge, proposed an 8-layer deep neural network model AlexNet, which receives the input size of 224×224 color image data and gets the probability distribution of 1000 categories after five convolutional layers and three fully connected layers. In order to reduce the dimensionality of the feature map, AlexNet added the Max Pooling layer after the first, second, and fifth convolutional layers. As shown in Figure 10.43, the number of parameters of the network reached 60 million. In order to train the model on NVIDIA GTX 580 GPU (3GB GPU memory) at the time, Alex Krizhevsky disassembled the convolutional layer and the first two fully connected layers on two GPUs for training separately, and merged the last layer into one GPU to do backward udpate. AlexNet achieved a Top-5 error rate of 15.3% in ImageNet, which is 10.9%

lower than the second place.

The innovations of AlexNet are:

- The number of layers has reached 8.
- Uses the ReLU activation function. Most of previous neural networks use the Sigmoid activation function, which is relatively complicated to calculate and is prone to gradient dispersion.
- Introduces the Dropout layer. Dropout improves the generalization ability of the model and prevents overfitting.

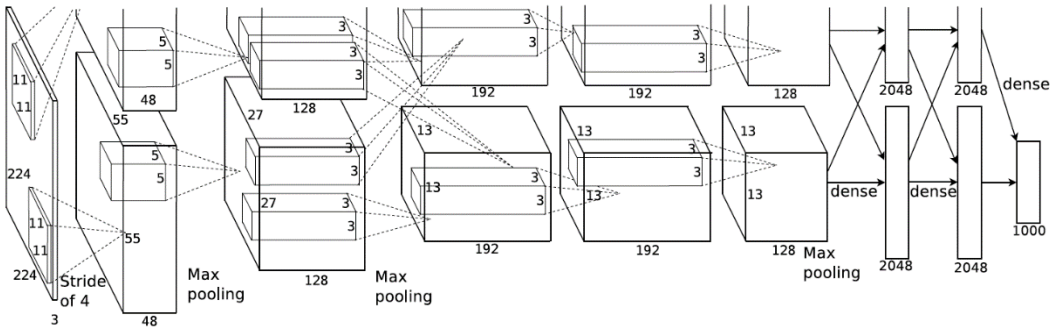


Figure 10.43 AlexNet architecture [3]

10.9.2 VGG series

The superior performance of the AlexNet model has inspired the industry to move in the direction of deeper network models. In 2014, the runner-up of the ImageNet classification task of the ILSVRC14 challenge, the VGG Lab of the University of Oxford, proposed a series of network models such as VGG11, VGG13, VGG16, and VGG19 (Figure 10.45), and increased the network depth to up to 19 layers [8]. Take VGG16 as an example, it accepts color picture data with size of 224×224, and then passes through 2 Conv-Conv-Pooling units and 3 Conv-Conv-Conv-Pooling units, and finally outputs the probability of current picture belonging to 1000 categories through a 3 fully connected layers as shown in Figure 10.44. VGG16 achieved a Top-5 error rate of 7.4% on ImageNet, which is 7.9% lower than AlexNet's error rate.

The innovations of the VGG series network are:

- The number of layers is increased to 19.
- Uses a smaller 3x3 convolution kernel, which has fewer parameters and lower computational cost compared to the 7x7 convolution kernel in AlexNet.
- Uses a smaller pooling layer window 2×2 and stride size $s=2$, while $s=2$and pooling window is 3x3 in AlexNet.

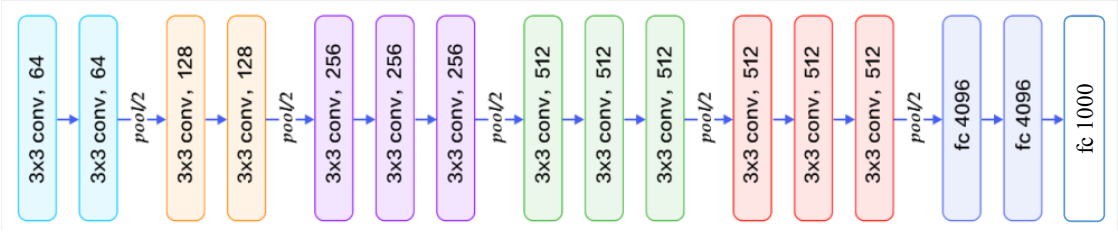


Figure 10.44 VGG16 architecture

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 10.45 VGG series network architecture [8]

10.9.3 GoogLeNet

The number of 3x3 convolution kernel has less parameters, the computational cost is lower, and the performance is even better. Therefore, the industry began to explore the smallest convolution kernel: the 1x1 convolution kernel. As shown in Figure 10.46, the input is a 3-channel 5x5 picture, and the convolution operation is performed with a single 1x1 convolution kernel. The data of each channel is calculated with the convolution kernel of the corresponding channel to obtain the intermediate matrix of the 3 channels, and the corresponding positions are added to get the final output tensor. For the input shape of $[b, h, w, c_{in}]$, the output of the 1x1 convolutional

layer is $[b, h, w, c_{out}]$, where c_{in} is the number of channels of input data, c_{out} is the number of channels of output data, and is also the number of 1x1 convolution kernels. A special feature of the 1x1 convolution kernel is that it can only transform the number of channels without changing the width and height of the feature map.

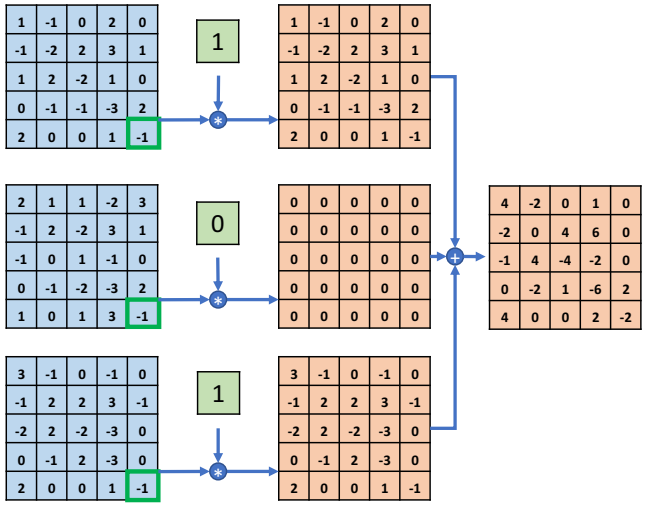


Figure 10.46 1×1 convolutional kernel example

In 2014, Google, the champion of the ILSVRC14 Challenge, proposed a large number of network models using 3x3 and 1x1 convolution kernels: GoogLeNet, with a network layer number of 22 [9]. Although the number of layers of GoogLeNet is much larger than that of AlexNet, its parameter amount is only half of AlexNet, and its performance is much better than AlexNet. On the ImageNet data set classification task, GoogLeNet achieved a Top-5 error rate of 6.7%, which is 0.7% lower than VGG16 in error rate.

The GoogLeNet network adopts the idea of modular design and forms a complex network structure by stacking a large number of Inception modules. As shown in Figure 10.47 below, the input of the Inception module is $\boldsymbol{X}$, and then passes through 4 sub-networks, and finally are spliced and merged on the channel axis to form the output of the Inception module. The 4 sub-networks are:

- 1×1 convolutional layer.
- 1×1 convolutional layer, and then through a 3x3 convolutional layer.
- 1×1 convolutional layer, and then through a 5x5 convolutional layer.
- 3×3 maximum pooling layer, and then through the 1x1 convolutional layer.

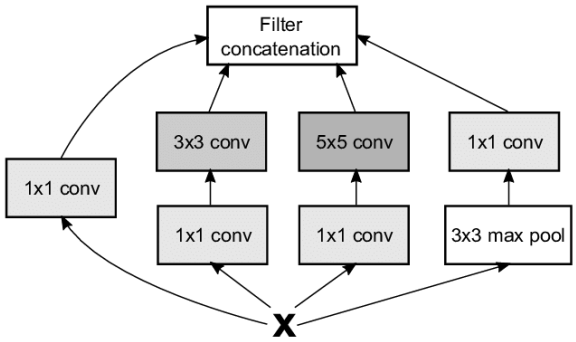


Figure 10.47 Inception module

The network structure of GoogLeNet is shown in Figure 10.48. The network structure in the red box is the network structure in Figure 10.47.

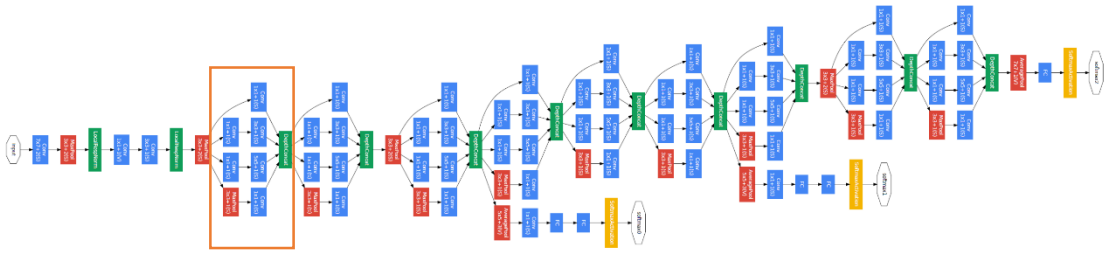


Figure 10.48 GoogleNet architecture [9]

10.10 Hands-on CIFAR10 and VGG13

MNIST is one of the most commonly used data sets for machine learning, but because handwritten digital pictures are very simple, and the MNIST data set only saves image gray information, it is not suitable for inputting a network model designed as RGB three-channel. This section will introduce another classic image classification data set: CIFAR10.

The CIFAR10 data set was released by Canadian Institute for Advanced Research. It contains color pictures of 10 categories of objects such as airplanes, cars, birds, and cats. Each category has collected 6,000 large and small pictures, totaling 60,000 pictures. Among them, 50,000 sheets are used as training data sets, and 10,000 sheets are used as test data sets. Each type of sample is shown in Figure 10.49.

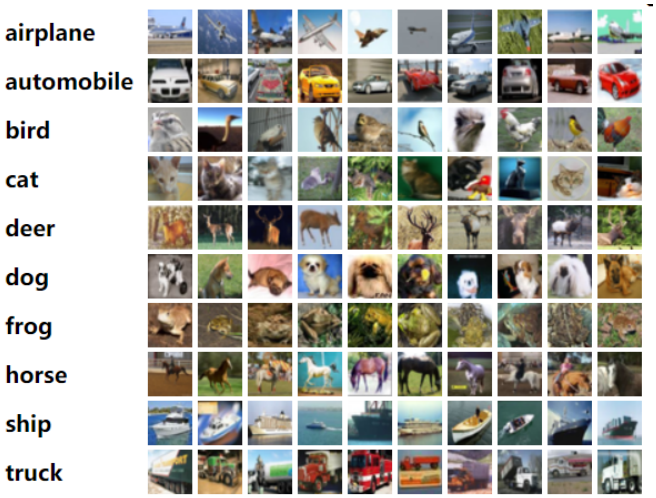


Figure 10.49 CIFAR10 Data Set[①]

In TensorFlow, similarly, there is no need to manually download, parse and load the CIFAR10 data set. The training set and test set can be directly loaded through the datasets.cifar10.load_data() function. For example,

```
# Load CIFAR10 data set
(x,y), (x_test, y_test) = datasets.cifar10.load_data()
# Delete one dimension of y, [b,1] => [b]
y = tf.squeeze(y, axis=1)
y_test = tf.squeeze(y_test, axis=1)
# Print the shape of training and testing sets
print(x.shape, y.shape, x_test.shape, y_test.shape)
# Create training set and preprocess
train_db = tf.data.Dataset.from_tensor_slices((x,y))
train_db = train_db.shuffle(1000).map(preprocess).batch(128)
# Create testing set and preprocess
test_db = tf.data.Dataset.from_tensor_slices((x_test,y_test))
test_db = test_db.map(preprocess).batch(128)
# Select a Batch
sample = next(iter(train_db))
print('sample:', sample[0].shape, sample[1].shape,
      tf.reduce_min(sample[0]), tf.reduce_max(sample[0]))
```

TensorFlow will automatically download the dataset to the path C:\Users\username\.keras\datasets, and users can view it, or manually delete the unnecessary dataset cache. After the above code runs, the shape of $\boldsymbol{X}$ and $\boldsymbol{y}$ in the training set is (50000, 32, 32, 3) and (50000), and the shape of $\boldsymbol{X}$ and $\boldsymbol{y}$ in the test set is (10000, 32, 32, 3) and (10000), which indicates the size of the picture is 32×32, those are color pictures, the number of samples in the training set is 50,000, and the number of samples in the test set is 10,000.

[①] Image source: https://www.cs.toronto.edu/~kriz/cifar.html

CIFAR10 image recognition task is not simple. This is mainly due to the fact that the image content of CIFAR10 requires a lot of details to be presented, and the resolution of the saved images is only 32×32, which makes the subject information blurry and even difficult for human eyes to distinguish. The expression ability of shallow neural networks is limited, and is difficult to reach better performance. In this section, we will modify the VGG13 network structure according to the characteristics of our data set to complete CIFAR10 image recognition as follows:

- Adjust the network input to 32×32. The original network input is 224×224, resulting in too large input feature dimensions and too large network parameters.
- The dimensions of the 3 fully connected layers are $[256,64,10]$ for the setting of 10 classification tasks.

Figure 10.50 is the adjusted VGG13 network structure, which we collectively call the VGG13 network model.

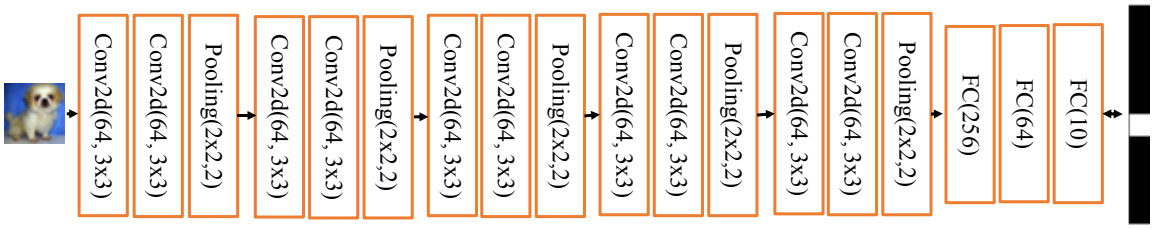


Figure 10.50 Adjusted VGG13 model structure

We implement the network as 2 sub-networks: convolutional sub-network and fully connected sub-network. The convolution sub-network is composed of 5 sub-modules, each of which contains the Conv-Conv-MaxPooling unit structure. The code is as follows:

```
conv_layers = [
    # Conv-Conv-Pooling unit 1
    # 64 3x3 convolutional kernels with same input and output size
    layers.Conv2D(64, kernel_size=[3, 3], padding="same", activation=tf.nn.r
elu),
    layers.Conv2D(64, kernel_size=[3, 3], padding="same", activation=tf.nn.r
elu),
    # Reduce the width and height size to half of its original
    layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='same'),

    # Conv-Conv-Pooling unit 2, output channel increases to 128, half width
and height
    layers.Conv2D(128, kernel_size=[3, 3], padding="same", activation=tf.nn.
relu),
    layers.Conv2D(128, kernel_size=[3, 3], padding="same", activation=tf.nn.
relu),
    layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='same'),

    # Conv-Conv-Pooling unit 3, output channel increases to 256, half width
and height
```

```
    layers.Conv2D(256, kernel_size=[3, 3], padding="same", activation=tf.nn.
relu),
    layers.Conv2D(256, kernel_size=[3, 3], padding="same", activation=tf.nn.
relu),
    layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='same'),

    # Conv-Conv-Pooling unit 4, output channel increases to 512, half width
and height
    layers.Conv2D(512, kernel_size=[3, 3], padding="same", activation=tf.nn.
relu),
    layers.Conv2D(512, kernel_size=[3, 3], padding="same", activation=tf.nn.
relu),
    layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='same'),

    # Conv-Conv-Pooling unit 5, output channel increases to 512, half width
and height
    layers.Conv2D(512, kernel_size=[3, 3], padding="same", activation=tf.nn.
relu),
    layers.Conv2D(512, kernel_size=[3, 3], padding="same", activation=tf.nn.
relu),
    layers.MaxPool2D(pool_size=[2, 2], strides=2, padding='same')
]
conv_net = Sequential(conv_layers)
```

The fully connected sub-network contains 3 fully connected layers, each layer adds a ReLU nonlinear activation function, except for the last layer. Code shows as below:

```
# Create 3 fully connected layer sub-network
fc_net = Sequential([
    layers.Dense(256, activation=tf.nn.relu),
    layers.Dense(128, activation=tf.nn.relu),
    layers.Dense(10, activation=None),
])
```

After the subnet is created, use the following code to view the parameters of the network:

```
# build network and print parameter info
conv_net.build(input_shape=[4, 32, 32, 3])
fc_net.build(input_shape=[4, 512])
conv_net.summary()
fc_net.summary()
```

The total number of parameters of the convolutional network is about 940,000, the total number of parameters of the fully connected network is about 177,000, and the total number of parameters of the network is about 950, 000, which is much less than the original version of VGG13.

Since we implemented the network as two sub-networks, when performing gradient update, it is necessary to merge the parameter of the two sub-networks as below:

```
# merge parameters of two sub-networks
variables = conv_net.trainable_variables + fc_net.trainable_variables
# calculate gradient for all parameters
grads = tape.gradient(loss, variables)
# update gradients
optimizer.apply_gradients(zip(grads, variables))
```

Run the cifar10_train.py file to start training the model. After training 50 Epochs, the test accuracy of the network reached 77.5%.

10.11 Convolutional layer variants

The research of convolutional neural networks has produced a variety of excellent network models, and various variants of convolutional layers have been proposed. This section will focus on several typical convolutional layer variants.

10.11.1 Dilated/Atrous Convolution

In order to reduce the number of parameters of the network, the design of the convolution kernel usually chooses a smaller 1×1 and 3×3 receptive field size. The small convolution kernel makes the network's receptive field area limited when extracting features, but increasing the receptive field area will increase the amount of network parameters and computational costs, so it is necessary to weigh the design.

Dilated/Atrous Convolution is a better solution to this problem. Dilated/Atrous Convolution adds a Dilation Rate parameter to the receptive field of ordinary convolution to control the sampling step size of the receptive field area, as shown in Figure 10.51 below. When the sampling step Dilation Rate of the receptive field is 1, the distance between the sampling points of each receptive field is 1, and the Dilated Convolution at this time degenerates to ordinary convolution; when the Dilation Rate is 2, one point is sampled every two units in the receptive field. As shown in the green grid in the green box in the middle of Figure 10.51, the distance between each sampling grid is 2. Similarly, the Dilation Rate on the right side of Figure 10.51 is 3, and the sampling step is 3. Although the increase in Dilation Rate will increase the area of the receptive field, the actual number of points involved in the calculation remains unchanged.

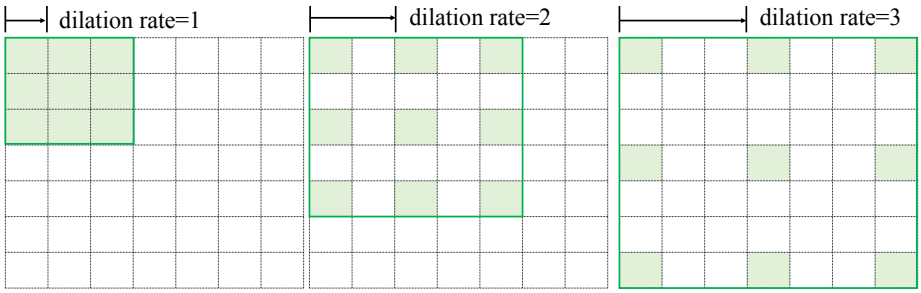


Figure 10.51 Receptive field step length with different dilation rate

Take the single-channel 7×7 tensor and a single 3×3 convolution kernel as an example, as shown in Figure 10.52 below. In the initial position, the receptive field is sampled from the top and right positions, and every other point is sampled. A total of 9 data points are collected, as shown in the green box in Figure 10.52. These 9 data points are multiplied by the convolution kernel and written into the corresponding position of the output tensor.

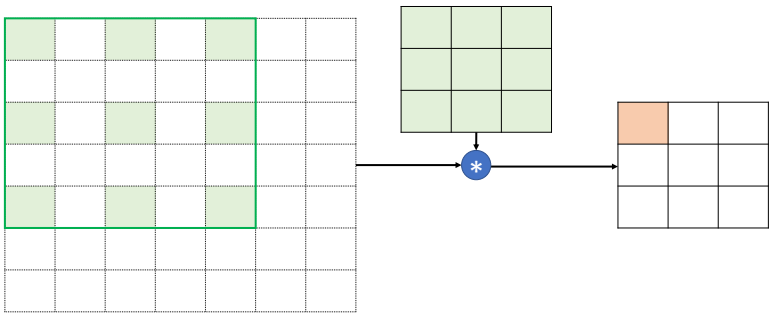


Figure 10.52 Dilated Convolution sample -1

The convolution kernel window moves one unit to the right according to the step size $s=1$, as shown in Figure 10.53. The same interval sampling is carried out. A total of 9 data points are sampled. The multiplication and accumulation operation is completed with the convolution kernel, and the output tensor is written to corresponding position until the convolution kernel moves to the bottom and rightmost position. It should be noted that the moving step size s of the convolution kernel window and the sampling step size Dilation Rate of the receptive field region are different concepts.

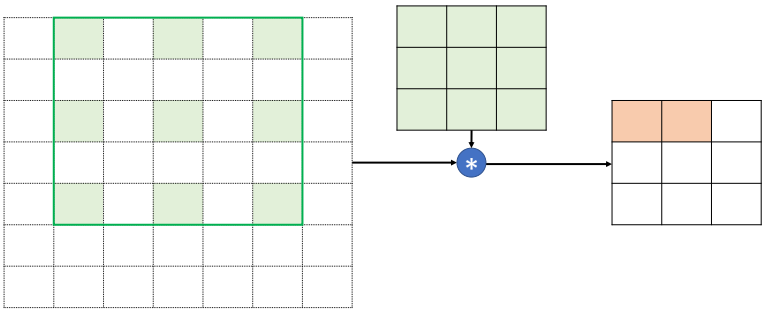


Figure 10.53 Dilated Convolution sample -2

Dilated convolution provides a larger receptive field window without increasing network parameters. However, when setting up a network model using hollow convolution, the Dilation Rate parameter needs to be carefully designed to avoid grid effects. At the same time, a larger

Dilation Rate parameter is not conducive to tasks such as small object detection and semantic segmentation.

In TensorFlow, you can choose to use normal convolution or dilated convolution by setting the dilation_rate parameter of the layers.Conv2D() class. E.g:

```
In [8]:
x = tf.random.normal([1,7,7,1]) # Input
# Dilated convolution, 1 3x3 kernel
layer = layers.Conv2D(1,kernel_size=3,strides=1,dilation_rate=2)
out = layer(x) # forward calculation
out.shape
Out[8]: TensorShape([1, 3, 3, 1])
```

When the dilation_rate parameter is set to the default value 1, the normal convolution method is used for calculation; when the dilation_rate parameter is greater than 1, the dilated convolution method is sampled for calculation.

10.11.2 Transposed Convolution

Transposed Convolution (or Fractionally Strided Convolution, sometimes it is also called Deconvolution. In fact, deconvolution is mathematically defined as the inverse process of convolution, but transposed convolution cannot recover the input of the original convolution, so it is not appropriate to call it deconvolution) by filling a large amount of padding between the inputs to achieve the effect that the output height and width are greater than the input height and width, so as to achieve the purpose of upsampling, as shown in Figure 10.54. We first introduce the calculation process of transposed convolution, and then introduce the relationship between transposed convolution and ordinary convolution.

To simplify the discussion, we only discuss the input with $h = w$, that is, the case where the input height and width are equal.

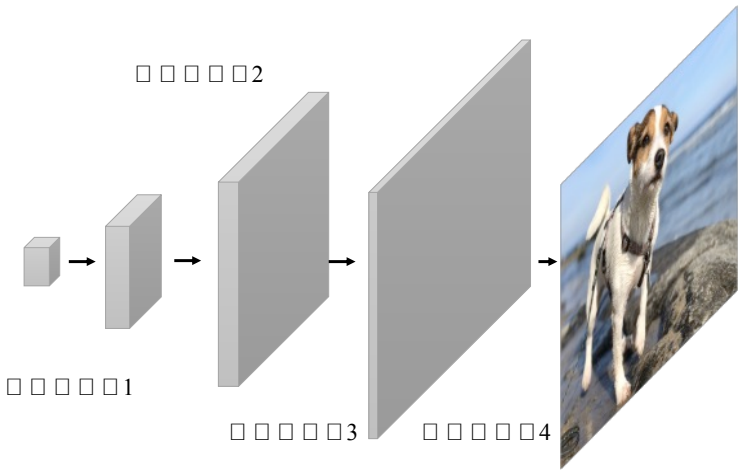


Figure 10.54 Transposed convolution for upsampling

$$o + 2p - k = n * s$$

Consider the following example: the single-channel feature map has 2×2 input, and the transposed convolution kernel is 3×3, $s = 2$, and padding $p = 0$. First, evenly insert $s - 1$ blank data points between the input data points, the resulting matrix is 3×3, as shown in the second matrix in Figure 10.55. Filling the corresponding rows/columns around the 3×3 matrix according to the filling amount $k - p - 1 = 3 - 0 - 1 = 2$. At this time, the height and width of the input tensor are 7×7, as shown in the third matrix in Figure 10.55.

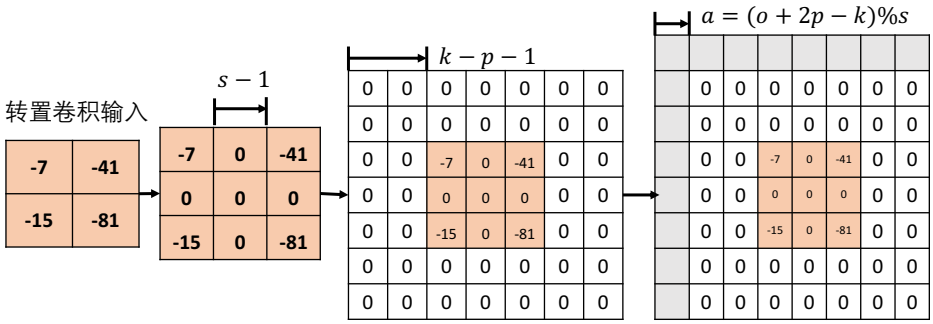


Figure 10.55 Input and padding example

On the 7×7 input tensor, apply the 3×3 convolution kernel operations with stride size $s' = 1$ and padding $p = 0$ (note that the step size s' of the ordinary convolution at this stage is always 1, which is different from the step size s of the transposed convolution). According to the ordinary convolution calculation formula, the output size is:

$$o = \left\lfloor \frac{i + 2 * p - k}{s'} \right\rfloor + 1 = \left\lfloor \frac{7 + 2 * 0 - 3}{1} \right\rfloor + 1 = 5$$

It means 5×5 output size. We directly follow this calculation process to give the final transposed convolution output and input relationship. When 在$o + 2p - k$ is a multiple of s, the relationship is satisfied

$$o = (i - 1)s + k - 2p$$

Transposed convolution is not the inverse process of ordinary convolution, but there is a certain connection between the two, and transposed convolution is also implemented based on ordinary convolution. Under the same setting, the input $\boldsymbol{x}$ is obtained after the ordinary convolution operation $\boldsymbol{o} = \text{Conv}(\boldsymbol{x})$, and sending $\boldsymbol{o}$ to the transposed convolution operation gives $\boldsymbol{x}' = \text{ConvTranspose}(\boldsymbol{o})$, where $\boldsymbol{x}' \neq \boldsymbol{x}$, but with same shape. We can use ordinary convolution operations with input as 5×5, stride size $s = 2$, padding $p = 0$, and 3×3 convolution kernel to verify the demonstration, as shown in Figure 10.56 below.

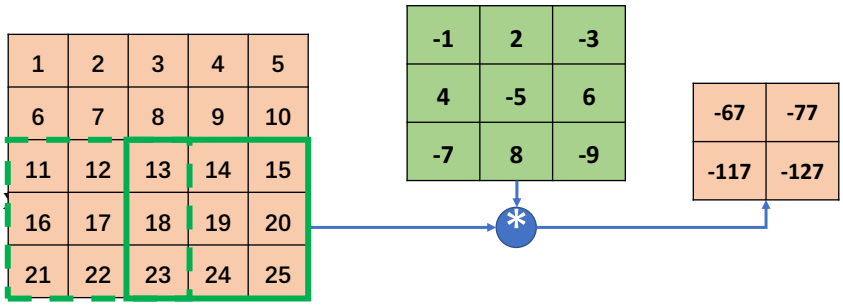


Figure 10.56 Use ordinary convolution to generate same size of input

It can be seen that the output with size 5×5 of the transposed convolution is sent to the ordinary convolution under the same set conditions, and the output of size 2×2 can be obtained. This size is exactly the input size of the transposed convolution. At the same time, we also observe that the output matrix is not exactly the input matrix feeded into the transposed convolution. Transposed convolution and ordinary convolution are not mutually inverse processes, and cannot recover the input content of the other party, but can only recover tensors of equal size. Therefore, it is not appropriate to call it deconvolution.

Based on TensorFlow to implement the transposed convolution operation of the above example, the code is as follows:

```
In [8]:
# Create matrix X with size 5x5
x = tf.range(25)+1
# Reshape X to certain shape
x = tf.reshape(x,[1,5,5,1])
x = tf.cast(x, tf.float32)
# Create constant matrix
w = tf.constant([[-1,2,-3.],[4,-5,6],[-7,8,-9]])
# Reshape dimension
w = tf.expand_dims(w,axis=2)
w = tf.expand_dims(w,axis=3)
# Regular convolution calculation
out = tf.nn.conv2d(x,w,strides=2,padding='VALID')
out
Out[9]: # Output size is 2x2
<tf.Tensor: id=14, shape=(1, 2, 2, 1), dtype=float32, numpy=
array([[[[ -67.],
         [ -77.]],

        [[-117.],
         [-127.]]]], dtype=float32)>
```

Now we use the output of ordinary convolution as the input of transposed convolution to

verify whether the output of transposed convolution is 5×5, the code is as follows:

```
In [10]:
# Transposed convolution calculation
xx = tf.nn.conv2d_transpose(out, w, strides=2,
    padding='VALID',
    output_shape=[1,5,5,1])
Out[10]: # Output size is 5x5
<tf.Tensor: id=117, shape=(5, 5), dtype=float32, numpy=
array([[   67.,  -134.,   278.,  -154.,   231.],
       [ -268.,   335.,  -710.,   385.,  -462.],
       [  586.,  -770.,  1620.,  -870.,  1074.],
       [ -468.,   585., -1210.,   635.,  -762.],
       [  819.,  -936.,  1942., -1016.,  1143.]], dtype=float32)>
```

It can be seen that transposed convolution can recover the input of ordinary convolution of the same size, but the output of transposed convolution is not equivalent to the input of ordinary convolution.

$o + 2p - k \neq n * s$

Let us analyze a detail of the relationship between input and output in the convolution operation in more depth. Consider the output expression of the convolution operation:

$$o = \left\lfloor \frac{i + 2 * p - k}{s} \right\rfloor + 1$$

When the stride size $s > 1$, the round-down operation of $\left\lfloor \frac{i+2*p-k}{s} \right\rfloor$ makes multiple input sizes i correspond to the same output size o. For example, consider the convolution operation with input size 6×6, convolution kernel size 3×3, and stride size 1. The code is as follows:

```
In [11]:
x = tf.random.normal([1,6,6,1])
# 6x6 input
out = tf.nn.conv2d(x,w,strides=2,padding='VALID')
out.shape
x = tf.random.normal([1,6,6,1])...
Out[12]: # Output size 2x2, same as when the input size is 5x5
<tf.Tensor: id=21, shape=(1, 2, 2, 1), dtype=float32, numpy=
array([[[[ 20.438847 ],
         [ 19.160788 ]],

        [[  0.8098897],
         [-28.30303  ]]]], dtype=float32)>
```

In this case, the convolutional output of the same size 2×2 can be obtained as shown in

Figure 10.56. Therefore, convolution operations with different input sizes may obtain the same output. Considering that the input and output relationship between convolution and transposed convolution is interchangeable, from the perspective of transposed convolution, after the input size i is subjected to the transposed convolution operation, different output size o may be obtained. Therefore, by filling the a rows and a columns in Figure 10.55 to achieve different sizes of output o, so as to restore the normal convolution with different sizes of input, the relationship of a is:

$$a = (o + 2p - k)\%s$$

The output of the transposed convolution becomes:

$$o = (i - 1)s + k - 2p + a$$

In TensorFlow, there is no need to manually specify a. We just specify the output size. TensorFlow will automatically derive the number of rows and columns that need to be filled, provided that the output size is legal. E.g:

```
In [13]:
# Get output of size 6x6
xx = tf.nn.conv2d_transpose(out, w, strides=2,
    padding='VALID',
    output_shape=[1,6,6,1])
xx
Out[13]:
<tf.Tensor: id=23, shape=(1, 6, 6, 1), dtype=float32, numpy=
array([[[[ -20.438847 ],
         [  40.877693 ],
         [ -80.477325 ],
         [  38.321575 ],
         [ -57.48236  ],
         [   0.       ]],...
```

The tensor with height and width 5×5 can also be obtained by changing the parameter output_shape=[1,5,5,1].

Matrix transposition

The transposition $\boldsymbol{W}'^{\mathrm{T}}$ of transposed convolution means that the sparse matrix $\boldsymbol{W}'$ generated by the convolution kernel matrix $\boldsymbol{W}$ needs to be transposed first, and then the matrix multiplication operation is performed, while the ordinary convolution does not have the step of transposition. This is why it is called transposed convolution.

Consider the ordinary Conv2d operation: $\boldsymbol{X}$ and $\boldsymbol{W}$, the convolution kernel needs to be cyclically moved in the row and column directions according to the strides to obtain the data of the receptive field involved in the operation, and the "multiply and accumulate" value at each window is calculated serially, which is extremely inefficient. In order to speed up the operation, mathematically, the convolution kernel $\boldsymbol{W}$ can be rearranged into a sparse matrix $\boldsymbol{W}'$ according

to strides, and then the operation $\boldsymbol{W'@X'}$ is completed once (in fact, the matrix $\boldsymbol{W'}$ is too sparse, resulting in many useless 0-multiplication operations, and many deep learning frameworks do not use this implementation).

Take the following convolution kernel as an example: the input $\boldsymbol{X}$ of 4 rows and 4 columns, the height and width as 3×3, stride of 1, and no padding. First, $\boldsymbol{X}$ will be flattened to $\boldsymbol{X'}$, as shown in Figure 10.57.

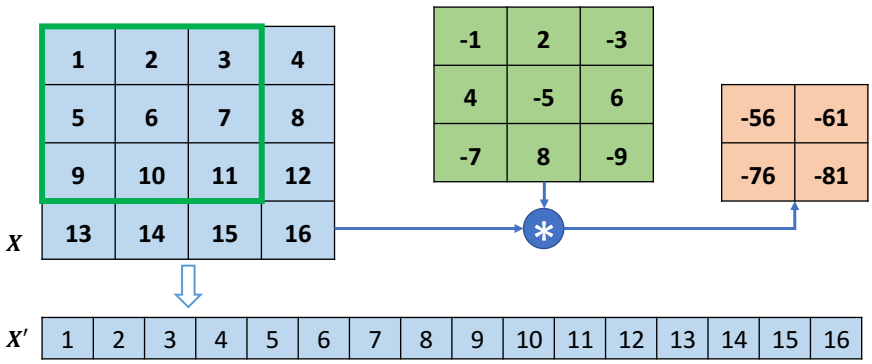


Figure 10.57 Transposed convolution X′

Then convert the convolution kernel $\boldsymbol{W}$ into a sparse matrix $\boldsymbol{W'}$, as shown in Figure 10.58.

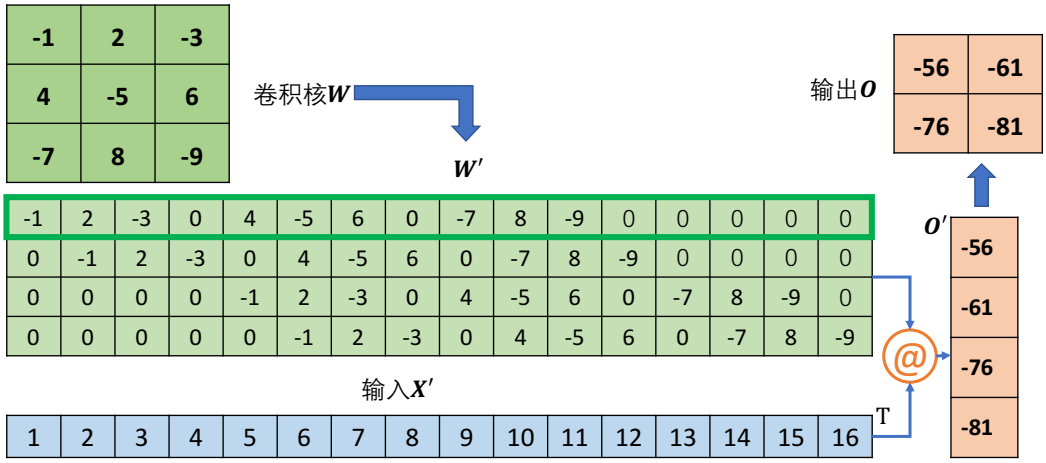


Figure 10.58 Transposed convolution W′

At this time, ordinary convolution operation can be realized by matrix multiplication once:

$$\boldsymbol{O'} = \boldsymbol{W'@X'}$$

If given $\boldsymbol{O}$, how to generate a tensor of the same shape and size as $\boldsymbol{X}$ it? Multiply the transposed matrix $\boldsymbol{W'}$ and the rearranged matrix $\boldsymbol{O'}$ as shown in Figure 10.57:

$$\boldsymbol{X'} = \boldsymbol{W'^{\mathrm{T}}@O'}$$

Reshape $\boldsymbol{X'}$ to the same as the original input size $\boldsymbol{X}$. For example, the shape of $\boldsymbol{O'}$ is $[4,1]$, the shape of $\boldsymbol{W'^{\mathrm{T}}}$ is $[16,4]$, the shape of $\boldsymbol{X'}$ obtained by matrix multiplication is $[16,1]$, and the tensor with shape $[4,4]$ can be generated after reshaping. Since transposed convolution needs to be transposed before it can be multiplied with the input matrix of transposed convolution during matrix operation, it is called transposed convolution.

Transposed convolution has the function of "magnifying feature maps" and has been widely used in generating confrontation networks and semantic segmentation. For example, the generator in DCGAN [12] achieves layer-by-layer "magnification" by stacking transposed convolution layers, and finally get a very realistic generated picture.

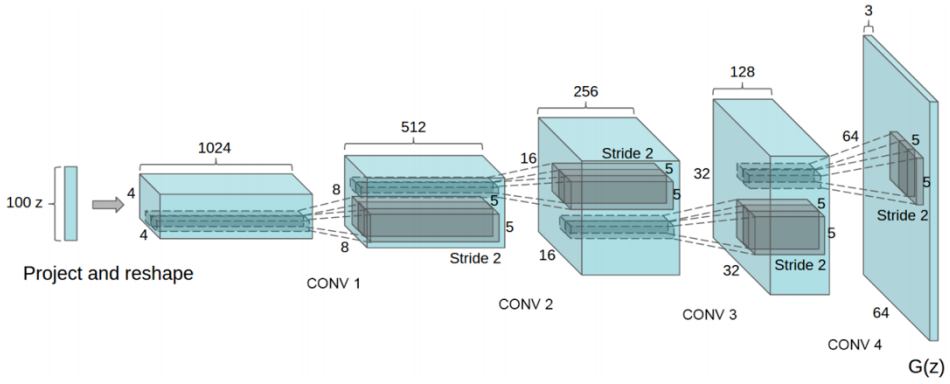


Figure 10.59 DCGAN architecture [12]

Transposed convolution implementation

In TensorFlow, the transposed convolution operation can be realized through nn.conv2d_transpose() function. We first complete the ordinary convolution operation through nn.conv2d. Note that the definition format of the convolution kernel of transposed convolution is $[k, k, c_{out}, c_{in}]$. E.g:

```
In [14]:
# Input 4x4
x = tf.range(16)+1
x = tf.reshape(x,[1,4,4,1])
x = tf.cast(x, tf.float32)
# 3x3 kernel
w = tf.constant([[-1,2,-3.],[4,-5,6],[-7,8,-9]])
w = tf.expand_dims(w,axis=2)
w = tf.expand_dims(w,axis=3)
# Regular convolutional operation
out = tf.nn.conv2d(x,w,strides=1,padding='VALID')
Out[14]:
<tf.Tensor: id=42, shape=(2, 2), dtype=float32, numpy=
array([[-56., -61.],
       [-76., -81.]], dtype=float32)>
```

With strides=1, padding='VALID', and the convolution kernel unchanged, we try to restore the height and width tensor of the same size as the input x through the transposed convolution operation of the convolution kernel w and the output. The code is as follows:

```
In [15]: # Restore 4x4 input
xx = tf.nn.conv2d_transpose(out, w, strides=1, padding='VALID',
output_shape=[1,4,4,1])
tf.squeeze(xx)
Out[15]:
<tf.Tensor: id=44, shape=(4, 4), dtype=float32, numpy=
array([[  56.,  -51.,   46.,  183.],
       [-148.,  -35.,   35., -123.],
       [  88.,   35.,  -35.,   63.],
       [ 532.,  -41.,   36.,  729.]], dtype=float32)>
```

It can be seen that the 4×4 feature map is generated by the transposed convolution, but the data of the feature map is not the same as the input x.

When using tf.nn.conv2d_transpose for transposed convolution operation, you need to manually set the output height and width. tf.nn.conv2d_transpose does not support customized padding settings, it can only be set to VALID or SAME.

When padding='VALID' is set, the output size is:

$$o = (i-1)s + k$$

When padding='SAME' is set, the output size is:

$$o = i \cdot s$$

If the reader is temporarily unable to understand the principle details of transposed convolution, he/she can keep the above two expressions in mind. For example, when calculate the 2×2 transposed convolution input and the 3×3 convolution kernel, strides=1, padding='VALID', the output size is:

$$h' = w' = (2-1)\cdot 1 + 3 = 4$$

When calculating 2×2 transposed convolution input and the 3×3 convolution kernel，strides=3, padding='SAME', the output size is：

$$h' = w' = 2 \cdot 3 = 6$$

Transposed convolution can also be the same as other layers. Create a transposed convolution layer through the layers.Conv2DTranspose class, and then call the instance to complete the forward calculation:

```
In [16]:
layer = layers.Conv2DTranspose(1,kernel_size=3,strides=1,padding='VALID')
xx2 = layer(out)
xx2
Out[16]:
<tf.Tensor: id=130, shape=(1, 4, 4, 1), dtype=float32, numpy=
array([[[[  9.7032385 ],
         [  5.485071  ],
         [ -1.6490463 ],
```

```
[  1.6279562 ]],...
```

10.11.3 Separate convolution

Here we take Depth-wise Separable Convolution as an example. When the ordinary convolution is operating on multi-channel input, each channel of the convolution kernel and each channel of the input are respectively convolved to obtain a multi-channel feature map, and then the corresponding elements are added to produce the final result of a single convolution kernel output as shown in Figure 10.60.

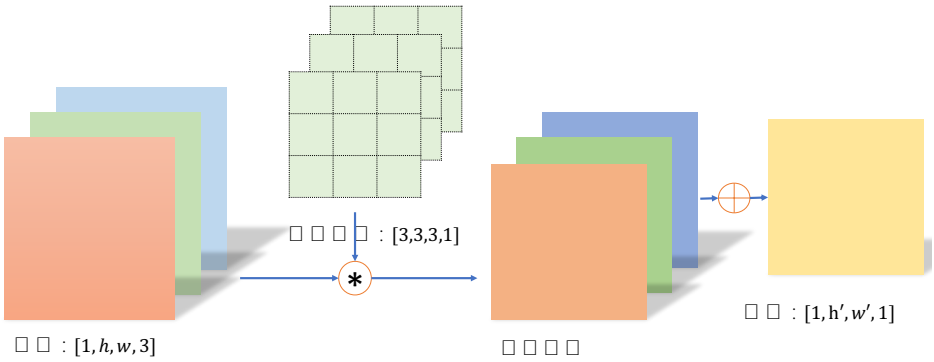


Figure 10.60 Schematic diagram of ordinary convolution calculation

The calculation process of separate convolution is different. Each channel of the convolution kernel is convolved with each input channel to obtain the intermediate features of multiple channels, as shown in Figure 10.61. This multi-channel intermediate feature tensor is then subjected to the ordinary convolution operation of multiple 1×1 convolution kernels to obtain multiple outputs with constant height and width. These outputs are spliced on the channel axis to produce the final separated convolutional layer output. It can be seen that the separated convolution layer includes a two-step convolution operation. The first convolution operation is a single convolution kernel, and the second convolution operation includes multiple convolution kernels.

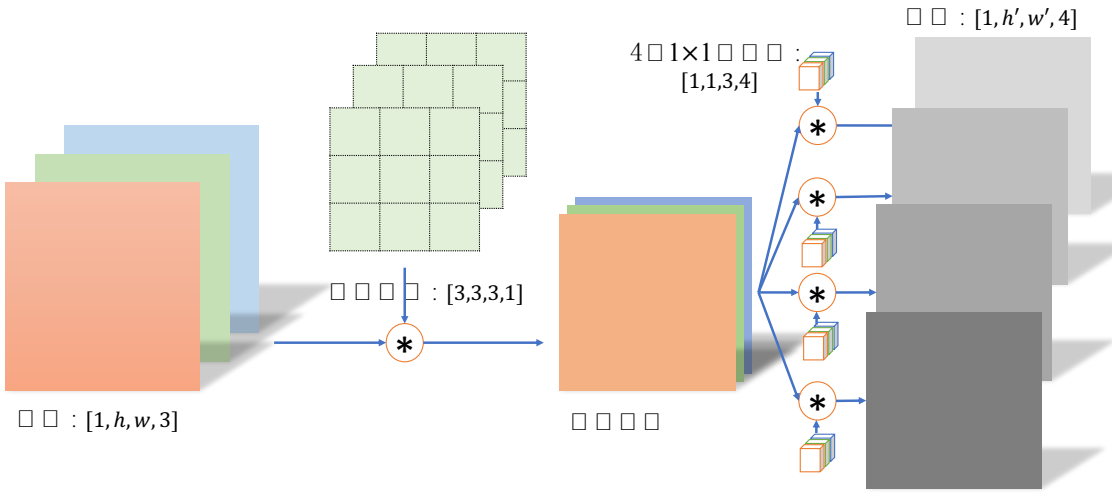


Figure 10.61 Schematic diagram of depth separable convolution calculation

So what are the advantages of using separate convolution? An obvious advantage is that for the same input and output, the parameters of the Separable Convolution are about 1/3 of the ordinary convolution. Consider the example of ordinary convolution and separate convolution in the figure above. The parameter quantity of ordinary convolution is

$$3 \cdot 3 \cdot 3 \cdot 4 = 108$$

The first part of the parameter of the separated convolution is

$$3 \cdot 3 \cdot 3 \cdot 1 = 27$$

The second part of the parameter is

$$1 \cdot 1 \cdot 3 \cdot 4 = 14$$

The total parameter amount of the separated convolution is only 39, but it can realize the same input and output size transformation of the ordinary convolution. Separate convolution has been widely used in areas sensitive to computational cost, such as Xception and MobileNets.

10.12 Deep Residual Network

The emergence of network models such as AlexNet, VGG, and GoogLeNet has brought the development of neural networks to a stage of dozens of layers. Researchers have found that the deeper the network, the more likely it is to obtain better generalization capabilities. But as the model deepens, the network becomes more and more difficult to train, which is mainly caused by gradient dispersion and gradient explosion. In a neural network with a deeper number of layers, when the gradient information is transmitted from the last layer of the network to the first layer of the network layer by layer, there will be a phenomenon that the gradient is close to 0 or the gradient value is very large during the transfer process. The deeper the network layer, the more serious this phenomenon may be.

So how to solve the gradient dispersion and gradient explosion phenomenon of deep neural networks? A very natural idea is that since shallow neural networks are not prone to these

gradients, you can try to add a fallback mechanism to the deep neural networks. When the deep neural network can easily fall back to the shallow neural network, the deep neural network can obtain model performance equivalent to that of the shallow neural network, but not worse.

By adding a direct connection between the input and output - Skip Connection, the neural network has the ability to fall back. Taking the VGG13 deep neural network as an example, assuming that the gradient dispersion phenomenon is observed in the VGG13 model, and the 10-layer network model does not observe the gradient dispersion phenomenon, then you can consider adding Skip Connection to the last two convolutional layers, as shown in Figure 10.62. In this way, the network model can automatically choose whether to complete the feature transformation through these two convolutional layers, or skip these two convolutional layers and choose Skip Connection, or combine the output of the two convolutional layers and Skip Connection .

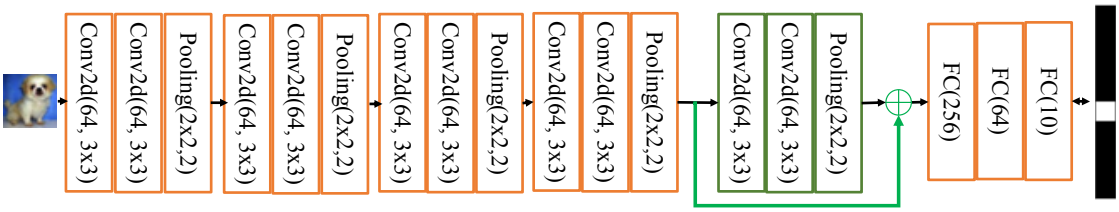


Figure 10.62 Architecture of VGG13 with Skip Connection

In 2015, He Kaiming and others from Microsoft Research Asia published a Skip Connection-based deep residual network (Residual Neural Network, referred to as ResNet) algorithm [10], and proposed 18 layers, 34 layers, 50 layers, 101 layers, and 152 layers network, i.e. ResNet-18, ResNet-34, ResNet-50, ResNet-101 and ResNet-152 models, and even successfully trained a very deep neural network with 1202 layers. ResNet has achieved the best performance on tasks such as classification and detection on the ImageNet dataset of the ILSVRC 2015 Challenge. The ResNet papers have so far received more than 25,000 citations, which shows the influence of ResNet in the artificial intelligence community.

10.12.1 ResNet Principle

ResNet implements the falllback mechanism by adding Skip Connection between the input and output of the convolutional layers, as shown in Figure 10.63. The input $\boldsymbol{x}$ passes through two convolutional layers to obtain the output $\mathcal{F}(\boldsymbol{x})$ after feature transformation, and the corresponding element of $\mathcal{F}(\boldsymbol{x})$ is added to $\boldsymbol{x}$ to get the final output:

$$\mathcal{H}(\boldsymbol{x}) = \boldsymbol{x} + \mathcal{F}(\boldsymbol{x})$$

$\mathcal{H}(\boldsymbol{x})$ is called Residual Block (ResBlock for short). Since the convolutional neural network surrounded by Skip Connection needs to learn the mapping $\mathcal{F}(\boldsymbol{x}) = \mathcal{H}(\boldsymbol{x}) - \boldsymbol{x}$, it is called the residual network.

In order to satisfy the addition of the input $\boldsymbol{x}$ and the output $\mathcal{F}(\boldsymbol{x})$ of the convolutional layer, the input shape needs to be exactly the same as the shape of the output $\mathcal{F}(\boldsymbol{x})$. When the shapes are inconsistent, the input $\boldsymbol{x}$ is generally transformed to the same shape of $\mathcal{F}(\boldsymbol{x})$ by adding additional convolution operations on Skip Connection, as shown in the function

identity($\boldsymbol{x}$) in Figure 10.63, where identity($\boldsymbol{x}$) mainly takes the 1×1 convolutional operation to adjust the input number of channels.

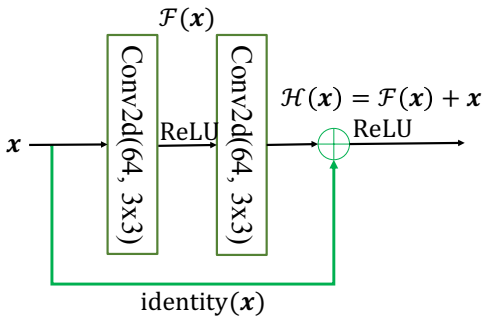


Figure 10.63 Residual module

Figure 10.64 below compares the 34-layer deep residual network, the 34-layer ordinary deep network and the 19-layer VGG network structure. It can be seen that the deep residual network reaches a deeper network layer by stacking residual modules, thereby obtaining a deep network model with stable training and superior performance.

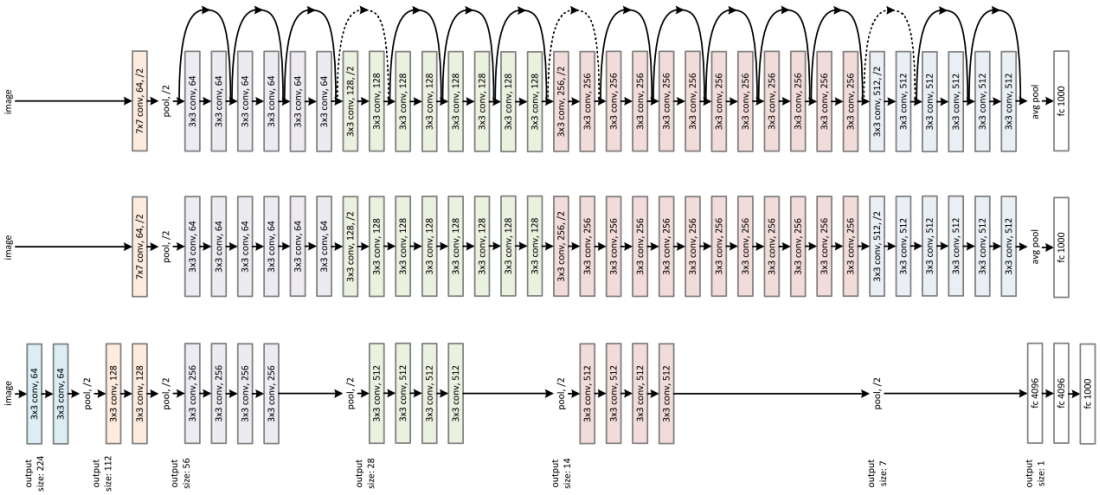


Figure 10.64 Network architecture comparison [10]

10.12.2 ResBlock implementation

The deep residual network does not add a new network layer type, but only adds a Skip Connection between the input and the output, so there is no underlying implementation for ResNet. The residual module can be implemented in TensorFlow by calling the ordinary convolutional layer.

First create a new class. Initialize the convolutional layer and activation function layer needed in the residual block, and then create a new convolutional layer, the code is as follows:

```
class BasicBlock(layers.Layer):
    # Residual block
    def __init__(self, filter_num, stride=1):
```

```
        super(BasicBlock, self).__init__()
        # Create Convolutional Layer 1
        self.conv1 = layers.Conv2D(filter_num, (3, 3), strides=stride, paddi
ng='same')
        self.bn1 = layers.BatchNormalization()
        self.relu = layers.Activation('relu')
        # Create Convolutional Layer 2
        self.conv2 = layers.Conv2D(filter_num, (3, 3), strides=1, padding='s
ame')
        self.bn2 = layers.BatchNormalization()
```

When the shape of $\mathcal{F}(\boldsymbol{x})$ and $\boldsymbol{x}$ is different, it cannot be added directly. We need to create a new convolutional layer $\text{identity}(\boldsymbol{x})$ to complete the shape conversion of $\boldsymbol{x}$. Following the above code, the implementation is as follows:

```
        if stride != 1: # Insert identity layer
            self.downsample = Sequential()
            self.downsample.add(layers.Conv2D(filter_num, (1, 1), strides=st
ride))
        else: # connect directly
            self.downsample = lambda x:x
```

During forward propagation, you only need to add $\mathcal{F}(\boldsymbol{x})$ and $\text{identity}(\boldsymbol{x})$ and add the ReLU activation function. The forward calculation function code is as follows:

```
    def call(self, inputs, training=None):
        # Forward calculation
        out = self.conv1(inputs) # 1st Conv layer
        out = self.bn1(out)
        out = self.relu(out)
        out = self.conv2(out) # 2nd Conv layer
        out = self.bn2(out)
        #  identity() conversion
        identity = self.downsample(inputs)
        # f(x)+x
        output = layers.add([out, identity])
        # activation function
        output = tf.nn.relu(output)
        return output
```

10.13 DenseNet

The idea of Skip Connection has achieved great success on ResNet. Researchers have begun to try different Skip Connection schemes, among which DenseNet [11] is more popular. DenseNet aggregates the feature map information of all the previous layers with the output of the current

layer through Skip Connection. Unlike ResNet's corresponding position addition method, DenseNet uses splicing operations in the channel axis dimension to aggregate feature information.

As shown in Figure 10.65 below, the input $\boldsymbol{X}_0$ is passed through the convolutional layer H_1 and the output $\boldsymbol{X}_1$ is spliced with the channel axis to obtain the aggregated feature tensor, which is sent to the convolutional layer H_2 to obtain the output $\boldsymbol{X}_2$. Similarly, $\boldsymbol{X}_2$ is spliced with $\boldsymbol{X}_1$ and $\boldsymbol{X}_0$ and sent to the next layer. Repeat this way until the output of the last layer $\boldsymbol{X}_4$ and the feature information of all previous layers: $\{\boldsymbol{X}_i\}_{i=0,1,2,3}$ are aggregated to the final output of the module. Such a densely connected module based on Skip Connection is called Dense Block.

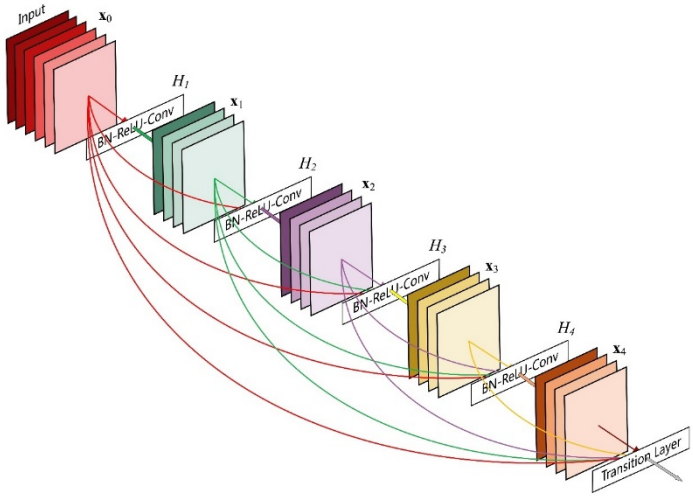


Figure 10.65 Dense Block architecture[②]

DenseNet constructs a complex deep neural network by stacking multiple Dense Blocks, as shown in Figure 10.66.

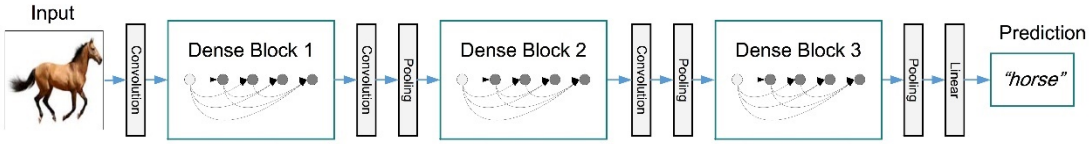


Figure 10.66 A typical DenseNet architecture[③]

Figure 10.67 compares the performance of different versions of DenseNet, the performance comparison of DenseNet and ResNet, and the training curves of DenseNet and ResNet.

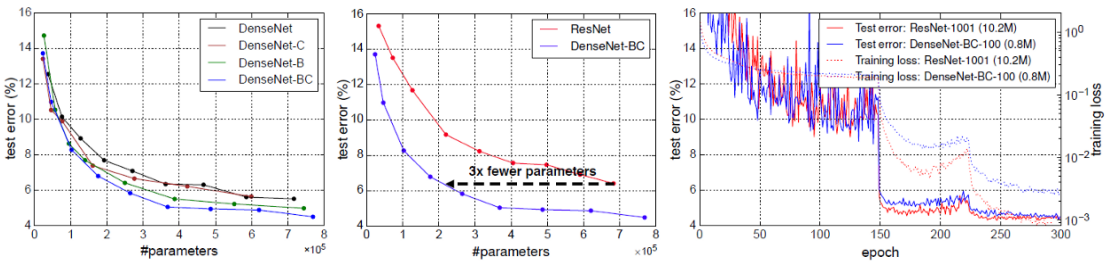


Figure 10.67 Comparison of DenseNet and ResNet performance [11]

---

② Image source: https://github.com/liuzhuang13/DenseNet

③ Image source: https://github.com/liuzhuang13/DenseNet

10.14 Hands-on CIFAR10 and ResNet18

In this section, we will implement the 18-layer deep residual network ResNet18, train and test it on the CIFAR10 image data set. We will compare its performance with the 13-layer ordinary neural network VGG13.

The standard ResNet18 accepts image data of size 224×224. We adjust ResNet18 appropriately so that its input size is 32×32 and its output dimension is 10. The adjusted ResNet18 network structure is shown in Figure 10.68.

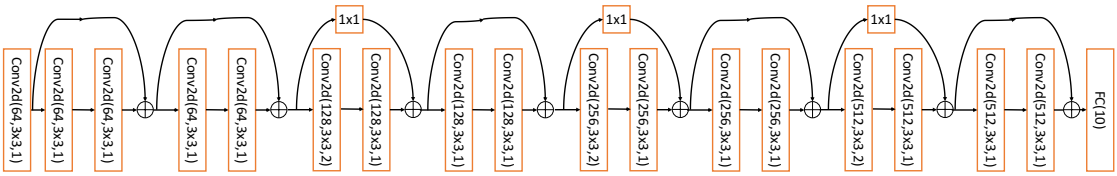


Figure 10.68 Adjusted ResNet18 architecture

First implement the residual module of the two convolutional layers in the middle, and residual block of Skip Connection 1x1 convolutional layer as below:

```
class BasicBlock(layers.Layer):
    # Residual block
    def __init__(self, filter_num, stride=1):
        super(BasicBlock, self).__init__()
        # 1st conv layer
        self.conv1 = layers.Conv2D(filter_num, (3, 3), strides=stride,
padding='same')
        self.bn1 = layers.BatchNormalization()
        self.relu = layers.Activation('relu')
        # 2nd conv layer
        self.conv2 = layers.Conv2D(filter_num, (3, 3), strides=1,
padding='same')
        self.bn2 = layers.BatchNormalization()

        if stride != 1:
            self.downsample = Sequential()
            self.downsample.add(layers.Conv2D(filter_num, (1, 1),
strides=stride))
        else:
            self.downsample = lambda x:x

    def call(self, inputs, training=None):
        # Forward calculation
```

```
        # [b, h, w, c], 1st conv layer
        out = self.conv1(inputs)
        out = self.bn1(out)
        out = self.relu(out)
        # 2nd conv layer
        out = self.conv2(out)
        out = self.bn2(out)
        # identity()
        identity = self.downsample(inputs)
        # Add two layers
        output = layers.add([out, identity])
        output = tf.nn.relu(output) # activation function

        return output
```

在设计深度卷积神经网络时，一般按照特征图高宽h/w逐渐减少，通道数c逐渐增大的经验法则。可以通过堆叠通道数逐渐增大的 Res Block 来实现高层特征的提取，通过 build_resblock 可以一次完成多个残差模块的新建。代码如下：

When designing a deep convolutional neural network, generally follow the rule of thumb that the height and width of the feature map gradually decrease and the number of channels gradually increases. The extraction of high-level features can be achieved by stacking Res Blocks with gradually increasing channel numbers, and multiple residual modules can be built at once through build_resblock as below:

```
    def build_resblock(self, filter_num, blocks, stride=1):
        # stack filter_num BasicBlocks
        res_blocks = Sequential()
        # Only 1st BasicBlock's stride may not be 1
        res_blocks.add(BasicBlock(filter_num, stride))

        for _ in range(1, blocks):# Stride of Other BasicBlocks are all 1
            res_blocks.add(BasicBlock(filter_num, stride=1))

        return res_blocks
```

Let's implement a general ResNet network model as below:

```
class ResNet(keras.Model):
    # General ResNet class
    def __init__(self, layer_dims, num_classes=10): # [2, 2, 2, 2]
        super(ResNet, self).__init__()
        self.stem = Sequential([layers.Conv2D(64, (3, 3), strides=(1, 1)),
                                layers.BatchNormalization(),
                                layers.Activation('relu'),
                                layers.MaxPool2D(pool_size=(2, 2), strides=(1, 1),
padding='same')
                                ])
```

```
        # Stack 4 Blocks
        self.layer1 = self.build_resblock(64,  layer_dims[0])
        self.layer2 = self.build_resblock(128, layer_dims[1], stride=2)
        self.layer3 = self.build_resblock(256, layer_dims[2], stride=2)
        self.layer4 = self.build_resblock(512, layer_dims[3], stride=2)

        # Pooling layer => 1x1
        self.avgpool = layers.GlobalAveragePooling2D()
        # Fully connected layer
        self.fc = layers.Dense(num_classes)

    def call(self, inputs, training=None):
        # Forward calculation
        x = self.stem(inputs)
        # 4 blocks
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        # Pooling layer
        x = self.avgpool(x)
        # Fully connected layer
        x = self.fc(x)

        return x
```

Different ResNets can be generated by adjusting the number of stacks and channels of each Res Block, such as with 64-64-128-128-256-256-512-512 channel configuration, a total of 8 Res Blocks, you can get ResNet18 network model. Each ResBlock contains 2 main convolutional layers, so the number of convolutional layers is $8 \cdot 2 = 16$, plus the fully connected layer at the end of the network, a total of 18 layers. Creating ResNet18 and ResNet34 can be simply implemented as follows:

```
def resnet18():
    return ResNet([2, 2, 2, 2])


def resnet34():
    return ResNet([3, 4, 6, 3])
```

Next, complete the loading of the CIFAR10 data set as follows:

```
(x,y), (x_test, y_test) = datasets.cifar10.load_data() # load data
y = tf.squeeze(y, axis=1) # sequeeze data
y_test = tf.squeeze(y_test, axis=1)
print(x.shape, y.shape, x_test.shape, y_test.shape)
```

```
train_db = tf.data.Dataset.from_tensor_slices((x,y)) # create training set
train_db = train_db.shuffle(1000).map(preprocess).batch(512)

test_db = tf.data.Dataset.from_tensor_slices((x_test,y_test)) #creat testing
set
test_db = test_db.map(preprocess).batch(512)
# sample an example
sample = next(iter(train_db))
print('sample:', sample[0].shape, sample[1].shape,
     tf.reduce_min(sample[0]), tf.reduce_max(sample[0]))
```

The data preprocessing logic is relatively simple. We just need directly mapp the data range to the interval $[-1,1]$. Here you can also perform standardization based on the mean and standard deviation of the ImageNet data pictures as below:

```
def preprocess(x, y):
    x = 2*tf.cast(x, dtype=tf.float32) / 255. - 1
    y = tf.cast(y, dtype=tf.int32)
    return x,y
```

The network training logic is the same as the normal classification network training part, and 50 Epochs are trained as below:

```
    for epoch in range(50): # Train epoch
        for step, (x,y) in enumerate(train_db):
            with tf.GradientTape() as tape:
                # [b, 32, 32, 3] => [b, 10], forward calculation
                logits = model(x)
                # [b] => [b, 10],one-hot encoding
                y_onehot = tf.one_hot(y, depth=10)
                # Calculate loss
                loss = tf.losses.categorical_crossentropy(y_onehot, logits,
from_logits=True)
                loss = tf.reduce_mean(loss)
            # Calculate gradient
            grads = tape.gradient(loss, model.trainable_variables)
            # Update parameters
            optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

ResNet18 has a total of 11 million network parameters. After 50 Epochs, the accuracy of the network reached 79.3%. Our code here is relatively streamlined. With the support of careful hyperparameters and data enhancement, the accuracy rate can be higher.

10.15 References

[1] G. E. Hinton, S. Osindero and Y.-W. Teh, “A Fast Learning Algorithm for Deep Belief Nets,” *Neural Comput.,* 18, pp. 1527-1554, 7 2006.

[2] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel, “Backpropagation Applied to Handwritten Zip Code Recognition,” *Neural Comput.,* 1, pp. 541-551, 12 1989.

[3] A. Krizhevsky, I. Sutskever and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou and K. Q. Weinberger, Curran Associates, Inc., 2012, pp. 1097-1105.

[4] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, “Gradient-based learning applied to document recognition,” *Proceedings of the IEEE*, 1998.

[5] M. D. Zeiler and R. Fergus, “Visualizing and Understanding Convolutional Networks, *Computer Vision -- ECCV 2014*, Cham, 2014.

[6] S. Ioffe and C. Szegedy, “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift,” *CoRR,* abs/1502.03167, 2015.

[7] Y. Wu and K. He, “Group Normalization,” *CoRR,* abs/1803.08494, 2018.

[8] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” *CoRR,* abs/1409.1556, 2014.

[9] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke and A. Rabinovich, “Going Deeper with Convolutions,” *Computer Vision and Pattern Recognition (CVPR)*, 2015.

[10] K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition,” *CoRR,* abs/1512.03385, 2015.

[11] G. Huang, Z. Liu and K. Q. Weinberger, “Densely Connected Convolutional Networks,” *CoRR,* abs/1608.06993, 2016.

[12] A. Radford, L. Metz and S. Chintala, *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks,* 2015.