Convolutional neural networks

Jianxin Wu

LAMDA Group
National Key Lab for Novel Software Technology
Nanjing University, China
wujx2001@gmail.com

May 14, 2020

Contents

This chapter describes how a Convolutional Neural Network (CNN) operates from a mathematical perspective. This chapter is self-contained, and the focus is to make it comprehensible for beginners to the CNN field.

The convolutional neural network (CNN) has shown excellent performance in many computer vision, machine learning, and pattern recognition problems. Many solid papers have been published on this topic, and quite a number of high quality open source CNN software packages have been made available.

There are also well-written CNN tutorials or CNN software manuals. However, we believe that introductory CNN material specifically prepared for beginners is still needed. Research papers are usually very terse and lack details. It might be difficult for beginners to read such papers. A tutorial targeting experienced researchers may not cover all the necessary details to understand how a CNN runs.

This chapter tries to present a document that

- is self-contained. It is expected that all required mathematical background knowledge is introduced in this chapter itself (or in other chapters in this book);

- has details for all the derivations. This chapter aims to explain all the necessary math in detail. We try not to ignore any important step in a derivation. Thus, it should be possible for a beginner to follow (although an expert may find this chapter a bit tautological);

- ignores implementation details. The purpose is for a reader to understand how a CNN runs at the mathematical level. We will ignore those implementation details. In CNN, making correct choices for various implementation details is one of the keys to its high accuracy (that is, "the devil is in the details"). However, we intentionally left this part out, in order for the reader to focus on the mathematics. After understanding the mathematical principles and details, it is more advantageous to learn these implementation and design details through hands-on experience by experimenting with CNN programming. The exercise problems in this chapter provide opportunities for hands-on CNN programming experiences.

CNNs are useful in a lot of applications, especially in image related tasks. Applications of CNNs include image classification, image semantic segmentation, object detection in images, etc. We will focus on image classification (or categorization) in this chapter. In image categorization, every image has a major object that occupies a large portion of the image. An image is classified into one of the classes based on the identity of its main object—e.g., dog, airplane, bird, etc.

1 Preliminaries

We start with a discussion of some background knowledge that is necessary in order to understand how a CNN runs. The reader can ignore this section if he/she is familiar with these basics.

1.1 Tensor and vectorization

Everybody is familiar with vectors and matrices. We use a symbol shown in boldface to represent a vector—e.g., $\boldsymbol{x} \in \mathbb{R}^D$ is a column vector with D elements. We use a capital letter to denote a matrix—e.g., $X \in \mathbb{R}^{H \times W}$ is a matrix with H rows and W columns. The vector $\boldsymbol{x}$ can also be viewed as a matrix with 1 column and D rows.

These concepts can be generalized to higher-order matrices—i.e., tensors. For example, $\boldsymbol{x} \in \mathbb{R}^{H \times W \times D}$ is an order 3 (or third order) tensor. It contains HWD elements, and each of them can be indexed by an index triplet (i, j, d), with $0 \le i < H$, $0 \le j < W$, and $0 \le d < D$. Another way to view an order 3 tensor is to treat it as containing D channels of matrices. Every channel is a matrix with size $H \times W$. The first channel contains all the numbers in the tensor that are indexed by $(i, j, 0)$. Note that in this chapter we assume the index starts from 0 rather than 1. When $D = 1$, an order 3 tensor reduces to a matrix.

We have interacted with tensors day-to-day. A scalar value is a zeroth-order (order 0) tensor; a vector is an order 1 tensor; and a matrix is a second order tensor. A color image is in fact an order 3 tensor. An image with H rows and W columns is a tensor with size $H \times W \times 3$: if a color image is stored in the RGB format, it has 3 channels (for R, G and B, respectively), and each channel is a $H \times W$ matrix (second order tensor) that contains the R (or G, or B) values of all pixels.

It is beneficial to represent images (or other types of raw data) as a tensor. In early computer vision and pattern recognition, a color image (which is an order 3 tensor) was often converted to the grayscale version (which is a matrix) because we know how to handle matrices much better than tensors. The color information is lost during this conversion. But color is very important in various image (or video) based learning and recognition problems, and we do want to process color information in a principled way—e.g., using a CNN.

Tensors are essential in CNN. The input, intermediate representation, and parameters in a CNN are all tensors. Tensors with order higher than 3 are also widely used in CNNs. For example, we will soon see that the convolution kernels in a convolution layer of a CNN form an order 4 tensor.

Given a tensor, we can arrange all the numbers inside it into a long vector, following a pre-specified order. For example, in Matlab/Octave, the `(:)` operator converts a matrix into a column vector in the column-first order. An example is:

$$A = \left[\begin{array}{cc} 1 & 2 \\ 3 & 4 \end{array} \right] , \quad A(:) = (1,3,2,4)^T = \left[\begin{array}{c} 1 \\ 3 \\ 2 \\ 4 \end{array} \right] . \tag{1}$$

In mathematics, we use the notation "vec" to represent this vectorization operator. That is, $\operatorname{vec}(A) = (1,3,2,4)^T$ in the example in Equation 1. In order to vectorize an order 3 tensor, we could vectorize its first channel (which is a matrix and we already know how to vectorize it), then the second channel, ..., till all channels are vectorized. The vectorization of the order 3 tensor is then the concatenation of the vectorization of all the channels in this order.

The vectorization of an order 3 tensor is a recursive process, which utilizes the vectorization of order 2 tensors. This recursive process can be applied to vectorize an order 4 (or even higher order) tensor in the same manner.

1.2 Vector calculus and the chain rule

The CNN learning process depends on vector calculus and the chain rule. Suppose z is a scalar (i.e., $z \in \mathbb{R}$) and $\boldsymbol{y} \in \mathbb{R}^H$ is a vector. If z is a function of $\boldsymbol{y}$, then the partial derivative of z with respect to $\boldsymbol{y}$ is a vector, defined as

$$\left[\frac{\partial z}{\partial \boldsymbol{y}} \right]_i = \frac{\partial z}{\partial y_i} . \tag{2}$$

In other words, $\frac{\partial z}{\partial \boldsymbol{y}}$ is a vector having *the same size* as $\boldsymbol{y}$, and its i-th element is $\frac{\partial z}{\partial y_i}$. Also note that

$$\frac{\partial z}{\partial \boldsymbol{y}^T} = \left(\frac{\partial z}{\partial \boldsymbol{y}} \right)^T .$$

Furthermore, suppose $\boldsymbol{x} \in \mathbb{R}^W$ is another vector, and $\boldsymbol{y}$ is a function of $\boldsymbol{x}$. Then, the partial derivative of $\boldsymbol{y}$ with respect to $\boldsymbol{x}$ is defined as

$$\left[\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}^T} \right]_{ij} = \frac{\partial y_i}{\partial x_j} . \tag{3}$$

This partial derivative is a $H \times W$ matrix, whose entry at the intersection of the i-th row and j-th column is $\frac{\partial y_i}{\partial x_j}$.

It is easy to see that z is a function of $\boldsymbol{x}$ in a chain-like argument: a function maps $\boldsymbol{x}$ to $\boldsymbol{y}$, and another function maps $\boldsymbol{y}$ to z. The chain rule can be used to compute $\frac{\partial z}{\partial \boldsymbol{x}^T}$, as

$$\frac{\partial z}{\partial \boldsymbol{x}^T} = \frac{\partial z}{\partial \boldsymbol{y}^T} \frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}^T} . \tag{4}$$

A sanity check for Equation 4 is to check the matrix/vector dimensions. Note that $\frac{\partial z}{\partial \boldsymbol{y}^T}$ is a row vector with H elements, or a $1 \times H$ matrix. (Be reminded that $\frac{\partial z}{\partial \boldsymbol{y}}$ is a column vector). Since $\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}^T}$ is an $H \times W$ matrix, the vector/matrix multiplication between them is valid, and the result should be a row vector with W elements, which matches the dimensionality of $\frac{\partial z}{\partial \boldsymbol{x}^T}$.

For specific rules to calculate partial derivatives of vectors and matrices, please refer to Chapter 2 and the Matrix Cookbook .

2 CNN overview

In this section, we will see how a CNN trains and predicts at the abstract level, with the details left for later sections.

2.1 The architecture

A CNN usually takes an order 3 tensor as its input—e.g., an image with H rows, W columns, and 3 channels (R, G, B color channels). Higher order tensor inputs, however, can be handled by CNN in a similar fashion. The input then sequentially goes through a number of processes. One processing step is usually called a layer, which could be a convolution layer, a pooling layer, a normalization layer, a fully connected layer, a loss layer, etc.

We will introduce the details of these layers later in this chapter. We will give detailed introductions to three types of layers: convolution, pooling, and ReLU, which are the key parts of almost all CNN models. Proper normalization—e.g., batch normalization—is important in the optimization process for learning good parameters in a CNN. Although it is not introduced in this chapter, we will present some related resources in the exercise problems.

For now, let us give an abstract description of the CNN structure first.

$$\boldsymbol{x}^1 \longrightarrow \boxed{\boldsymbol{w}^1} \longrightarrow \boldsymbol{x}^2 \longrightarrow \cdots \longrightarrow \boldsymbol{x}^{L-1} \longrightarrow \boxed{\boldsymbol{w}^{L-1}} \longrightarrow \boldsymbol{x}^L \longrightarrow \boxed{\boldsymbol{w}^L} \longrightarrow z \quad (5)$$

The above Equation 5 illustrates how a CNN runs layer by layer in a forward pass. The input is $\boldsymbol{x}^1$, usually an image (order 3 tensor). It goes through the processing in the first layer, which is the first box. We denote the parameters involved in the first layer's processing collectively as a tensor $\boldsymbol{w}^1$. The output of the first layer is $\boldsymbol{x}^2$, which also acts as the input to the second layer's processing. This processing proceeds till all layers in the CNN have been finished, which outputs $\boldsymbol{x}^L$.

One additional layer, however, is added for backward error propagation, a method that learns good parameter values in the CNN. Let's suppose the problem at hand is an image classification problem with C classes. A commonly used strategy is to output $\boldsymbol{x}^L$ as a C dimensional vector, whose i-th entry encodes the prediction (posterior probability of $\boldsymbol{x}^1$ coming from the i-th class). To make $\boldsymbol{x}^L$ a probability mass function, we can set the processing in the $(L-1)$-th layer as a softmax transformation of $\boldsymbol{x}^{L-1}$ (cf. Chapter 9). In other applications, the output $\boldsymbol{x}^L$ may have other forms and interpretations.

The last layer is a loss layer. Let us suppose $\boldsymbol{t}$ is the corresponding target (groundtruth) value for the input $\boldsymbol{x}^1$, then a cost or loss function can be used to measure the discrepancy between the CNN prediction $\boldsymbol{x}^L$ and the target $\boldsymbol{t}$. For example, a simple loss function could be

$$z = \frac{1}{2}\|\boldsymbol{t} - \boldsymbol{x}^L\|^2 \,, \tag{6}$$

although more complex loss functions are usually used. This squared ℓ_2 loss can be used in a regression problem.

In a classification problem, the cross entropy (cf. Chapter 10) loss is often used. The groundtruth in a classification problem is a categorical variable t. We first convert the categorical variable t to a C dimensional vector $\boldsymbol{t}$ (cf. Chapter 9). Now both $\boldsymbol{t}$ and $\boldsymbol{x}^L$ are probability mass functions, and the cross entropy loss measures the distance between them. Hence, we can minimize the cross entropy loss. Equation 5 explicitly models the loss function as a loss layer, whose processing is modeled as a box with parameters $\boldsymbol{w}^L$, although in many cases a loss layer does not involve any parameter—i.e., $\boldsymbol{w}^L = \emptyset$.

Note that there are other layers that do not have any parameter—that is, $\boldsymbol{w}^i$ may be empty for some $i < L$. The softmax layer is one such example. This layer can convert a vector into a probability mass function. The input to a softmax layer is a vector, whose values may be positive, zero, or negative. Suppose layer l is a softmax layer and its input is a vector $\boldsymbol{x}^l \in \mathbb{R}^d$. Then, its output is a vector $\boldsymbol{x}^{l+1} \in \mathbb{R}^d$, which is computed as

$$x_i^{l+1} = \frac{\exp(x_i^l)}{\sum_{j=1}^d \exp(x_j^l)} \,, \tag{7}$$

that is, a softmax transformed version of the input. After the softmax layer's processing, values in $\boldsymbol{x}^{l+1}$ form a probability mass function, and can be used as input to the cross entropy loss.

2.2 The forward run

Suppose all the parameters of a CNN model $\boldsymbol{w}^1, \boldsymbol{w}^2, \dots, \boldsymbol{w}^{L-1}$ have been learned; then we are ready to use this model for prediction. Prediction only involves running the CNN model forward—i.e., in the direction of the arrows in Equation 5.

Let's take the image classification problem as an example. Starting from the input $\boldsymbol{x}^1$, we make it pass the processing of the first layer (the box with parameters $\boldsymbol{w}^1$), and get $\boldsymbol{x}^2$. In turn, $\boldsymbol{x}^2$ is passed into the second layer, etc. Finally, we achieve $\boldsymbol{x}^L \in \mathbb{R}^C$, which estimates the posterior probabilities of $\boldsymbol{x}^1$ belonging to the C categories. We can output the CNN prediction as

$$\arg\max_i x_i^L \,. \tag{8}$$

Note that the loss layer is not needed in prediction. It is only useful when we try to learn CNN parameters using a set of training examples. Now, the problem is: how do we learn the model parameters?

2.3 Stochastic gradient descent (SGD)

As in many other learning systems, the parameters of a CNN model are optimized to minimize the loss z—i.e., we want the predictions of a CNN model to match the groundtruth labels on the training set.

Let's suppose one training example $\boldsymbol{x}^1$ is given for training such parameters. The training process involves running the CNN network in both directions. We first run the network in the forward direction to get $\boldsymbol{x}^L$ to achieve a prediction using the current CNN parameters. Instead of outputting this prediction, however, we need to compare it with the target $\boldsymbol{t}$ corresponding to $\boldsymbol{x}^1$—that is, continue running the forward pass till the last loss layer. Finally, we achieve a loss z.

The loss z is then a supervision signal, guiding how the parameters of the model should be modified (updated). And the Stochastic Gradient Descent (SGD) way of modifying the parameters is

$$\boldsymbol{w}^i \longleftarrow \boldsymbol{w}^i - \eta \frac{\partial z}{\partial \boldsymbol{w}^i} \,. \tag{9}$$

A cautious note about the notation. In most CNN materials, a superscript indicates the "time" (e.g., training epochs). But in this chapter, we use the superscript to denote the layer index. Please do not get confused. We do not use an additional index variable to represent time. In Equation 9, the $\longleftarrow$ sign implicitly indicates that the parameters $\boldsymbol{w}^i$ (of the i-layer) are updated from time t to $t+1$. If a time index t is explicitly used, this equation will look like

$$\left(\boldsymbol{w}^i\right)^{t+1} = \left(\boldsymbol{w}^i\right)^{t} - \eta \frac{\partial z}{\partial \left(\boldsymbol{w}^i\right)^{t}} \,. \tag{10}$$

In Equation 9, the partial derivative $\frac{\partial z}{\partial \boldsymbol{w}^i}$ measures the rate of increase of z with respect to the changes in different dimensions of $\boldsymbol{w}^i$. This partial derivative vector is called the *gradient* in mathematical optimization. Hence, in a small local region around the current value of $\boldsymbol{w}^i$, to move $\boldsymbol{w}^i$ in the direction determined by the gradient will increase the objective value z. In order to minimize the loss function, we should update $\boldsymbol{w}^i$ along the opposite direction of the gradient. This updating rule is called gradient descent. Gradient descent is illustrated in Figure 1, in which the gradient is denoted by $\boldsymbol{g}$.

If we move too far in the negative gradient direction, however, the loss function may increase. Hence, in every update we only change the parameters by a small proportion of the negative gradient, controlled by η, which is called the learning rate. $\eta > 0$ is usually set to a small number (e.g., $\eta = 0.001$) in deep neural network learning.

One update based on $\boldsymbol{x}^1$ will make the loss smaller for this particular training example if the learning rate is not too large. However, it is very possible that it will cause the loss of some other training examples to become larger. Hence, we need to update the parameters using all training examples. When all training

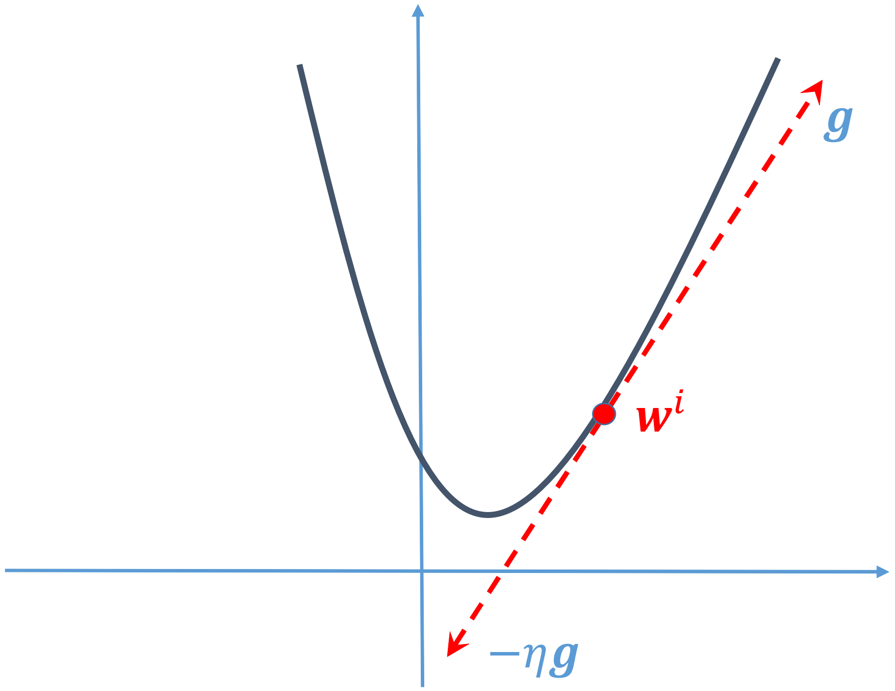


Figure 1: Illustration of the gradient descent method, in which η is the learning rate.

examples have been used to update the parameters, we say one *epoch* has been processed.

One epoch will in general reduce the average loss on the training set until the learning system overfits the training data. Hence, we can repeat the gradient descent updating for many epochs and terminate at some point to obtain the CNN parameters (e.g., we can terminate when the average loss on a validation set increases).

Gradient descent may seem simple in its math form (Equation 9), but it is a very tricky operation in practice. For example, if we update the parameters using the gradient calculated from only one training example, we will observe an unstable loss function: the average loss of all training examples will bounce up and down at very high frequency. This is because the gradient is estimated using only one training example instead of the entire training set—the gradient computed using only one example can be very unstable.

Contrary to single example based parameter updating, we can compute the gradient using all training examples and then update the parameters. However, this *batch* processing strategy requires a lot of computations because the parameters are updated only once in an epoch, and is hence impractical, especially when the number of training examples is large.

A compromise is to use a *mini-batch* of training examples, to compute the gradient using this mini-batch, and to update the parameters correspondingly. Updating the parameters using the gradient estimated from a (usually) small subset of training examples is called *stochastic* gradient descent. For example, we can set 32 or 64 examples as a mini-batch. Stochastic gradient descent (using the mini-batch strategy) is the mainstream method used to learn a CNN's parameters. We also want to note that when mini-batches are used, the input of the CNN becomes a 4th order tensor—e.g., $H \times W \times 3 \times 32$ if the mini-batch size is 32.

A new problem now becomes apparent: how to compute the gradient, which seems a very complex task?

2.4 Error back propagation

The last layer's partial derivatives are easy to compute. Because $\boldsymbol{x}^L$ is connected to z directly under the control of parameters $\boldsymbol{w}^L$, it is easy to compute $\frac{\partial z}{\partial \boldsymbol{w}^L}$. This step is only needed when $\boldsymbol{w}^L$ is not empty. In the same spirit, it is also easy to compute $\frac{\partial z}{\partial \boldsymbol{x}^L}$. For example, if the squared ℓ_2 loss is used, we have an empty $\frac{\partial z}{\partial \boldsymbol{w}^L}$, and

$$\frac{\partial z}{\partial \boldsymbol{x}^L} = \boldsymbol{x}^L - \boldsymbol{t}\,.$$

In fact, for every layer, we compute two sets of gradients: the partial derivatives of z with respect to the layer parameters $\boldsymbol{w}^i$, and that layer's input $\boldsymbol{x}^i$.

- The term $\frac{\partial z}{\partial \boldsymbol{w}^i}$, as seen in Equation 9, can be used to update the current (i-th) layer's parameters;

- The term $\frac{\partial z}{\partial \boldsymbol{x}^i}$ can be used to update parameters backwards—e.g., to the $(i-1)$-th layer. An intuitive explanation is: $\boldsymbol{x}^i$ is the output of the $(i-1)$-th layer and $\frac{\partial z}{\partial \boldsymbol{x}^i}$ is how $\boldsymbol{x}^i$ should be changed to reduce the loss function. Hence, we could view $\frac{\partial z}{\partial \boldsymbol{x}^i}$ as the part of the "error" supervision information propagated from z backward till the current layer, in a layer by layer fashion. Thus, we can continue the back propagation process, and use $\frac{\partial z}{\partial \boldsymbol{x}^i}$ to propagate the errors backward to the $(i-1)$-th layer.

This layer-by-layer backward updating procedure makes learning a CNN much easier. In fact, this strategy is also the standard practice for other types of neural networks beyond CNN; it is called error back propagation, or simply back propagation.

Let's take the i-th layer as an example. When we are updating the i-th layer, the back propagation process for the $(i+1)$-th layer must have been finished. That is, we already computed the terms $\frac{\partial z}{\partial \boldsymbol{w}^{i+1}}$ and $\frac{\partial z}{\partial \boldsymbol{x}^{i+1}}$. Both are stored in memory and ready for use.

Now our task is to compute $\frac{\partial z}{\partial \boldsymbol{w}^i}$ and $\frac{\partial z}{\partial \boldsymbol{x}^i}$. Using the chain rule, we have

$$\frac{\partial z}{\partial(\operatorname{vec}(\boldsymbol{w}^i)^T)} = \frac{\partial z}{\partial(\operatorname{vec}(\boldsymbol{x}^{i+1})^T)} \frac{\partial \operatorname{vec}(\boldsymbol{x}^{i+1})}{\partial(\operatorname{vec}(\boldsymbol{w}^i)^T)}\,, \tag{11}$$

$$\frac{\partial z}{\partial(\operatorname{vec}(\boldsymbol{x}^i)^T)} = \frac{\partial z}{\partial(\operatorname{vec}(\boldsymbol{x}^{i+1})^T)} \frac{\partial \operatorname{vec}(\boldsymbol{x}^{i+1})}{\partial(\operatorname{vec}(\boldsymbol{x}^i)^T)}\,. \tag{12}$$

Since $\frac{\partial z}{\partial \boldsymbol{x}^{i+1}}$ is already computed and stored in memory, it requires just a matrix reshaping operation (vec) and an additional transpose operation to get $\frac{\partial z}{\partial(\operatorname{vec}(\boldsymbol{x}^{i+1})^T)}$, which is the first term in the right hand side (RHS) of both equations. So long as we can compute $\frac{\partial \operatorname{vec}(\boldsymbol{x}^{i+1})}{\partial(\operatorname{vec}(\boldsymbol{w}^i)^T)}$ and $\frac{\partial \operatorname{vec}(\boldsymbol{x}^{i+1})}{\partial(\operatorname{vec}(\boldsymbol{x}^i)^T)}$, we can easily get what we want (the left hand side of both equations).

$\frac{\partial \operatorname{vec}(\boldsymbol{x}^{i+1})}{\partial(\operatorname{vec}(\boldsymbol{w}^i)^T)}$ and $\frac{\partial \operatorname{vec}(\boldsymbol{x}^{i+1})}{\partial(\operatorname{vec}(\boldsymbol{x}^i)^T)}$ are much easier to compute than directly computing $\frac{\partial z}{\partial(\operatorname{vec}(\boldsymbol{w}^i)^T)}$ and $\frac{\partial z}{\partial(\operatorname{vec}(\boldsymbol{x}^i)^T)}$, because $\boldsymbol{x}^i$ is directly related to $\boldsymbol{x}^{i+1}$, through a function with parameters $\boldsymbol{w}^i$. The details of these partial derivatives will be discussed in the following sections for different layers.

3 Layer input, output, and notations

Now that the CNN architecture is clear, we will discuss in detail the different types of layers, starting from the ReLU layer, which is the simplest layer among those we discuss in this chapter. But before we start, we need to further refine our notations.

Suppose we are considering the l-th layer, whose inputs form an order 3 tensor $\boldsymbol{x}^l$ with $\boldsymbol{x}^l \in \mathbb{R}^{H^l \times W^l \times D^l}$. Thus, we need a triplet index set (i^l, j^l, d^l) to locate any specific element in $\boldsymbol{x}^l$. The triplet (i^l, j^l, d^l) refers to one element in $\boldsymbol{x}^l$, which is in the d^l-th channel, and at spatial location (i^l, j^l) (at the i^l-th row, and j^l-th column). In actual CNN learning, the mini-batch strategy is usually used. In that case, $\boldsymbol{x}^l$ becomes an order 4 tensor in $\mathbb{R}^{H^l \times W^l \times D^l \times N}$ where N is the mini-batch size. For simplicity we assume that $N = 1$ in this chapter. The results in this chapter, however, are easy to adopt to mini-batch versions.

In order to simplify the notations that will appear later, we follow the zero-based indexing convention, which specifies that $0 \le i^l < H^l$, $0 \le j^l < W^l$, and $0 \le d^l < D^l$.

In the l-th layer, a function will transform the input $\boldsymbol{x}^l$ to an output $\boldsymbol{y}$, which is also the input to the next layer. Thus, we notice that $\boldsymbol{y}$ and $\boldsymbol{x}^{l+1}$ in fact refers to the same object, and it is very helpful to keep this point in mind. We assume the output has size $H^{l+1} \times W^{l+1} \times D^{l+1}$, and an element in the output is indexed by a triplet $(i^{l+1}, j^{l+1}, d^{l+1})$, $0 \le i^{l+1} < H^{l+1}$, $0 \le j^{l+1} < W^{l+1}$, $0 \le d^{l+1} < D^{l+1}$.

4 The ReLU layer

A ReLU layer does not change the size of the input—that is, $\boldsymbol{x}^l$ and $\boldsymbol{y}$ share the same size. In fact, the Rectified Linear Unit (hence the name ReLU) can be regarded as a truncation performed individually for every element in the input tensor:

$$y_{i,j,d} = \max\{0, x^l_{i,j,d}\}\,, \tag{13}$$

with $0 \le i < H^l = H^{l+1}$, $0 \le j < W^l = W^{l+1}$, and $0 \le d < D^l = D^{l+1}$.

There is no parameter inside a ReLU layer, hence no need for parameter learning in this layer.

Based on Equation 13, it is obvious that

$$\frac{\mathrm{d} y_{i,j,d}}{\mathrm{d} x^l_{i,j,d}} = \llbracket x^l_{i,j,d} > 0 \rrbracket\,, \tag{14}$$

where $\llbracket \cdot \rrbracket$ is the indicator function, being 1 if its argument is true, and 0 otherwise.

Hence, we have

$$\left[\frac{\partial z}{\partial \boldsymbol{x}^l}\right]_{i,j,d} = \begin{cases} \left[\frac{\partial z}{\partial \boldsymbol{y}}\right]_{i,j,d} & \text{if } \boldsymbol{x}^l_{i,j,d} > 0 \\ 0 & \text{otherwise} \end{cases}\,. \tag{15}$$

Note that $\boldsymbol{y}$ is an alias for $\boldsymbol{x}^{l+1}$.

Strictly speaking, the function $\max(0, x)$ is not differentiable at $x = 0$, hence Equation 14 is a little bit problematic in theory. In practice, it is not an issue and ReLU is safe to use.

The purpose of ReLU is to increase the nonlinearity of the CNN. Since the semantic information in an image (e.g., a person and a husky dog sitting next to each other on a bench in a garden) is obviously a highly nonlinear mapping of pixel values in the input, we want the mapping from CNN input to its output

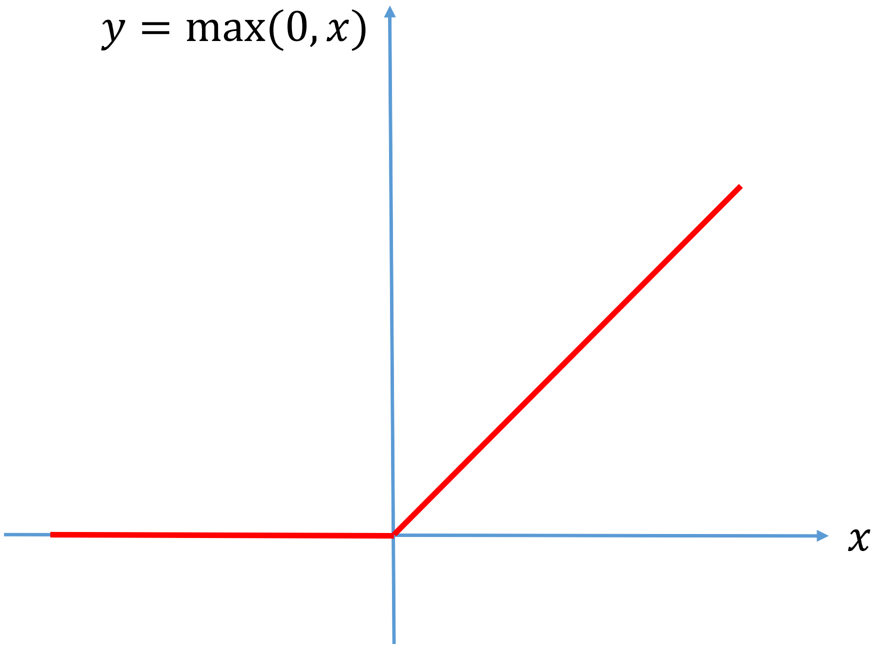


Figure 2: The ReLU function.

also to be highly nonlinear. The ReLU function, although simple, is a nonlinear function, as illustrated in Figure 2.

We may treat $x^l_{i,j,d}$ as one of the $H^l W^l D^l$ *features* extracted by CNN layers 1 to $l-1$, which can be positive, zero, or negative. For example, $x^l_{i,j,d}$ may be positive if a region inside the input image has certain patterns (like a dog's head or a cat's head or some other patterns similar to that); and $x^l_{i,j,d}$ is negative or zero when that region does not exhibit these patterns. The ReLU layer will set all negative values to 0, which means that $y^l_{i,j,d}$ will be *activated* only for images possessing these patterns in that particular region.

Intuitively, this property is useful for recognizing complex patterns and objects. For example, it is only weak evidence to support "the input image contains a cat" if a feature is activated and that feature's pattern looks like a cat's head. However, if we find many activated features after the ReLU layer whose target patterns correspond to cat's head, torso, fur, legs, etc., we have higher confidence (at layer $l+1$) to say that a cat probably exists in the input image.

Other nonlinear transformations have been used in the neural network community to produce nonlinearity—for example, the logistic sigmoid function

$$y = \sigma(x) = \frac{1}{1+\exp(-x)} \,.$$

However, the logistic sigmoid performs significantly worse than ReLU in CNN learning. Note that $0 < y < 1$ if a sigmoid function is used and

$$\frac{\mathrm{d}y}{\mathrm{d}x} = y(1-y)\,,$$

we have

$$0 < \frac{\mathrm{d}y}{\mathrm{d}x} \leq \frac{1}{4}\,.$$

Hence, in the error back propagation process, the gradient $\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}x}$ will have much smaller magnitude than $\frac{\partial z}{\partial y}$ (at most $\frac{1}{4}$). In other words, a sigmoid layer will cause the magnitude of the gradient to significantly reduce, and after several sigmoid layers, the gradient will *vanish* (i.e., all its components will be close to 0). A vanishing gradient makes gradient based learning (e.g., SGD) very difficult. Another major drawback of the sigmoid is that it is saturated. When the magnitude of x is large—e.g., when $x > 6$ or $x < -6$—the corresponding gradient is almost 0.

On the other hand, the ReLU layer sets the gradient of some features in the l-th layer to 0, but these features are not activated (i.e., we are not interested in them). For those activated features, the gradient is back propagated without any change, which is beneficial for SGD learning. The introduction of ReLU to replace sigmoid is an important change in CNN, which significantly reduces the difficulty in learning CNN parameters and improves its accuracy. There are also more complex variants of ReLU, for example, parametric ReLU and exponential linear unit, which we do not touch on in this chapter.

5 The convolution layer

Next, we turn to the convolution layer, which is the most involved one among those we discuss in this chapter. It is also the most important layer in a CNN, hence the name convolutional neural networks.

5.1 What is a convolution?

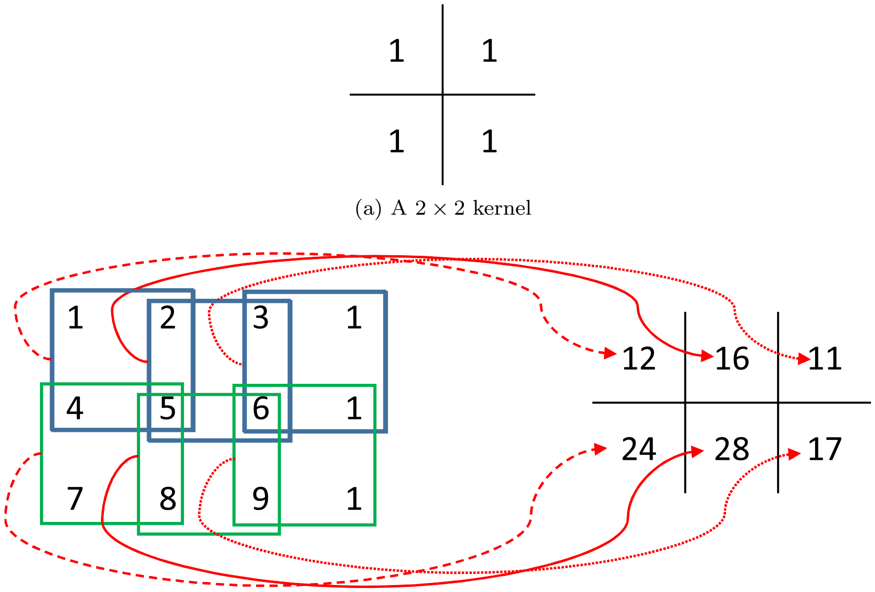
Let us start by convolving a matrix with one single convolution kernel. Suppose the input image is 3×4 and the convolution kernel size is 2×2, as illustrated in Figure 3.

If we overlay the convolution kernel on top of the input image, we can compute the product between the numbers at the same location in the kernel and the input, and we get a single number by summing these products together. For example, if we overlay the kernel with the top left region in the input, the convolution result at that spatial location is

$$1 \times 1 + 1 \times 4 + 1 \times 2 + 1 \times 5 = 12\,.$$

We then move the kernel down by one pixel and get the next convolution result as

$$1 \times 4 + 1 \times 7 + 1 \times 5 + 1 \times 8 = 24\,.$$



(a) A 2×2 kernel

(b) The convolution input and output

Figure 3: Illustration of the convolution operation. ()

We keep moving the kernel down till it reaches the bottom border of the input matrix (image). Then, we return the kernel to the top, and move the kernel to its right by one element (pixel). We repeat the convolution for every possible pixel location until we have moved the kernel to the bottom right corner of the input image, as shown in Figure 3.

For order 3 tensors, the convolution operation is defined similarly. Suppose the input in the l-th layer is an order 3 tensor with size $H^l \times W^l \times D^l$. One convolution kernel is also an order 3 tensor with size $H \times W \times D^l$. When we overlay the kernel on top of the input tensor at the spatial location $(0, 0, 0)$, we compute the products of corresponding elements in all the D^l channels and sum the HWD^l products to get the convolution result at this spatial location. Then, we move the kernel from top to bottom and from left to right to complete the convolution.

In a convolution layer, multiple convolution kernels are usually used. Assuming D kernels are used and each kernel is of spatial span $H \times W$, we denote all the kernels as $\boldsymbol{f}$. $\boldsymbol{f}$ is an order 4 tensor in $\mathbb{R}^{H \times W \times D^l \times D}$. Similarly, we use index variables $0 \le i < H$, $0 \le j < W$, $0 \le d^l < D^l$ and $0 \le d < D$ to pinpoint a specific element in the kernels. Also note that the set of kernels $\boldsymbol{f}$ refer to the same object as the notation $\boldsymbol{w}^l$ in Equation 5. We change the notation slightly to make the derivation a little simpler. It is also clear that even if the mini-batch strategy is used, the kernels remain unchanged.

As shown in Figure 3, the spatial extent of the output is smaller than that of the input so long as the convolution kernel is larger than 1×1. Sometimes we need the input and output images to have the same height and width, and a simple padding trick can be used. If the input is $H^l \times W^l \times D^l$ and the kernel size is $H \times W \times D^l \times D$, the convolution result has size

$$(H^l - H + 1) \times (W^l - W + 1) \times D\,.$$

For every channel of the input, if we *pad* (i.e., insert) $\lfloor \frac{H-1}{2} \rfloor$ rows above the first row and $\lfloor \frac{H}{2} \rfloor$ rows below the last row, and pad $\lfloor \frac{W-1}{2} \rfloor$ columns to the left of the first column and $\lfloor \frac{W}{2} \rfloor$ columns to the right of the last column of the input, the convolution output will be $H^l \times W^l \times D$ in size—i.e., having the same spatial extent as the input ($\lfloor \cdot \rfloor$ is the floor function). Elements of the padded rows and columns are usually set to 0, but other values are also possible.

Stride is another important concept in convolution. In Figure 3, we convolve the kernel with the input at every possible spatial location, which corresponds to the stride $s = 1$. However, if $s > 1$, every movement of the kernel skips $s - 1$ pixel locations (i.e., the convolution is performed once every s pixels both horizontally and vertically). When $s > 1$, a convolution's output will be much smaller than that of the input—H^{l+1} and W^{l+1} will be roughly $1/s$ of H^l and W^l, respectively.

In this section, we consider the simple case when the stride is 1 and no padding is used. Hence, we have $\boldsymbol{y}$ (or $\boldsymbol{x}^{l+1}$) in $\mathbb{R}^{H^{l+1} \times W^{l+1} \times D^{l+1}}$, with $H^{l+1} = H^l - H + 1$, $W^{l+1} = W^l - W + 1$, and $D^{l+1} = D$.

In precise mathematics, the convolution procedure can be expressed as an equation:

$$y_{i^{l+1}, j^{l+1}, d} = \sum_{i=0}^{H-1} \sum_{j=0}^{W-1} \sum_{d^l=0}^{D^l-1} f_{i,j,d^l,d} \times x^l_{i^{l+1}+i, j^{l+1}+j, d^l}\,. \tag{16}$$

Equation 16 is repeated for all $0 \le d < D = D^{l+1}$, and for any spatial location (i^{l+1}, j^{l+1}) satisfying

$$0 \le i^{l+1} < H^l - H + 1 = H^{l+1}\,, \tag{17}$$

$$0 \le j^{l+1} < W^l - W + 1 = W^{l+1}\,. \tag{18}$$

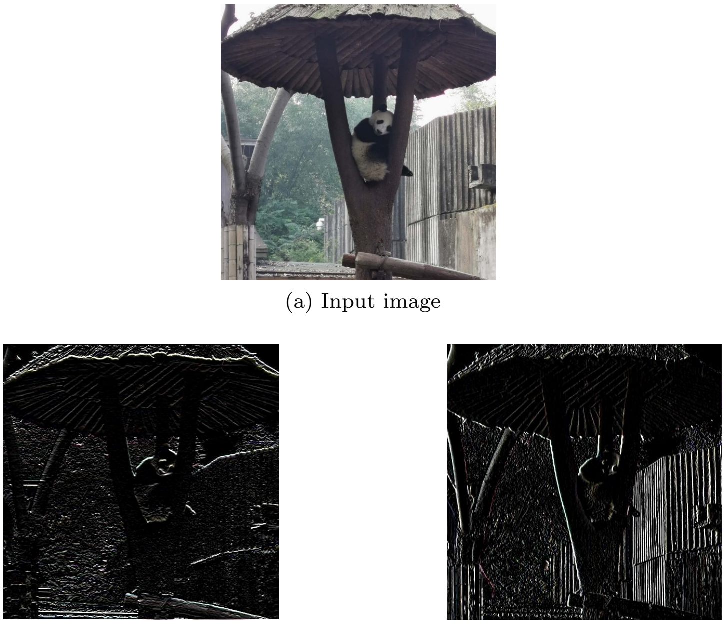
In this equation, $x^l_{i^{l+1}+i, j^{l+1}+j, d^l}$ refers to the element of $\boldsymbol{x}^l$ indexed by the triplet $(i^{l+1}+i, j^{l+1}+j, d^l)$.

A bias term b_d is usually added to $y_{i^{l+1}, j^{l+1}, d}$. We omit this term in this chapter for clearer presentation.

5.2 Why to convolve?

Figure 4 shows a color input image (4a) and its convolution results using two different kernels (4b and 4c). A 3×3 convolution matrix

$$K = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$



(a) Input image

(b) Horizontal edge

(c) Vertical edge

Figure 4: An image and the effect of different convolution kernels. ()

is used. The convolution kernel should be of size $3 \times 3 \times 3$, in which we set every channel to K. When there is a horizontal edge at location (x, y) (i.e., when the pixels at spatial location $(x+1, y)$ and $(x-1, y)$ differ by a large amount), we expect the convolution result to have high magnitude. As shown in Figure 4b, the convolution results indeed highlight the horizontal edges. When we set every channel of the convolution kernel to K^T (the transpose of K), the convolution result amplifies vertical edges, as shown in Figure 4c. The matrix (or filter) K and K^T are called Sobel operators.[1]

If we add a bias term to the convolution operation, we can make the convolution result positive at horizontal (vertical) edges in a certain direction (e.g., a horizontal edge with the pixels above it brighter than the pixels below it), and negative at other locations. If the next layer is a ReLU layer, the output of the next layer in fact defines many "edge detection features," which activate only at horizontal or vertical edges in certain directions. If we replace the Sobel kernel by other kernels (e.g., those learned by SGD), we can learn features that activate for edges with different angles.

When we move further down in the deep network, subsequent layers can learn to activate only for specific (but more complex) patterns—e.g., groups of edges that form a particular shape. This is because any feature in layer $l + 1$

[1] The Sobel operator is named after Irwin Sobel, an American researcher in digital image processing.

considers the combined effect of many features in layer l. These more complex patterns will be further assembled by deeper layers to activate for semantically meaningful object parts or even a particular type of object—e.g., dog, cat, tree, beach, etc.

One more benefit of the convolution layer is that all spatial locations share the same convolution kernel, which greatly reduces the number of parameters needed for a convolution layer. For example, if multiple dogs appear in an input image, the same "dog-head-like pattern" feature might be activated at multiple locations, corresponding to heads of different dogs.

In a deep neural network setup, convolution also encourages parameter sharing. For example, suppose "dog-head-like pattern" and "cat-head-like pattern" are two features learned by a deep convolutional network. The CNN does not need to devote two sets of disjoint parameters (e.g., convolution kernels in multiple layers) to them. The CNN's bottom layers can learn "eye-like pattern" and "animal-fur-texture pattern," which are shared by both these more abstract features. In short, the combination of convolution kernels and deep and hierarchical structures is very effective in learning good representations (features) from images for visual recognition tasks.

We want to add a note here. Although we have used phrases such as "dog-head-like pattern," the representation or feature learned by a CNN may not correspond exactly to semantic concepts such as "dog's head." A CNN feature may activate frequently for dogs' heads and often be deactivated for other types of patterns. However, there are also possible false activations at other locations, and possible deactivations at dogs' heads.

In fact, a key concept in CNN (or more generally deep learning) is *distributed representation*. For example, suppose our task is to recognize N different types of objects, and a CNN extracts M features from any input image. It is most likely that any one of the M features is useful for recognizing all N object categories, and to recognize one object type requires the joint effort of all M features.

5.3 Convolution as matrix product

Equation 16 seems pretty complex. However, there is a way to expand $\boldsymbol{x}^l$ and simplify the convolution as a matrix product.

Let's consider a special case with $D^l = D = 1$, $H = W = 2$, and $H^l = 3$, $W^l = 4$. That is, we consider convolving a small single channel 3×4 matrix (or image) with one 2×2 filter. Using the example in Figure 3, we have

$$\begin{bmatrix} 1 & 2 & 3 & 1 \\ 4 & 5 & 6 & 1 \\ 7 & 8 & 9 & 1 \end{bmatrix} * \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 12 & 16 & 11 \\ 24 & 28 & 17 \end{bmatrix}, \tag{19}$$

where the first matrix is denoted as A, and $*$ is the convolution operator.

Now let's run a Matlab/Octave command `B=im2col(A,[2 2])`; we arrive at a B matrix that is an expanded version of A:

$$B = \begin{bmatrix} 1 & 4 & 2 & 5 & 3 & 6 \\ 4 & 7 & 5 & 8 & 6 & 9 \\ 2 & 5 & 3 & 6 & 1 & 1 \\ 5 & 8 & 6 & 9 & 1 & 1 \end{bmatrix} .$$

It is obvious that the first column of B corresponds to the first 2×2 region in A, in a column-first order, corresponding to $(i^{l+1}, j^{l+1}) = (0,0)$. Similarly, the second to last column in B correspond to regions in A with (i^{l+1}, j^{l+1}) being $(1,0)$, $(0,1)$, $(1,1)$, $(0,2)$ and $(1,2)$, respectively. That is, the Matlab/Octave `im2col` function explicitly expands the required elements for performing each individual convolution into a column in the matrix B. The transpose of B, B^T, is called the `im2row` expansion of A. Note that the parameter `[2 2]` specifies the convolution kernel size.

Now, if we vectorize the convolution kernel itself into a vector (in the same column-first order) $(1,1,1,1)^T$, we find that[2]

$$B^T \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} = \begin{bmatrix} 12 \\ 24 \\ 16 \\ 28 \\ 11 \\ 17 \end{bmatrix} . \tag{20}$$

If we reshape this resulting vector in Equation 20 properly, we get the exact convolution result matrix in Equation 19.

That is, the convolution operator is a linear operator. We can multiply the expanded input matrix and the vectorized filter to get a result vector, and by reshaping this vector properly we get the correct convolution result.

We can generalize this idea to more complex situations and formalize them. If $D^l > 1$ (that is, the input $\boldsymbol{x}^l$ has more than one channels), the expansion operator could first expand the first channel of $\boldsymbol{x}^l$, then the second, …, till all D^l channels are expanded. The expanded channels will be stacked together, that is, one row in the `im2row` expansion will have $H \times W \times D^l$ elements, rather than $H \times W$.

More formally, suppose $\boldsymbol{x}^l$ is a third order tensor in $\mathbb{R}^{H^l \times W^l \times D^l}$, with one element in $\boldsymbol{x}^l$ being indexed by a triplet (i^l, j^l, d^l). We also consider a set of convolution kernels $\boldsymbol{f}$, whose spatial sizes are all $H \times W$. Then, the expansion operator (`im2row`) converts $\boldsymbol{x}^l$ into a matrix $\phi(\boldsymbol{x}^l)$. We use two indexes (p, q)

[2] The notation and presentation of this chapter is heavily affected by the MatConvNet software package's manual (http://arxiv.org/abs/1412.4564, which is Matlab based). The transpose of an `im2col` expansion is equivalent to an `im2row` expansion, in which the numbers involved in one convolution form one row in the `im2row` expanded matrix. The derivation in this section uses `im2row`, complying with the implementation in MatConvNet. Caffe, a widely used CNN software package (http://caffe.berkeleyvision.org/, which is C++ based) uses `im2col`. These formulations are mathematically equivalent to each other.

to index an element in this matrix. The expansion operator copies the element at (i^l, j^l, d^l) in $\boldsymbol{x}^l$ to the (p, q)-th entry in $\phi(\boldsymbol{x}^l)$.

From the description of the expansion process, it is clear that given a fixed (p, q), we can calculate its corresponding (i^l, j^l, d^l) triplet, because obviously

$$p = \quad i^{l+1} + (H^l - H + 1) \times j^{l+1} \,, \tag{21}$$

$$q = \quad i + H \times j + H \times W \times d^l \,, \tag{22}$$

$$i^l = \quad i^{l+1} + i \,, \tag{23}$$

$$j^l = \quad j^{l+1} + j \,. \tag{24}$$

In Equation 22, dividing q by HW and taking the integer part of the quotient, we can determine which channel (d^l) it belongs to. Similarly, we can get the offsets inside the convolution kernel as (i, j), in which $0 \le i < H$ and $0 \le j < W$. q completely determines one specific location inside the convolution kernel by the triplet (i, j, d^l).

Note that the convolution result is $\boldsymbol{x}^{l+1}$, whose spatial extent is $H^{l+1} = H^l - H + 1$ and $W^{l+1} = W^l - W + 1$. Thus, in Equation 21, the remainder and quotient of dividing p by $H^{l+1} = H^l - H + 1$ will give us the offset in the convolved result (i^{l+1}, j^{l+1}), or, the top-left spatial location of the region in $\boldsymbol{x}^l$ (which is to be convolved with the kernel).

Based on the definition of convolution, it is clear that we can use Equations 23 and 24 to find the offset in the input $\boldsymbol{x}^l$ as $i^l = i^{l+1} + i$ and $j^l = j^{l+1} + j$. That is, the mapping from (p, q) to (i^l, j^l, d^l) is one-to-one. However, we want to emphasize that the reverse mapping from (i^l, j^l, d^l) to (p, q) is one-to-many, a fact that is useful in deriving the back propagation rules in a convolution layer.

Now we use the standard vec operator to convert the set of convolution kernels $\boldsymbol{f}$ (order 4 tensor) into a matrix. Let's start from one kernel, which can be vectorized into a vector in $\mathbb{R}^{HWD^l}$. Thus, all convolution kernels can be reshaped into a matrix with HWD^l rows and D columns (remember that $D^{l+1} = D$.) Let's call this matrix F.

Finally, with all these notations, we have a beautiful equation to calculate convolution results (cf. Equation 20, in which $\phi(\boldsymbol{x}^l)$ is B^T):

$$\operatorname{vec}(\boldsymbol{y}) = \operatorname{vec}(\boldsymbol{x}^{l+1}) = \operatorname{vec}\left(\phi(\boldsymbol{x}^l) F\right) \,. \tag{25}$$

Note that $\operatorname{vec}(\boldsymbol{y}) \in \mathbb{R}^{H^{l+1}W^{l+1}D}$, $\phi(\boldsymbol{x}^l) \in \mathbb{R}^{(H^{l+1}W^{l+1}) \times (HWD^l)}$, and $F \in \mathbb{R}^{(HWD^l) \times D}$. The matrix multiplication $\phi(\boldsymbol{x}^l)F$ results in a matrix of size $(H^{l+1}W^{l+1}) \times D$. The vectorization of this resultant matrix generates a vector in $\mathbb{R}^{H^{l+1}W^{l+1}D}$, which matches the dimensionality of $\operatorname{vec}(\boldsymbol{y})$.

5.4 The Kronecker product

A short detour to the Kronecker product is needed to compute the derivatives.

Table 1: Variables, their sizes and meanings. Note that "alias" means a variable has a different name or can be reshaped into another form.

	Alias	Size & Meaning
X	$\boldsymbol{x}^l$	$H^l W^l \times D^l$, the input tensor
F	$\boldsymbol{f}$, $\boldsymbol{w}^l$	$HWD^l \times D$, D kernels, each $H \times W$ and D^l channels
Y	$\boldsymbol{y}$, $\boldsymbol{x}^{l+1}$	$H^{l+1}W^{l+1} \times D^{l+1}$, the output, $D^{l+1} = D$
$\phi(\boldsymbol{x}^l)$		$H^{l+1}W^{l+1} \times HWD^l$, the `im2row` expansion of $\boldsymbol{x}^l$
M		$H^{l+1}W^{l+1}HWD^l \times H^l W^l D^l$, the indicator matrix for $\phi(\boldsymbol{x}^l)$
$\frac{\partial z}{\partial Y}$	$\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{y})}$	$H^{l+1}W^{l+1} \times D^{l+1}$, gradient for $\boldsymbol{y}$
$\frac{\partial z}{\partial F}$	$\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{f})}$	$HWD^l \times D$, gradient to update the convolution kernels
$\frac{\partial z}{\partial X}$	$\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{x}^l)}$	$H^l W^l \times D^l$, gradient for $\boldsymbol{x}^l$, useful for back propagation

Given two matrices $A \in \mathbb{R}^{m\times n}$ and $B \in \mathbb{R}^{p\times q}$, the Kronecker product $A \otimes B$ is an $mp \times nq$ matrix, defined as a block matrix

$$A \otimes B = \left[\begin{array}{c|c|c} a_{11}B & \cdots & a_{1n}B \\ \hline \vdots & \ddots & \vdots \\ \hline a_{m1}B & \cdots & a_{mn}B \end{array}\right] . \tag{26}$$

The Kronecker product has the following properties that will be useful for us:

$$(A \otimes B)^T = A^T \otimes B^T \,, \tag{27}$$

$$\operatorname{vec}(AXB) = (B^T \otimes A) \operatorname{vec}(X) \,, \tag{28}$$

for matrices A, X, and B with proper dimensions (e.g., when the matrix multiplication AXB is defined.) Note that Equation 28 can be utilized from both directions.

With the help of $\otimes$, we can write down

$$\operatorname{vec}(\boldsymbol{y}) = \operatorname{vec}\left(\phi(\boldsymbol{x}^l)FI\right) = \left(I \otimes \phi(\boldsymbol{x}^l)\right) \operatorname{vec}(F) \,, \tag{29}$$

$$\operatorname{vec}(\boldsymbol{y}) = \operatorname{vec}\left(I\phi(\boldsymbol{x}^l)F\right) = (F^T \otimes I) \operatorname{vec}(\phi(\boldsymbol{x}^l)) \,, \tag{30}$$

where I is an identity matrix of proper size. In Equation 29, the size of I is determined by the number of columns in F; hence $I \in \mathbb{R}^{D\times D}$ in Equation 29. Similarly, in Equation 30, $I \in \mathbb{R}^{(H^{l+1}W^{l+1})\times(H^{l+1}W^{l+1})}$.

The derivation for gradient computation rules in a convolution layer involves many variables and notations. We summarize the variables used in this derivation in Table 1. Note that some of these notations will soon be introduced in later sections.

5.5 Backward propagation: updating the parameters

As previously mentioned, we need to compute two derivatives: $\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{x}^l)}$ and $\frac{\partial z}{\partial \operatorname{vec}(F)}$, where the first term $\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{x}^l)}$ will be used for backward propagation to the previous ($(l-1)$-th) layer, and the second term will determine how the parameters of the current (l-th) layer will be updated. A friendly reminder is to remember that $\boldsymbol{f}$, F, and $\boldsymbol{w}^i$ refer to the same thing (modulo reshaping of the vector or matrix or tensor). Similarly, we can reshape $\boldsymbol{y}$ into a matrix $Y \in \mathbb{R}^{(H^{l+1}W^{l+1})\times D}$; then $\boldsymbol{y}$, Y, and $\boldsymbol{x}^{l+1}$ refer to the same object (again modulo reshaping).

From the chain rule (Equation 11), it is easy to compute $\frac{\partial z}{\partial \operatorname{vec}(F)}$ as

$$\frac{\partial z}{\partial(\operatorname{vec}(F))^T} = \frac{\partial z}{\partial(\operatorname{vec}(Y)^T)} \frac{\partial \operatorname{vec}(\boldsymbol{y})}{\partial(\operatorname{vec}(F)^T)} . \tag{31}$$

The first term in the RHS is already computed in the $(l+1)$-th layer as (equivalently) $\frac{\partial z}{\partial(\operatorname{vec}(\boldsymbol{x}^{l+1}))^T}$. The second term, based on Equation 29, is pretty straightforward:

$$\frac{\partial \operatorname{vec}(\boldsymbol{y})}{\partial(\operatorname{vec}(F)^T)} = \frac{\partial\left(\left(I \otimes \phi(\boldsymbol{x}^l)\right) \operatorname{vec}(F)\right)}{\partial(\operatorname{vec}(F)^T)} = I \otimes \phi(\boldsymbol{x}^l) . \tag{32}$$

Note that we have used the fact $\frac{\partial X \boldsymbol{a}^T}{\partial \boldsymbol{a}} = X$ or $\frac{\partial X \boldsymbol{a}}{\partial \boldsymbol{a}^T} = X$ so long as the matrix multiplications are well defined. This equation leads to

$$\frac{\partial z}{\partial(\operatorname{vec}(F))^T} = \frac{\partial z}{\partial(\operatorname{vec}(\boldsymbol{y})^T)} (I \otimes \phi(\boldsymbol{x}^l)) . \tag{33}$$

Making a transpose, we get

$$\begin{aligned}
\frac{\partial z}{\partial \operatorname{vec}(F)} &= \left(I \otimes \phi(\boldsymbol{x}^l)\right)^T \frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{y})} && (34)\\
&= \left(I \otimes \phi(\boldsymbol{x}^l)^T\right) \operatorname{vec}\left(\frac{\partial z}{\partial Y}\right) && (35)\\
&= \operatorname{vec}\left(\phi(\boldsymbol{x}^l)^T \frac{\partial z}{\partial Y} I\right) && (36)\\
&= \operatorname{vec}\left(\phi(\boldsymbol{x}^l)^T \frac{\partial z}{\partial Y}\right) . && (37)
\end{aligned}$$

Note that both Equation 28 (from RHS to LHS) and Equation 27 are used in the above derivation.

Thus, we conclude that

$$\frac{\partial z}{\partial F} = \phi(\boldsymbol{x}^l)^T \frac{\partial z}{\partial Y} , \tag{38}$$

which is a simple rule to update the parameters in the l-th layer: the gradient with respect to the convolution parameters is the product between $\phi(\boldsymbol{x}^l)^T$ (the `im2col` expansion) and $\frac{\partial z}{\partial Y}$ (the supervision signal transferred from the $(l+1)$-th layer).

5.6 Even higher dimensional indicator matrices

The function $\phi(\cdot)$ has been very useful in our analysis. It is pretty high dimensional—e.g., $\phi(\boldsymbol{x}^l)$ has $H^{l+1}W^{l+1}HWD^l$ elements. From the above, we know that an element in $\phi(\boldsymbol{x}^l)$ is indexed by a pair p and q.

A quick recap about $\phi(\boldsymbol{x}^l)$: 1) from q we can determine d^l, which channel of the convolution kernel is used; and can also determine i and j, the spatial offsets inside the kernel; 2) from p we can determine i^{l+1} and j^{l+1}, the spatial offsets inside the convolved result $\boldsymbol{x}^{l+1}$; and, 3) the spatial offsets in the input $\boldsymbol{x}^l$ can be determined as $i^l = i^{l+1} + i$ and $j^l = j^{l+1} + j$.

That is, the mapping $m : (p, q) \mapsto (i^l, j^l, d^l)$ is one-to-one, and thus is a valid function. The inverse mapping, however, is one-to-many (thus not a valid function). If we use m^{-1} to represent the inverse mapping, we know that $m^{-1}(i^l, j^l, d^l)$ is a set S, where each $(p, q) \in S$ satisfies that $m(p, q) = (i^l, j^l, d^l)$.

Now we take a look at $\phi(\boldsymbol{x}^l)$ from a different perspective. In order to fully specify $\phi(\boldsymbol{x}^l)$, what information is required? It is obvious that the following three types of information are needed (and only those). For *every* element of $\phi(\boldsymbol{x}^l)$, we need to know

(A) Which region does it belong to—i.e., what is the value of p ($0 \le p < H^{l+1}W^{l+1}$)?

(B) Which element is it inside the region (or equivalently inside the convolution kernel)—i.e., what is the value of q ($0 \le q < HWD^l$)?

The above two pieces of information determine a location (p, q) inside $\phi(\boldsymbol{x}^l)$. The only missing information is

(C) What is the value in that position—i.e., $\left[\phi(\boldsymbol{x}^l)\right]_{pq}$?

Since every element in $\phi(\boldsymbol{x}^l)$ is a verbatim copy of one element from $\boldsymbol{x}^l$, we can turn [C] into a different but equivalent one:

(C.1) For $\left[\phi(\boldsymbol{x}^l)\right]_{pq}$, where is this value copied from? Or, what is its original location inside $\boldsymbol{x}^l$—i.e., an index u that satisfies $0 \le u < H^lW^lD^l$?

(C.2) The entire $\boldsymbol{x}^l$.

It is easy to see that the collective information in [A, B, C.1] (for the entire range of p, q and u), and [C.2] ($\boldsymbol{x}^l$) contains exactly the same amount of information as $\phi(\boldsymbol{x}^l)$.

Since $0 \le p < H^{l+1}W^{l+1}$, $0 \le q < HWD^l$, and $0 \le u < H^lW^lD^l$, we can use a matrix

$$M \in \mathbb{R}^{(H^{l+1}W^{l+1}HWD^l)\times(H^lW^lD^l)}$$

to encode the information in [A, B, C.1]. One row index of this matrix corresponds to one location inside $\phi(\boldsymbol{x}^l)$ (i.e., a (p, q) pair). One row of M has $H^lW^lD^l$ elements, and each element can be indexed by (i^l, j^l, d^l). Thus, each element in this matrix is indexed by a 5-tuple: (p, q, i^l, j^l, d^l).

Then, we can use the "indicator" method to encode the function $m(p,q) = (i^l, j^l, d^l)$ into M. That is, for any possible element in M, its row index x determines a (p,q) pair, and its column index y determines a (i^l, j^l, d^l) triplet, and M is defined as

$$M(x,y) = \begin{cases} 1 & \text{if } m(p,q) = (i^l, j^l, d^l) \\ 0 & \text{otherwise} \end{cases} . \tag{39}$$

The M matrix has the following properties:

- It is very high dimensional;
- But it is also very sparse: there is only 1 non-zero entry in the $H^l W^l D^l$ elements in one row, because m is a function;
- M, which uses information [A, B, C.1], only encodes the one-to-one correspondence between any element in $\phi(\boldsymbol{x}^l)$ and any element in $\boldsymbol{x}^l$; it does not encode any specific value in $\boldsymbol{x}^l$;
- Most importantly, putting together the one-to-one correspondence information in M and the value information in $\boldsymbol{x}^l$, obviously we have

$$\operatorname{vec}(\phi(\boldsymbol{x}^l)) = M \operatorname{vec}(\boldsymbol{x}^l) . \tag{40}$$

5.7 Backward propagation: preparing the supervision signal for the previous layer

In the l-th layer, we still need to compute $\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{x}^l)}$. For this purpose, we want to reshape $\boldsymbol{x}^l$ into a matrix $X \in \mathbb{R}^{(H^l W^l) \times D^l}$, and use these two equivalent forms (modulo reshaping) interchangeably.

The chain rule states that (cf. Equation 12)

$$\frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{x}^l)^T)} = \frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{y})^T)} \frac{\partial \operatorname{vec}(\boldsymbol{y})}{\partial (\operatorname{vec}(\boldsymbol{x}^l)^T)} .$$

We will start by studying the second term in the RHS (utilizing Equations 30 and 40):

$$\frac{\partial \operatorname{vec}(\boldsymbol{y})}{\partial (\operatorname{vec}(\boldsymbol{x}^l)^T)} = \frac{\partial (F^T \otimes I) \operatorname{vec}(\phi(\boldsymbol{x}^l))}{\partial (\operatorname{vec}(\boldsymbol{x}^l)^T)} = (F^T \otimes I) M . \tag{41}$$

Thus,

$$\frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{x}^l)^T)} = \frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{y})^T)} (F^T \otimes I) M . \tag{42}$$

Since (using Equation 28 from right to left)

$$\frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{y})^T)} (F^T \otimes I) = \left((F \otimes I) \frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{y})} \right)^T \tag{43}$$

$$= \left((F \otimes I) \operatorname{vec}\left(\frac{\partial z}{\partial Y} \right) \right)^T \tag{44}$$

$$= \operatorname{vec}\left(I \frac{\partial z}{\partial Y} F^T \right)^T \tag{45}$$

$$= \operatorname{vec}\left(\frac{\partial z}{\partial Y} F^T \right)^T , \tag{46}$$

we have

$$\frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{x}^l)^T)} = \operatorname{vec}\left(\frac{\partial z}{\partial Y} F^T \right)^T M , \tag{47}$$

or equivalently

$$\frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{x}^l))} = M^T \operatorname{vec}\left(\frac{\partial z}{\partial Y} F^T \right) . \tag{48}$$

Let's have a closer look at the RHS. $\frac{\partial z}{\partial Y} F^T \in \mathbb{R}^{(H^{l+1}W^{l+1}) \times (HWD^l)}$, and $\operatorname{vec}\left(\frac{\partial z}{\partial Y} F^T \right)$ is a vector in $\mathbb{R}^{H^{l+1}W^{l+1}HWD^l}$. On the other hand, M^T is an indicator matrix in $\mathbb{R}^{(H^l W^l D^l) \times (H^{l+1}W^{l+1}HWD^l)}$.

In order to pinpoint one element in $\operatorname{vec}(\boldsymbol{x}^l)$ or one row in M^T, we need an index triplet (i^l, j^l, d^l), with $0 \le i^l < H^l$, $0 \le j^l < W^l$, and $0 \le d^l < D^l$. Similarly, to locate a column in M^T or an element in $\frac{\partial z}{\partial Y} F^T$, we need an index pair (p, q), with $0 \le p < H^{l+1}W^{l+1}$ and $0 \le q < HWD^l$.

Thus, the (i^l, j^l, d^l)-th entry of $\frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{x}^l))}$ equals the multiplication of two vectors: the row in M^T (or the column in M) that is indexed by (i^l, j^l, d^l), and $\operatorname{vec}\left(\frac{\partial z}{\partial Y} F^T \right)$.

Furthermore, since M^T is an indicator matrix, in the row vector indexed by (i^l, j^l, d^l), only those entries whose index (p, q) satisfies $m(p, q) = (i^l, j^l, d^l)$ have a value 1; all other entries are 0. Thus, the (i^l, j^l, d^l)-th entry of $\frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{x}^l))}$ equals the sum of these corresponding entries in $\operatorname{vec}\left(\frac{\partial z}{\partial Y} F^T \right)$.

Transferring the above description into precise mathematical form, we get the following succinct equation:

$$\left[\frac{\partial z}{\partial X} \right]_{(i^l, j^l, d^l)} = \sum_{(p,q) \in m^{-1}(i^l, j^l, d^l)} \left[\frac{\partial z}{\partial Y} F^T \right]_{(p,q)} . \tag{49}$$

In other words, to compute $\frac{\partial z}{\partial X}$, we do not need to explicitly use the extremely high dimensional matrix M. Instead, Equation 49 and Equations 21 to 24 can be used to efficiently find $\frac{\partial z}{\partial X}$.

We use the simple convolution example in Figure 3 to illustrate the inverse mapping m^{-1}, which is shown in Figure 5.

In the right half of Figure 5, the 6×4 matrix is $\frac{\partial z}{\partial Y} F^T$. In order to compute the partial derivative of z with respect to one element in the input X, we need to find which elements in $\frac{\partial z}{\partial Y} F^T$ are involved and add them. In the left half of Figure 5, we show that the input element 5 (shown in larger font) is involved in four convolution operations, shown by the red, green, blue, and black boxes,

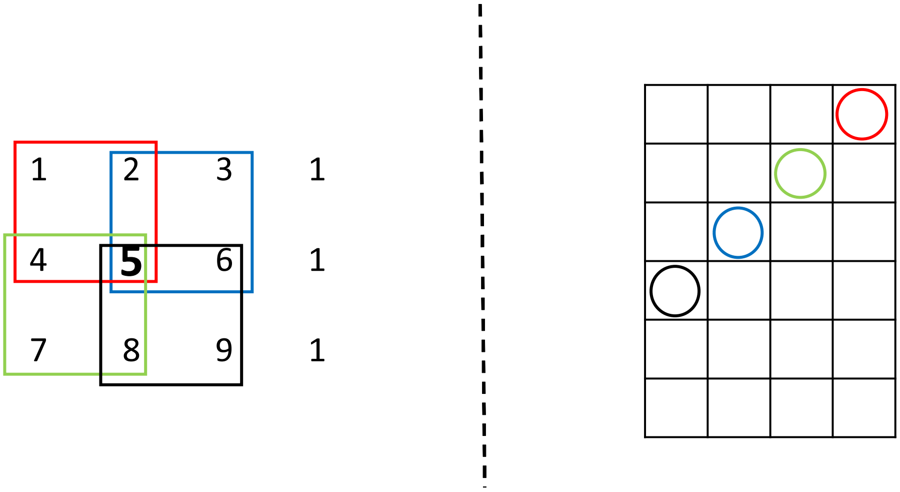


Figure 5: Illustration of how to compute $\frac{\partial z}{\partial X}$. ()

respectively. These four convolution operations correspond to $p = 1, 2, 3, 4$. For example, when $p = 2$ (the green box), 5 is the third element in the convolution; hence $q = 3$ when $p = 2$, and we put a green circle in the $(2, 3)$-th element of the $\frac{\partial z}{\partial Y} F^T$ matrix. After all four circles have been put in the $\frac{\partial z}{\partial Y} F^T$ matrix, the partial derivative is the sum of elements in these four locations of $\frac{\partial z}{\partial Y} F^T$.

The set $m^{-1}(i^l, j^l, d^l)$ contains at most HW elements. Hence, Equation 49 requires at most HW summations to compute one element of $\frac{\partial z}{\partial X}$.[3]

5.8 Fully connected layer as a convolution layer

As aforementioned, one benefit of the convolution layer is that convolution is a local operation. The spatial extent of a kernel is often small (e.g., 3×3). One element in $\boldsymbol{x}^{l+1}$ is usually computed using only a small number of elements in its input $\boldsymbol{x}^l$.

A fully connected layer refers to a layer where the computation of any element in the output $\boldsymbol{x}^{l+1}$ (or $\boldsymbol{y}$) requires all elements in the input $\boldsymbol{x}^l$. A fully connected layer is sometimes useful at the end of a deep CNN model. For example, if after many convolution, ReLU, and pooling (which will be discussed soon) layers, the output of the current layer contains distributed representations for the input image, we want to use all these features in the current layer to build features with stronger capabilities in the next one. A fully connected layer is useful for this purpose.

Suppose the input of a layer $\boldsymbol{x}^l$ has size $H^l \times W^l \times D^l$. If we use convolution kernels whose size is $H^l \times W^l \times D^l$, then D such kernels form an order 4 tensor

[3] In Caffe, this computation is implemented by a function called `col2im`. In MatConvNet, this operation is operated in a `row2im` manner, although the name `row2im` is not explicitly used.

with size $H^l \times W^l \times D^l \times D$. The output is $\boldsymbol{y} \in \mathbb{R}^D$. It is obvious that to compute any element in $\boldsymbol{y}$, we need to use all elements in the input $\boldsymbol{x}^l$. Hence, this layer is a fully connected layer, but can be implemented as a convolution layer. Hence, we do not need to derive learning rules for a fully connected layer separately.

6 The pooling layer

We will use the same notation inherited from the convolution layer to introduce pooling layers.

Let $\boldsymbol{x}^l \in \mathbb{R}^{H^l \times W^l \times D^l}$ be the input to the l-th layer, which is now a pooling layer. The pooling operation requires no parameter (i.e., $\boldsymbol{w}^i$ is null, hence parameter learning is not needed for this layer). The spatial extent of the pooling ($H \times W$) is specified in the design of the CNN structure. Assume that H divides H^l, W divides W^l, and the stride equals the pooling spatial extent;[4] the output of pooling ($\boldsymbol{y}$ or equivalently $\boldsymbol{x}^{l+1}$) will be an order 3 tensor of size $H^{l+1} \times W^{l+1} \times D^{l+1}$, with

$$H^{l+1} = \frac{H^l}{H}\,, \quad W^{l+1} = \frac{W^l}{W}\,, \quad D^{l+1} = D^l\,. \tag{50}$$

A pooling layer operates on $\boldsymbol{x}^l$ channel by channel independently. Within each channel, the matrix with $H^l \times W^l$ elements is divided into $H^{l+1} \times W^{l+1}$ nonoverlapping subregions, each subregion being $H \times W$ in size. The pooling operator then maps a subregion into a single number.

Two types of pooling operators are widely used: max pooling and average pooling. In max pooling, the pooling operator maps a subregion to its maximum value, while the average pooling maps a subregion to its average value. In precise mathematics,

$$\text{max}: \qquad y_{i^{l+1},j^{l+1},d} = \max_{0 \le i < H, 0 \le j < W} x^l_{i^{l+1} \times H + i, j^{l+1} \times W + j, d}\,, \tag{51}$$

$$\text{average}: \qquad y_{i^{l+1},j^{l+1},d} = \frac{1}{HW} \sum_{0 \le i < H, 0 \le j < W} x^l_{i^{l+1} \times H + i, j^{l+1} \times W + j, d}\,, \tag{52}$$

where $0 \le i^{l+1} < H^{l+1}$, $0 \le j^{l+1} < W^{l+1}$, and $0 \le d < D^{l+1} = D^l$.

Pooling is a local operator, and its forward computation is pretty straightforward. Now we focus on the back propagation. Only max pooling is discussed, and we can resort to the indicator matrix again. Average pooling can be dealt with using a similar idea.

All we need to encode in this indicator matrix is: for every element in $\boldsymbol{y}$, where does it come from in $\boldsymbol{x}^l$?

We need a triplet (i^l, j^l, d^l) to pinpoint one element in the input $\boldsymbol{x}^l$, and another triplet $(i^{l+1}, j^{l+1}, d^{l+1})$ to locate one element in $\boldsymbol{y}$. The pooling output $y_{i^{l+1},j^{l+1},d^{l+1}}$ comes from $x^l_{i^l,j^l,d^l}$, if and only if the following conditions are met:

[4] That is, the strides in the vertical and horizontal direction are H and W, respectively. The most widely used pooling setup is $H = W = 2$ with a stride of 2.

- They are in the same channel;
- The (i^l, j^l)-th spatial entry belongs to the (i^{l+1}, j^{l+1})-th subregion;
- The (i^l, j^l)-th spatial entry is the largest one in that subregion.

Translating these conditions into equations, we get

$$d^{l+1} = d^l \,, \tag{53}$$

$$\left\lfloor \frac{i^l}{H} \right\rfloor = i^{l+1}, \left\lfloor \frac{j^l}{W} \right\rfloor = j^{l+1} \,, \tag{54}$$

$$x^l_{i^l,j^l,d^l} \geq y_{i+i^{l+1}\times H, j+j^{l+1}\times W, d^l} \,, \forall\, 0 \leq i < H, 0 \leq j < W \,, \tag{55}$$

where $\lfloor \cdot \rfloor$ is the floor function. If the stride is not H (W) in the vertical (horizontal) direction, Equation 54 must be changed accordingly.

Given a $(i^{l+1}, j^{l+1}, d^{l+1})$ triplet, there is only one (i^l, j^l, d^l) triplet that satisfies all these conditions. Thus, we define an indicator matrix

$$S(\boldsymbol{x}^l) \in \mathbb{R}^{(H^{l+1}W^{l+1}D^{l+1}) \times (H^l W^l D^l)} \,. \tag{56}$$

One triplet of indexes $(i^{l+1}, j^{l+1}, d^{l+1})$ specifies a row in S, while (i^l, j^l, d^l) specifies a column. These two triplets together pinpoint one element in $S(\boldsymbol{x}^l)$. We set that element to 1 if Equations 53 to 55 are simultaneously satisfied, and 0 otherwise. One row of $S(\boldsymbol{x}^l)$ corresponds to one element in $\boldsymbol{y}$, and one column corresponds to one element in $\boldsymbol{x}^l$.

With the help of this indicator matrix, we have

$$\operatorname{vec}(\boldsymbol{y}) = S(\boldsymbol{x}^l) \operatorname{vec}(\boldsymbol{x}^l) \,. \tag{57}$$

Then, it is obvious that

$$\frac{\partial \operatorname{vec}(\boldsymbol{y})}{\partial (\operatorname{vec}(\boldsymbol{x}^l)^T)} = S(\boldsymbol{x}^l), \qquad \frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{x}^l)^T)} = \frac{\partial z}{\partial (\operatorname{vec}(\boldsymbol{y})^T)} S(\boldsymbol{x}^l) \,, \tag{58}$$

and consequently

$$\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{x}^l)} = S(\boldsymbol{x}^l)^T \frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{y})} \,. \tag{59}$$

$S(\boldsymbol{x}^l)$ is very sparse. It has exactly one nonzero entry in every row. Thus, we do not need to use the entire matrix in the computation. Instead, we just need to record the locations of those nonzero entries—there are only $H^{l+1}W^{l+1}D^{l+1}$ such entries in $S(\boldsymbol{x}^l)$.

A simple example can explain the meaning of these equations. Let us consider a 2×2 max pooling with stride 2. For a given channel d^l, the first spatial subregion contains four elements in the input, with $(i, j) = (0, 0)$, $(1, 0)$, $(0, 1)$, and $(1, 1)$, and let us suppose the element at spatial location $(0, 1)$ is the largest among them. In the forward pass, the value indexed by $(0, 1, d^l)$ in the input (i.e., $x^l_{0,1,d^l}$) will be assigned to the element in the $(0, 0, d^l)$-th element in the output (i.e., $y_{0,0,d^l}$).

If the strides are H and W, respectively, one column in $S(\boldsymbol{x}^l)$ contains at most one nonzero element. In the above example, the column of $S(\boldsymbol{x}^l)$ indexed by $(0,0,d^l)$, $(1,0,d^l)$, and $(1,1,d^l)$ are all zero vectors. The column corresponding to $(0,1,d^l)$ contains only one nonzero entry, whose row index is determined by $(0,0,d^l)$. Hence, in the back propagation, we have

$$\left[\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{x}^l)}\right]_{(0,1,d^l)} = \left[\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{y})}\right]_{(0,0,d^l)},$$

and

$$\left[\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{x}^l)}\right]_{(0,0,d^l)} = \left[\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{x}^l)}\right]_{(1,0,d^l)} = \left[\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{x}^l)}\right]_{(1,1,d^l)} = 0\,.$$

However, if the pooling strides are smaller than H and W in the vertical and horizontal directions, respectively, one element in the input tensor may be the largest element in several pooling subregions. Hence, there can be more than one nonzero entry in one column of $S(\boldsymbol{x}^l)$. Let us consider the example input in Figure 5. If a 2×2 max pooling is applied to it and the stride is 1 in both directions, the element `9` is the largest in two pooling regions: $\left[\begin{smallmatrix} 5 & 6 \\ 8 & 9 \end{smallmatrix}\right]$ and $\left[\begin{smallmatrix} 6 & 1 \\ 9 & 1 \end{smallmatrix}\right]$. Hence, in the column of $S(\boldsymbol{x}^l)$ corresponding to the element `9` (indexed by $(2,2,d^l)$ in the input tensor), there are two nonzero entries whose row indexes correspond to $(i^{l+1}, j^{l+1}, d^{l+1}) = (1,1,d^l)$ and $(1,2,d^l)$. Thus, in this example, we have

$$\left[\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{x}^l)}\right]_{(2,2,d^l)} = \left[\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{y})}\right]_{(1,1,d^l)} + \left[\frac{\partial z}{\partial \operatorname{vec}(\boldsymbol{y})}\right]_{(1,2,d^l)}.$$

7 A case study: the VGG-16 net

We have introduced the convolution, pooling, ReLU, and fully connected layers so far, and have briefly mentioned the softmax layer. With these layers, we can build many powerful deep CNN models.

7.1 VGG-Verydeep-16

The VGG-Verydeep-16 CNN model is a pretrained CNN model released by the Oxford VGG group.[5] We use it as an example to study the detailed structure of CNN networks. The VGG-16 model architecture is listed in Table 2.

There are six types of layers in this model.

Convolution A convolution layer is abbreviated as "Conv." Its description includes four parts: number of channels; kernel spatial extent (kernel size); padding ('p'); and stride ('st') size.

ReLU No description is needed for a ReLU layer.

[5] http://www.robots.ox.ac.uk/~vgg/research/very_deep/

Table 2: The VGG-Verydeep-16 architecture and receptive field

	type	description	r. size		type	description	r. size
1	Conv	64;3x3;p=1,st=1	212	20	Conv	512;3x3;p=1,st=1	20
2	ReLU		210	21	ReLU		18
3	Conv	64;3x3;p=1,st=1	210	22	Conv	512;3x3;p=1,st=1	18
4	ReLU		208	23	ReLU		16
5	Pool	2x2;st=2	208	24	Pool	2x2;st=2	16
6	Conv	128;3x3;p=1,st=1	104	25	Conv	512;3x3;p=1,st=1	8
7	ReLU		102	26	ReLU		6
8	Conv	128;3x3;p=1,st=1	102	27	Conv	512;3x3;p=1,st=1	6
9	ReLU		100	28	ReLU		4
10	Pool	2x2;st=2	100	29	Conv	512;3x3;p=1,st=1	4
11	Conv	256;3x3;p=1,st=1	50	30	ReLU		2
12	ReLU		48	31	Pool		2
13	Conv	256;3x3;p=1,st=1	48	32	FC	(7x7x512)x4096	1
14	ReLU		46	33	ReLU		
15	Conv	256;3x3;p=1,st=1	46	34	Drop	0.5	
16	ReLU		44	35	FC	4096x4096	
17	Pool	2x2;st=2	44	36	ReLU		
18	Conv	512;3x3;p=1,st=1	22	37	Drop	0.5	
19	ReLU		20	38	FC	4096x1000	
				39	σ	(softmax layer)	

Pool A pooling layer is abbreviated as "Pool." Only max pooling is used in VGG-16. The pooling kernel size is always 2×2 and the stride is always 2 in VGG-16.

Fully connected A fully connected layer is abbreviated as "FC." Fully connected layers are implemented using convolution in VGG-16. Its size is shown in the format $n_1 \times n_2$, where n_1 is the size of the input tensor, and n_2 is the size of the output tensor. Although n_1 can be a triplet (such as $7 \times 7 \times 512$), n_2 is always an integer.

Dropout A dropout layer is abbreviated as "Drop." Dropout is a technique to improve the generalization of deep learning methods. It sets the weights connected to a certain percentage of nodes in the network to 0 (and VGG-16 sets the percentage to 0.5 in the two dropout layers).

Softmax This is abbreviated as "σ."

We want to add a few notes about this example deep CNN architecture.

- A convolution layer is always followed by a ReLU layer in VGG-16. The ReLU layers increase the nonlinearity of the CNN model.

- The convolution layers between two pooling layers have the same number of channels, kernel size, and stride. In fact, stacking two 3×3 convolution layers is equivalent to one 5×5 convolution layer; and stacking three 3×3 convolution kernels replaces a 7×7 convolution layer. Stacking a few (2 or 3) smaller convolution kernels, however, computes faster than a large convolution kernel. In addition, the number of parameters is also reduced—e.g., $2 \times 3 \times 3 = 18 < 25 = 5 \times 5$. The ReLU layers inserted in between small convolution layers are also helpful.

- The input to VGG-16 is an image with size $224 \times 224 \times 3$. Because the padding is one in the convolution kernels (meaning one row or column is added outside of the four edges of the input), convolution will not change the spatial extent. The pooling layers will reduce the input size by a factor of 2. Hence, the output after the last (5th) pooling layer has spatial extent 7×7 (and 512 channels). We may interpret this tensor as $7 \times 7 \times 512 = 25088$ "features." The first fully connected layer converts these into 4096 features. The number of features remains at 4096 after the second fully connected layer.

- The VGG-16 is trained for the ImageNet classification challenge, which is an object recognition problem with 1000 classes. The last fully connected layer (4096×1000) outputs a length 1000 vector for every input image, and the softmax layer converts this length 1000 vector into the estimated posterior probability for the 1000 classes.

7.2 Receptive field

Another important concept in CNN is the receptive field size (abbreviated as "r. size" in Table 2). Let us look at one element in the input to the first fully connected layer (`32|FC`). Because it is the output of a max pooling, we need values in a 2×2 spatial extent in the input to the max pool layer to compute this element (and we only need elements in this spatial extent). This 2×2 spatial extent is called the *receptive field* for this element. In Table 2, we listed the spatial extent for any element in the output of the last pooling layer. Note that because the receptive field is square, we only use one number (e.g., 48 for 48×48). The receptive field size listed for one layer is the spatial extent in the input to that layer.

A 3×3 convolution layer will increase the receptive field by 2, and a pooling layer will double the spatial extent. As shown in Table 2, receptive field size in the input to the first layer is 212×212. In other words, in order to compute any single element in the $7 \times 7 \times 512$ output of the last pooling layer, a 212×212 image patch is required (including the padded pixels in all convolution layers).

It is obvious that the receptive field size increases when the network becomes deeper, especially when a pooling layer is added to the deep net. Unlike traditional computer vision and image processing features that depend only on a small receptive field (e.g., 16×16), deep CNN computes its representation (or features) using large receptive fields. The larger receptive field characteristic is an important reason why CNN has achieved higher accuracy than classic methods in image recognition.

8 Hands-on CNN experiences

We hope this introductory chapter on CNN is clear, self-contained, and easy to understand to our readers.

Once a reader is confident in his/her understanding of CNN at the mathematical level, in the next step it is very helpful to get some hands-on CNN experiences. For example, one can validate what has been talked about in this chapter using the MatConvNet software package if you prefer the Matlab environment.[6] For C++ lovers, Caffe is a widely used tool.[7] The Theano package is a python package for deep learning.[8] Many more resources for deep learning (not only CNN) are available—e.g., Torch,[9] PyTorch,[10] MXNet,[11] Keras,[12] TensorFlow,[13] and more. The exercise problems in this chapter are offered as appropriate first-time CNN programming practice.

[6] http://www.vlfeat.org/matconvnet/

[7] http://caffe.berkeleyvision.org/

[8] http://deeplearning.net/software/theano/

[9] http://torch.ch/

[10] http://pytorch.org/

[11] https://mxnet.incubator.apache.org/

[12] https://keras.io/

[13] https://www.tensorflow.org/

Exercises

1. Dropout is a very useful technique in training neural networks, which is proposed by Srivastava et al. in a paper titled "Dropout: A Simple Way to Prevent Neural Networks from Overfitting" in JMLR .[14] *Carefully* read this paper and answer the following questions (please organize your answer to every question in one brief sentence).

   (a) How does dropout operate during training?

   (b) How does dropout operate during testing?

   (c) What is the benefit of dropout?

   (d) Why can dropout achieve this benefit?

2. The VGG16 CNN model (also called VGG-Verydeep-16) was publicized by Karen Simonyan and Andrew Zisserman in a paper titled "Very Deep Convolutional Networks for Large-Scale Image Recognition" in the arXiv preprint server .[15] And, the GoogLeNet model was publicized by Szegedy et al. in a paper titled "Going Deeper with Convolutions" in the arXiv preprint server .[16] These two papers were publicized around the same time and share some similar ideas. *Carefully* read both papers and answer the following questions (please organize your answer to every question in one brief sentence).

   (a) Why do they use small convolution kernels (mainly 3×3) rather than larger ones?

   (b) Why are both networks quite deep (i.e., with many layers, around 20)?

   (c) What difficulties are caused by the large depth? How are they solved in these two networks?

3. Batch Normalization (BN) is another very useful technique in training deep neural networks, which is proposed by Sergey Ioffe and Christian Szegedy, in a paper titled "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift" in ICML 2015 .[17] *Carefully* read this paper and answer the following questions (please organize your answer to every question in one brief sentence).

   (a) What is internal covariate shift?

   (b) How does BN deal with this?

   (c) How does BN operate in a convolution layer?

   (d) What is the benefit of using BN?

[14] Available at http://jmlr.org/papers/v15/srivastava14a.html

[15] Available at https://arxiv.org/abs/1409.1556, later published in ICLR 2015 as a conference track paper.

[16] Available at https://arxiv.org/abs/1409.4842, later published in CVPR 2015.

[17] Available at http://jmlr.org/proceedings/papers/v37/ioffe15.pdf

4. ResNet is a very deep neural network learning technique proposed by He et al. in a paper titled "Deep Residual Learning for Image Recognition" in CVPR 2016 .[18] *Carefully* read this paper and answer the following questions (please organize your answer to every question in one brief sentence).

   (a) Although VGG16 and GoogLeNet have encountered difficulties in training networks of around 20–30 layers, what enables ResNet to train networks as deep as 1000 layers?

   (b) VGG16 is a feed-forward network, where each layer has only one input and only one output. Conversely, GoogLeNet and ResNet are DAGs (directed acyclic graphs), where one layer can have multiple inputs and multiple outputs, so long as the data flow in the network structure does *not* form a cycle. What is the benefit of DAGs vs. feed-forward?

   (c) VGG16 has two fully connected layers (fc6 and fc7), while ResNet and GoogLeNet do not have fully connected layers (except the last layer for classification). What is used to replace FC layers in them? What is the benefit?

5. *AlexNet* refers to the deep convolutional neural network trained on the ILSVRC challenge data, which is a groundbreaking work of deep CNN for computer vision tasks. The technical details of AlexNet are reported in the paper "ImageNet Classification with Deep Convolutional Neural Networks" by Alex Krizhevsky, Ilya Sutskever and Geoffrey E. Hinton in NIPS 25 .[19] It proposed the ReLU activation function and creatively used GPUs to accelerate the computations. *Carefully* read this paper and answer the following questions (please organize your answer to every question in one brief sentence).

   (a) Describe your understanding of how ReLU helps its success? And, how do the GPUs help out?

   (b) Using the average of predictions from several networks helps reduce the error rates. Why?

   (c) Where is the dropout technique applied? How does it help? And what is the cost of using dropout?

   (d) How many parameters are there in AlexNet? Why is the dataset size (1.2 million images) important for the success of AlexNet?

6. We will try different CNN structures on the MNIST dataset. We denote the "baseline" network in the MNIST example in MatConvNet as BASE in this question.[20] In this question, a convolution layer is denoted as

[18] Available at https://arxiv.org/pdf/1512.03385.pdf

[19] This paper is available at http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks

[20] MatConvNet version 1.0-beta20. Please refer to MatConvNet for all the details of BASE, such as parameter initialization and learning rate.

"$x \times y \times \text{nIn} \times \text{nOut}$," whose kernel size is $x \times y$, with nIn input and nOut output channels, with stride 1 and pad 0. The pooling layers are 2×2 max pooling with stride equal 2. The BASE network has four blocks. The first consists of a $5 \times 5 \times 1 \times 20$ convolution and a max pooling; the second block is composed of a $5 \times 5 \times 20 \times 50$ convolution and a max pooling; the third block is a $4 \times 4 \times 50 \times 500$ convolution (FC) plus a ReLU layer; and the final block is the classification layer ($1 \times 1 \times 500 \times 10$ convolution).

(a) The MNIST dataset is available at http://yann.lecun.com/exdb/mnist/. Read the instructions on that page, and write a program to transform the data to formats that suit your favorite deep learning software.

(b) Learning deep learning models often involves random numbers. Before the training starts, set the random number generator's seed to 0. Then, use the BASE network structure and the *first 10000 training examples* to learn its parameters. What is the test set error rate (on the 10000 test examples) after 20 training epochs?

(c) From now on, if not otherwise specified, we assume the first 10000 training examples and 20 epochs are used. Now we define the BN network structure, which adds a batch normalization layer after every convolution layer in the first three blocks. What is its error rate? What can you say about BN vs. BASE?

(d) If you add a dropout layer after the classification layer in the 4th block, what is the new error rate of BASE and BN? Comment on the use of dropout?

(e) Now we define the SK network structure, which refers to small kernel size. SK is based on BN. The first block (5×5 convolution plus pooling) is now changed to two 3×3 convolutions, and BN + ReLU is applied after every convolution. For example, block 1 is now $3 \times 3 \times 1 \times 20$ convolution + BN + ReLU + $3 \times 3 \times 20 \times 20$ convolution + BN + ReLU + pool. What is SK's error rate? Comment on that (e.g., how and why the error rate changes).

(f) Now we define the SK-s network structure. The notation 's' refers to a multiplier that changes the number of channels in convolution layers. For example, SK is the same as SK-1. And, SK-2 means the number of channels in all convolution layers (except the one in block 4) are multiplied by 2. Train networks for SK-2, SK-1.5, SK-1, SK-0.5, and SK-0.2. Report their error rates and comment on them.

(g) Now we experiment with different training set sizes using the SK-0.2 network structure. Using the first 500, 1000, 2000, 5000, 10000, 20000, and 60000 (all) training examples, what error rates do you achieve? Comment on your observations.

(h) Using the SK-0.2 network structure, study how different training sets affect its performance. Train 6 networks, and use the $(10000 \times (i-1)+1)$-

th to ($i \times 10000$)-th training examples in training the i-th network. Are CNNs stable in terms of different training sets?

(i) Now we study how randomness affects CNN learning. Instead of setting the random number generator's seed to 0, use 1, 12, 123, 1234, 12345, and 123456 as the seed to train 6 different SK-0.2 networks. What are their error rates? Comment on your observations.

(j) Finally, in SK-0.2, change all ReLU layers to sigmoid layers. Comment on the comparison of error rates between using ReLU and sigmoid activation functions.